

Ashok B. Mehta

# Introduction to SystemVerilog



# Introduction to SystemVerilog

Ashok B. Mehta

# Introduction to SystemVerilog



Springer

Ashok B. Mehta  
DefineView Consulting  
Los Gatos, CA, USA

ISBN 978-3-030-71318-8      ISBN 978-3-030-71319-5 (eBook)  
<https://doi.org/10.1007/978-3-030-71319-5>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG  
The registered company address is: Gwerbestrasse 11, 6330 Cham, Switzerland

*To  
my dear wife, Ashraf Zahedi  
and  
my dear parents, Rukshamani Behn  
and Babubhai Mehta*

# Foreword

Hardware Description and Verification Languages (HDLs and HVLs) have always been difficult topics to absorb. When Ashok Mehta and I started our engineering careers in the early 1980s (he at Digital Equipment and I at rival Data General), digital hardware design was all about individual logic gates, timing analysis, and maybe power consumption. You were mostly above the level of dealing with transistors unless you were doing analog design. You could get by with little or no knowledge of software programming concepts. Hardware verification was accomplished by building massively sized physical prototypes, applying power to them, and hoping they didn't start smoking.

As Gordon Moore keenly observed more than a decade earlier, the logic density on integrated circuits had been doubling every 2 years. By the end of the 1980s, you could no longer deal with individual logic gates. HDLs together with simulation and synthesis technology helped raise the level of abstraction so that a single line of code could now represent hundreds to even hundreds of thousands of logic gates. But the true advancement came from having the same HDL description used as the source for both simulation and synthesis. Verilog was one of the first HDLs designed with those capabilities in mind, as well as providing a description for a testbench for verification merging HDL and HVL together.

Digital hardware can be thought of at the lowest level as a massive number of concurrent processes with each logic gate being an independent process. Depending on the design, each process may range from highly repetitive to highly configurable by different sets of parameters. These processes communicate with each other through networks of wires transmitting ones and zeros. Describing this behavior is one aspect of an HDL that sets it apart from most other software programming languages. Another unique HDL aspect is dealing with structures of many different bit sizes as opposed to fixed sizes like 8-bit bytes and 32/64-bit words. The testbench that generates stimulus to your design and analyzes its output needs to rely on these features as well. Remember that although you can raise the abstraction level of the internals of your design as much as you want, the boundary of your design with the testbench remains at the lowest level, that of wires sending ones and zeros.

In the 1990s, a number of strategies for raising the abstraction level of the test-bench not encumbered by synthesis restrictions came into practice, which diverged HVL from HDL. These HVLs integrated many concepts from software languages like object-oriented programming (OOP), dynamically sized arrays, queues, and dynamically created processes. They also added concepts strictly for verification like assertions, constrained random stimulus, and functional coverage. SystemVerilog was the result of an industry initiative to merge HDL and HVL back together giving hardware design and verification engineers a single type system, a unified set of rules for expression evaluation, and unified simulation execution semantics.

Pulling everything together into a single language has been quite a challenge, but it's even a bigger challenge to learn the language without any of the history behind it; plus, it's just a huge amount of material. Some features had value in the original HVL they came from, but once merged no longer make as much sense (i.e., wildcard associative array indexes and program blocks). Much of the material for learning SystemVerilog assumes a lot of prior knowledge, and a language reference manual (LRM) usually cannot remove any prior features for legacy reasons. Of course, the LRM was never intended to be written as a tutorial. In this book, Ashok addresses the challenge of learning SystemVerilog from scratch based on his deep experience with design and verification.

As I mentioned earlier, Ashok and I both started our career paths around the same time and our paths intertwined frequently. Eventually my path put me on teams that developed the Verilog and SystemVerilog languages, while Ashok's path put him on teams that were using it. He understands which topics are relevant to most users. Ashok uses this experience in his book to explain SystemVerilog from a broad user's perspective. He does not get bogged down trying to explain every minute detail of the language. Of course, that means skimming over parts that others might have included, and vice versa. This book along with its companion set of examples makes this excellent reading for anyone starting from the beginning or diving into the middle of an existing SystemVerilog design or verification project.

Dave Rich  
Verification Methodology Architect  
Mentor, A Siemens Business

# Preface

SystemVerilog is the IEEE standard language used today for design and verification of ASICs and FPGAs. It is a comprehensive language with features for RTL design (HDL), verification at transaction level with high level modular testbench development (object-oriented programming (OOP)), temporal/sequential domain checking using SystemVerilog Assertions, and coverage using SystemVerilog Functional Coverage. Four distinct languages, namely, SystemVerilog for design, SystemVerilog for verification, SystemVerilog Assertions, and SystemVerilog Functional Coverage are all rolled into the unified IEEE standard SystemVerilog language.

There are many good books available on SystemVerilog. Some books focus on SystemVerilog for Design while others focus on SystemVerilog for Verification. There is also very good subject matter available on the web. However, there is not a single book that covers the entire language. That is the motivation for this book. The book covers the entire language (except for features such as PLI/DPI, Gate level and Specify Block). The intent is to provide good fundamental information on each feature of the language with plenty of examples. Each feature is explained with simulatable examples and simulation logs to let the reader soak up the subject matter. Once the fundamentals are clear, the readers can then pursue further complexity of the features from various sources.

This is fundamentally a reference book that explores features of the language at its core syntax/semantic level. It will be useful to both a newcomer to SystemVerilog as well as an experienced user who wants to refresh on a certain topic.

The examples presented in the book can be downloaded from [www.defineview.com](http://www.defineview.com). On top of the page, you will see a link called “SystemVerilog Book Examples.” Click on it and it will download a .zip file which has the examples. Examples are categorized by chapter headings.

## Recommended Books to Supplement the Material Presented in this Book

SystemVerilog for Verification. Third Edition : A Guide to Learning the Testbench Language Features. By [Chris Spear, Greg Tumbush](#)

SystemVerilog for Design. Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling. By [Stuart Sutherland, Simon Davidmann, Peter Flake](#)

SystemVerilog Assertions and Functional Coverage. Third Edition: A Guide to Language, Methodology and Applications. By Ashok B. Mehta

## Chapters of the Book

### Chapter 1: Introduction

This chapter introduces evolution of the IEEE standard SystemVerilog language. It describes how the language subsets such as SystemVerilog for Design and Verification, SystemVerilog Assertions, and SystemVerilog Functional Coverage fold into a single unified language.

### Chapter 2: Data Types

This chapter describes the rich set of data types that SystemVerilog offers. Integer datatypes, Real datatypes and Net datatypes are discussed. In addition, user-defined types, static, local, automatic and global variables, enumerated types, string data types, and event data types are discussed. Each data type is explained with examples and simulation logs.

### Chapter 3: Arrays

This chapter introduces arrays offered by the language. Specifically, packed, unpacked, associative, and dynamic arrays are discussed. Array assignment, indexing, slicing, array manipulation methods, and array ordering methods are also discussed.

### Chapter 4: Queues

This chapter explores nuances of SystemVerilog “queues,” including queue methods, queue of queues, dynamic array of queues, etc.

### Chapter 5: Structures

This chapter discusses nuances of SystemVerilog structures, including packed structures, unpacked structures, structure within a structure, structure as module I/O, etc.

### Chapter 6: Union

This chapter discusses Unions. Packed, unpacked, and tagged unions.

## **Chapter 7: Packages**

This chapter discusses nuances of packages, how to declare and share them, how to reference data within a package, etc.

## **Chapter 8: Class**

This chapter delves into the detail of SystemVerilog “class,” which is the gist of object-oriented programming (OOP). It covers all aspects of a “class,” including polymorphism, shallow copy/deep copy, parameterized classes, upcasting/down-casting, virtual methods, virtual classes, etc.

## **Chapter 9: SystemVerilog “module”**

This chapter explores the nuances of SystemVerilog “module,” a fundamental building block of SystemVerilog. It includes, module headers (ansi, non-ansi), module parameters, localparams, nested modules, etc.

## **Chapter 10: SystemVerilog “program”**

This chapter describes a SystemVerilog “program” and its differentiation with SystemVerilog “module.” Among other aspects, it describes how race can be avoided between a testbench and a DUT.

## **Chapter 11: SystemVerilog “interface”**

This chapter discusses nuances of SystemVerilog “interface,” including modports (import/export), tasks/functions in an interface, parameterized interfaces, etc.

## **Chapter 12: Operators**

This chapter describes all available operators of the language, including assignment, increment/decrement, arithmetic, relational, equality, logical, bitwise, shift, conditional, concatenation, replication, streaming, wildcard equality, unary reduction, etc.

## **Chapter 13: Constraint Random Test Generation and Verification**

This chapter describes the constrained random verification methodology and discusses how to generate constrained random values and use them effectively for functional verification. The chapter discusses, among other things, rand/randc variables, randomization of arrays and queues, constraint blocks, weighted distribution, iterative constraints, soft constraints, randomization methods, system functions/methods, random stability, randcase/randsequence, productions, etc.

## **Chapter 14: SystemVerilog Assertions**

This chapter explores SystemVerilog Assertions (SVA). It discusses SVA methodology, immediate/deferred assertions, concurrent assertions and its operators, property, sequence, multi-threading, multi-clock properties, “bind,” sampled value functions, global clocking past/future functions, abort properties, etc.

## **Chapter 15: SystemVerilog Functional Coverage**

This chapter explores SystemVerilog functional coverage in detail. It discusses methodology components, covergroups, coverpoint, various types of “bins,” including, binsof, intersect, cross, transition, wildcard, ignore\_bins, illegal\_bins, etc. It also discusses sample/strobe methods and ways to query coverage.

## **Chapter 16: SystemVerilog Processes**

This chapter discusses SystemVerilog processes such as “initial”, “always”, “always\_ff”, “always\_latch”, “always\_comb”, “always @\*”. It also discusses fork-join, fork-join\_any, fork-join\_none, level sensitive time control, named event time control, etc.

## **Chapter 17: Procedural Programming Statements**

These chapter discusses procedural programming statements, such as if-else-if, case/casex/casez, unique-if, priority-if, unique-case, priority-case, loop statements, jump statements, etc.

## **Chapter 18: Interprocess Synchronization**

This chapter discusses interprocess synchronization mechanisms such as Semaphores and Mailboxes, including semaphore and mailbox methods, parameterized mailbox, etc.

## **Chapter 19: Clocking Blocks**

This chapter discusses the finer nuances of clocking blocks, including clocking blocks with interfaces, global clocking, etc.

## **Chapter 20: Checkers**

In this chapter we explore the construct of a “checker,” which allows one to group several assertions in a bigger block with its well-defined functionality and interfaces providing modularity and reusability. The “checker” is a powerful way to design modular and reusable code. Nested checkers, formal/actual arguments, and checkers in a package are discussed.

## **Chapter 21: “let” Declaration**

This chapter delves into the detail of “let” declaration. “let” declarations have local scope in contrast with `define, which has a global scope. A “let” declaration defines a template expression (a let body), customized by its ports (aka parameters). “let” with parameters, “let” in immediate and concurrent assertions, are also discussed.

## **Chapter 22: Tasks and Functions**

This chapter discusses SystemVerilog “tasks” and “functions,” including static/automatic tasks and functions, parameterized tasks/functions, etc. Argument passing and argument binding are also discussed.

## **Chapter 23: Procedural and Continuous Assignments**

This chapter will delves into the nuances of procedural and continuous assignments. It discusses features such as blocking and non-blocking procedural assignment, assign/deassign, force-release, etc.

## **Chapter 24: Utility System Tasks and Functions**

SystemVerilog offers a multitude of utility system tasks and functions. This chapter discusses, simulation control system tasks, simulation time system functions, timescale system functions, conversion functions, array querying system functions, math functions, bit-vector functions, severity system tasks, random and probabilistic distribution functions, queue management stochastic analysis tasks, etc.

**Chapter 25: I/O System Tasks and Functions**

This chapter discusses display tasks, file I/O tasks and functions, memory load tasks, memory dump tasks, command line input, and VCD tasks.

**Chapter 26: Generate Construct**

Generate blocks allow creating structural level code. The chapter discusses nuances of generate constructs, including loop constructs and conditional constructs.

**Chapter 27: Compiler Directives**

SystemVerilog offers a multitude of compiler directives to steer the course of your code. The chapter discusses, `define, `ifdef, `elsif, `ifndef, `timescale, `default\_net-type, etc.

Los Gatos, CA, USA

Ashok B. Mehta

# Acknowledgments

I am incredibly grateful to many who helped with review and editing of the book. In particular to:

**Vijay Akkati (QUALCOMM)** for review of the book chapters  
**Dr. Sandeep Goel (TSMC)** for motivation and encouragement  
**Bob Slee (EDA Direct)** for his sustained support facilitating close cooperation with the EDA vendors  
**Frank Lee (TSMC)** for his help and guidance through my career

Special gratitude to

**Dave Rich (Seimens)**  
**Mark Glasser (NVIDIA)**

Dave and Mark's help with in-depth review and editing of the chapters along with providing many good suggestions on improving the quality and structure of the book has been immense in the viability of the book. Their effort has made the book that much more robust.

Thanks also to Duolos for providing the EDA Playground platform, which was used heavily in the simulation of the examples in the book.

And last but certainly not the least, I would like to thank my wife Ashraf Zahedi for her enthusiasm and encouragement throughout the writing of this book and putting up with long nights and weekends required to finish the book. She is the cornerstone of my life, always with a positive attitude to carry the day through the ups and downs of life.

# Contents

<b>1</b>	<b>Introduction . . . . .</b>	<b>1</b>
1.1	SystemVerilog Language Evolution . . . . .	2
<b>2</b>	<b>Data Types . . . . .</b>	<b>5</b>
2.1	Integer Data Types . . . . .	5
2.1.1	Integer, int, longint, shortint, logic, byte, reg . . . . .	6
2.1.2	Signed Types . . . . .	8
2.1.3	Bits vs. Bytes . . . . .	10
2.2	Real Data Types . . . . .	10
2.2.1	“real” Data-Type Conversion Functions . . . . .	11
2.3	Nets . . . . .	12
2.3.1	“wire” and “tri” . . . . .	13
2.3.2	Unresolved “wire” Type: “uwire” . . . . .	15
2.3.3	Resolved vs. Unresolved Type . . . . .	16
2.3.4	“wand” and “triand” . . . . .	17
2.3.5	“wor” and “trior” . . . . .	17
2.3.6	“tri0” and “tri1” . . . . .	17
2.4	Drive Strengths . . . . .	19
2.5	Variable vs. Net . . . . .	22
2.6	“var” . . . . .	23
2.7	Variable and Net Initialization . . . . .	23
2.8	Static, Automatic, and Local Variables . . . . .	25
2.8.1	Static vs. Local Variables . . . . .	25
2.8.2	Automatic vs. Static Variable . . . . .	27
2.8.3	Variable Lifetimes . . . . .	29
2.9	Enumerated Types . . . . .	30
2.9.1	Enumerated-Type Methods . . . . .	32
2.9.2	Enumerated Type with Ranges . . . . .	36
2.10	User-Defined Type: Typedef . . . . .	38

2.11	String Data Type .....	40
2.11.1	String Operators.....	41
2.11.2	String Methods.....	44
2.12	Event Data Type.....	47
2.12.1	Event Sequencing: <code>wait_order()</code> .....	50
2.13	Static Casting .....	51
2.13.1	Bit-Stream Casting.....	56
2.14	Dynamic Casting .....	57
<b>3</b>	<b>Arrays.....</b>	<b>61</b>
3.1	Packed and Unpacked Arrays .....	61
3.1.1	2-D Packed Array.....	62
3.1.2	3-D Packed Array.....	63
3.1.3	1-D Packed and 1-D Unpacked Array .....	65
3.1.4	4-D Unpacked Array .....	67
3.1.5	1-D Packed and 3-D Unpacked Array .....	68
3.1.6	2-D Packed and 2D-Unpacked Array.....	69
3.1.7	3-D Packed and 1-D Unpacked Array .....	71
3.2	Assigning, Indexing, and Slicing of Arrays.....	72
3.2.1	Packed and Unpacked Arrays as Arguments to Subroutines .....	74
3.3	Dynamic Arrays .....	74
3.3.1	Dynamic Arrays – Resizing .....	75
3.3.2	Copying of Dynamic Arrays .....	79
3.3.3	Dynamic Array of Arrays.....	80
3.4	Associative Arrays .....	83
3.4.1	Wild Card Index.....	84
3.4.2	String Index .....	85
3.4.3	Class Index.....	85
3.4.4	String Index – Example .....	86
3.4.5	Associative Array Methods .....	87
3.4.6	Associative Array – Default Value.....	90
3.4.7	Creating a Dynamic Array of Associative Arrays .....	91
3.5	Array Manipulation Methods.....	92
3.5.1	Array Locator Methods .....	92
3.5.2	Array Ordering Methods .....	99
3.5.3	Array Reduction Methods .....	100
<b>4</b>	<b>Queues .....</b>	<b>105</b>
4.1	Queue Methods .....	109
4.2	Queue of SystemVerilog Classes .....	117
4.3	Queue of Queues: Dynamic Array of Queues .....	118
<b>5</b>	<b>Structures.....</b>	<b>121</b>
5.1	Packed Structure .....	122
5.2	Unpacked Structure .....	126
5.3	Structure as Module I/O.....	128

5.4	Structure as an Argument to Task or Function. . . . .	130
5.5	Structure Within a Structure. . . . .	131
<b>6</b>	<b>Union . . . . .</b>	<b>133</b>
6.1	Packed and Unpacked Unions . . . . .	135
6.1.1	Unpacked Unions. . . . .	136
6.1.2	Tagged Unions . . . . .	140
6.1.3	Packed Union. . . . .	142
<b>7</b>	<b>Packages. . . . .</b>	<b>145</b>
<b>8</b>	<b>Class . . . . .</b>	<b>155</b>
8.1	Basics. . . . .	155
8.2	Base Class . . . . .	156
8.3	Extended Class and Inheritance. . . . .	162
8.3.1	Inheritance Memory Allocation. . . . .	167
8.4	Class Constructor. . . . .	169
8.4.1	Base Class Constructor . . . . .	169
8.4.2	Chain Constructor ( super.new()). . . . .	169
8.5	Static Properties. . . . .	172
8.6	Static Methods . . . . .	176
8.7	“this” . . . . .	180
8.8	Class Assignment. . . . .	184
8.9	Shallow Copy. . . . .	186
8.10	Deep Copy . . . . .	189
8.11	Upcasting and Downcasting. . . . .	194
8.12	Polymorphism . . . . .	198
8.13	Virtual Methods, Pure Virtual Methods, and Virtual Classes. . . . .	203
8.13.1	Virtual Methods . . . . .	203
8.13.2	Virtual (Abstract) Class and Pure Virtual Method. . . . .	205
8.14	Data Hiding (“local,” “protected,” “const”) and Encapsulation. . . . .	211
8.14.1	Local Members . . . . .	212
8.14.2	Protected Members . . . . .	215
8.14.3	“const” Class Properties. . . . .	217
8.15	Class Scope Resolution Operator ( ::) and “extern” . . . . .	221
8.16	Parameterized Class. . . . .	225
8.16.1	Value Parameters . . . . .	226
8.16.2	Type Parameters. . . . .	227
8.16.3	Parameterized Class with Static Properties . . . . .	229
8.16.4	Extending Parameterized Class . . . . .	232
8.17	Difference Between Class and Struct. . . . .	233
<b>9</b>	<b>SystemVerilog “module” . . . . .</b>	<b>235</b>
9.1	Module Header Definition . . . . .	236
9.1.1	ANSI-Style Module Header. . . . .	237
9.1.2	ANSI-Style <i>first</i> Port Rules . . . . .	239
9.1.3	ANSI-Style <i>subsequent</i> Port Rules . . . . .	240

9.1.4	Non-ANSI-Style Module Header .....	241
9.2	Default Port Values .....	245
9.3	Module Instantiation .....	247
9.3.1	\$root .....	249
9.4	Nested Modules .....	249
9.5	Module Parameters .....	250
9.5.1	Overriding Module Parameters: <i>defparam</i> .....	251
9.5.2	Overriding Module Parameters: <i>Module Instance Parameter Value Assignment</i> .....	252
9.5.3	Localparam .....	253
9.5.4	Parameter Dependence .....	254
<b>10</b>	<b>SystemVerilog “program”</b> .....	257
10.1	Eliminating Testbench Races .....	261
<b>11</b>	<b>SystemVerilog “interface”</b> .....	267
11.1	Interfaces .....	267
11.2	Modports .....	274
11.3	Tasks and Functions in an Interface .....	278
11.3.1	“import” Tasks in a Modport .....	280
11.3.2	“export” Tasks in a Modport .....	282
11.4	Parameterized Interface .....	283
11.5	Clocking Block in an Interface .....	286
<b>12</b>	<b>Operators</b> .....	289
12.1	Assignment Operators .....	289
12.2	Increment and Decrement Operators .....	290
12.3	Arithmetic Operators .....	292
12.4	Relational Operators .....	296
12.5	Equality Operators .....	298
12.5.1	Wildcard Equality Operators .....	300
12.6	Logical Operators .....	301
12.7	Bitwise Operators .....	303
12.8	Unary Reduction Operators .....	305
12.9	Shift Operators .....	307
12.10	Conditional Operators .....	308
12.11	Concatenation Operators .....	310
12.12	Replication Operators .....	311
12.13	Streaming Operators (pack/unpack) .....	312
12.13.1	Packing of Bits .....	314
12.13.2	Unpacking of Bits .....	318
12.14	“inside” Operator (Set Membership Operator) .....	321
<b>13</b>	<b>Constrained Random Test Generation and Verification</b> .....	325
13.1	Productivity Gain with CRV .....	326
13.2	Constrained Random Verification (CRV) Methodology .....	327
13.3	SystemVerilog Support for CRV .....	328

13.4	Constraints . . . . .	329
13.4.1	Constraints: Turning On and OFF . . . . .	333
13.5	Random Variables (rand and randc) . . . . .	337
13.5.1	Static Random Variables . . . . .	339
13.5.2	Randomizing Arrays and Queues . . . . .	340
13.5.3	Randomizing Object Handles . . . . .	342
13.6	Constraint Blocks . . . . .	343
13.6.1	External Constraint Blocks . . . . .	343
13.6.2	Weighted Distribution . . . . .	345
13.6.3	“unique” Constraint . . . . .	348
13.6.4	Implication and If-Else . . . . .	351
13.6.5	Iterative Constraint ( <i>foreach</i> ) . . . . .	352
13.6.6	Array Reduction Methods for Constraint . . . . .	355
13.6.7	Functions in Constraints . . . . .	356
13.6.8	Soft Constraints . . . . .	357
13.7	Randomization Methods . . . . .	361
13.7.1	Pre-randomization and Post-randomization . . . . .	362
13.7.2	Local Scope Resolution (local::) . . . . .	367
13.8	rand_mode(): Disabling Random Variables . . . . .	367
13.9	constraint_mode(): Control Constraints . . . . .	370
13.10	randomize() with Arguments: In-Line Random Variable Control . . . . .	372
13.11	Random Number Generation System Functions and Methods . . . . .	376
13.11.1	Random Number Generator (RNG) . . . . .	376
13.11.2	\$urandom() and \$urandom_range() . . . . .	377
13.11.3	srandom(), get_randstate(), and set_randstate() . . . . .	380
13.12	Random Stability . . . . .	385
13.13	Randcase . . . . .	389
13.14	randsequence . . . . .	391
13.14.1	Random Production Weights and If-Else Statement . . . . .	393
13.14.2	Repeat Production Statement . . . . .	395
13.14.3	rand join . . . . .	397
13.14.4	break and return . . . . .	401
13.14.5	Passing Values Between Productions . . . . .	404
<b>14</b>	<b>SystemVerilog Assertions . . . . .</b>	<b>409</b>
14.1	SystemVerilog Assertions Evolution . . . . .	410
14.2	What Is an Assertion? . . . . .	410
14.3	Why Assertions? What Are the Advantages? . . . . .	411
14.3.1	Assertions Shorten Time to Develop . . . . .	412
14.3.2	Assertions Improve Observability . . . . .	413
14.3.3	Other Major Benefits . . . . .	414
14.3.4	One-Time Effort, Many Benefits . . . . .	415
14.4	Assertions in Static Formal . . . . .	416

14.5	Methodology Components .....	417
14.5.1	Types of Assertions to Add .....	417
14.5.2	How to Add Assertions? What Is the Protocol?.....	418
14.5.3	How Do I Know I Have Enough Assertions? .....	419
14.5.4	A Simple PCI Read Example: Creating an Assertion Test Plan.....	420
14.6	Assertion Types .....	422
14.7	Conventions Used in This Chapter. ....	423
14.8	Immediate Assertions.....	423
14.8.1	Deferred Immediate Assertions .....	426
14.9	Concurrent Assertions: Basics .....	429
14.9.1	Implication Operator .....	434
14.10	Clocking Basics .....	436
14.10.1	Default Clocking Block .....	438
14.10.2	Sampling Edge (Clock Edge).....	442
14.10.3	Active Region.....	443
14.10.4	Observed Region .....	443
14.10.5	Reactive Region .....	443
14.10.6	Preponed Region .....	443
14.11	Concurrent Assertions Are Multi-threaded .....	447
14.12	Formal Arguments .....	448
14.13	Disable (Property) Operator: disable iff. ....	449
14.14	Severity Levels.....	451
14.15	Binding Properties .....	452
14.15.1	Binding Properties (Scope Visibility) .....	453
14.15.2	VHDL DUT Binding with SystemVerilog Assertions....	455
14.16	Difference Between “sequence” and “property” .....	456
14.17	Sampled Value Functions.....	457
14.17.1	\$rose: Edge Detection in Property/Sequence.....	458
14.17.2	\$fell: Edge Detection in Property/Sequence .....	459
14.17.3	Edge Detection Is Useful Because.....	459
14.17.4	\$stable .....	460
14.17.5	\$past .....	461
14.18	Operators .....	466
14.18.1	##m: Clock Delay .....	466
14.18.2	##[m:n]: Clock Delay Range .....	468
14.18.3	[*m]: Consecutive Repetition Operator.....	474
14.18.4	[*m:n]: Consecutive Repetition Range Operator.....	476
14.18.5	[=m]: Non-consecutive Repetition.....	482
14.18.6	[=m:n]: Non-consecutive Repetition Range Operator .....	484
14.18.7	[->] Non-consecutive GoTo Repetition Operator .....	486
14.18.8	Difference Between [=m:n] and [->m:n].....	487
14.18.9	Application: GoTo Repetition – Non-consecutive Operator .....	489

14.18.10	Sig1 throughout Seq1.....	489
14.18.11	Seq1 within Seq2 .....	492
14.18.12	Application: Seq1 “within” Seq2.....	492
14.18.13	Seq1 and Seq2 .....	495
14.18.14	Application: “and” Operator .....	498
14.18.15	Seq1 or Seq2 .....	498
14.18.16	Seq1 “intersect” Seq2 .....	500
14.18.17	Application: “intersect” Operator .....	501
14.18.18	<i>first_match</i> .....	505
14.18.19	Application: <i>first_match</i> . ....	505
14.18.20	<i>not</i> Operator.....	508
14.18.21	<i>if</i> (Expression) <i>property_expr1 Else property_expr2</i> ...	509
14.18.22	“iff” and “implies”.....	509
14.19	System Functions and Tasks .....	510
14.19.1	\$onehot and \$onehot0 .....	510
14.19.2	\$isunknown .....	511
14.19.3	\$countones .....	511
14.19.4	\$countbits.....	512
14.19.5	\$assertoff, \$asserton, and \$assertkill .....	514
14.20	Multiply Clocked Sequences and Properties .....	514
14.20.1	Multiply Clocked Sequences .....	515
14.20.2	Multiply Clocked Properties: “and” Operator .....	516
14.20.3	Multiply Clocked Properties: “or” Operator .....	517
14.20.4	Multiply Clocked Properties – “not”: Operator .....	517
14.20.5	Multiply Clocked Properties: Clock Resolution .....	519
14.20.6	Multiply Clocked Properties: Legal and Illegal Conditions .....	519
14.21	Local Variables.....	520
14.21.1	Application: Local Variables .....	524
14.22	End Point of a Sequence (.triggered).....	525
14.22.1	End Point of a Sequence (.matched) .....	528
14.23	“expect” .....	530
14.24	IEEE 1800-2012/2017 Features.....	532
14.24.1	\$changed .....	532
14.24.2	Future Global Clocking Sampled Value Functions .....	532
14.24.3	past Global Clocking Sampled Value Functions .....	533
14.24.4	“followed by” Properties #:# and #:# .....	534
14.24.5	“always” and “s_always” Property .....	536
14.24.6	“eventually” and “s_eventually” .....	538
14.24.7	“until,” “s_until,” “until_with,” and “s_until_with”.....	539
14.24.8	“nexttime” and “s_nexttime” .....	541
14.25	Abort Properties: reject_on, accept_on, sync_reject_on, and sync_accept_on .....	545
14.26	\$assertpassoff, \$assertpasson, \$assertfailoff, \$assertfailon, \$assertnonvacuouson, and \$assertvacuousoff .....	548

14.27 Embedding Concurrent Assertions in Procedural Block . . . . .	549
14.28 Nested Implications . . . . .	552
<b>15 SystemVerilog Functional Coverage . . . . .</b>	<b>553</b>
15.1 Difference Between Code Coverage and Functional Coverage . . . . .	554
15.1.1 Code Coverage . . . . .	554
15.1.2 Functional Coverage . . . . .	556
15.2 Functional Coverage Methodology . . . . .	557
15.3 Covergroup: Basics . . . . .	558
15.4 Coverpoint: Basics . . . . .	560
15.4.1 Covergroup/Coverpoint Example . . . . .	562
15.4.2 Coverpoint Using a Function or an Expression . . . . .	563
15.5 “bins”: Basics . . . . .	564
15.5.1 Covergroup/Coverpoint Example with “bins” . . . . .	566
15.5.2 “bins” Filtering . . . . .	567
15.6 Covergroup: Formal and Actual Arguments . . . . .	568
15.7 SystemVerilog Class-Based Coverage . . . . .	569
15.7.1 Class: Embedded Covergroup in a Class . . . . .	569
15.7.2 Multiple Covergroups in a Class . . . . .	570
15.7.3 Overriding Covergroups in a Class . . . . .	570
15.7.4 Parameterizing Coverpoints in a Class . . . . .	574
15.8 “cross” Coverage . . . . .	575
15.9 “bins” for Transition Coverage . . . . .	581
15.10 “wildcard bins” . . . . .	583
15.11 “ignore_bins” . . . . .	583
15.12 “illegal_bins” . . . . .	588
15.13 “binsof” and “intersect” . . . . .	588
15.14 User-Defined “sample()” Method . . . . .	591
15.15 Querying for Coverage . . . . .	596
15.16 Strobe () Method . . . . .	597
15.17 Coverage Options: Instance-Specific Example (Fig. 15.18) . . . . .	597
15.18 Coverage Options for “covergroup” Type: Example (Fig. 15.19) . . . . .	598
15.19 Coverage Options: Instance-Specific Per-syntactic Level . . . . .	598
15.20 Coverage System Tasks, Functions, and Methods . . . . .	598
<b>16 SystemVerilog Processes . . . . .</b>	<b>601</b>
16.1 “initial” and “always” Procedural Blocks . . . . .	601
16.1.1 “initial” Procedural Block . . . . .	602
16.1.2 “always” Procedural Block: General . . . . .	603
16.1.3 “always_comb” . . . . .	606
16.1.4 always @ (*) . . . . .	609
16.1.5 “always_latch” . . . . .	610
16.1.6 “always_ff” . . . . .	612
16.2 “final” procedure . . . . .	614

16.3	Parallel Blocks: fork-join . . . . .	615
16.3.1	fork-join . . . . .	615
16.3.2	fork-join_any . . . . .	617
16.3.3	fork-join_none . . . . .	619
16.4	Level-Sensitive Time Control . . . . .	620
16.4.1	Wait Fork . . . . .	622
16.5	Named Event Time Control . . . . .	624
16.5.1	Merging of Named Events . . . . .	625
16.5.2	Event Comparison . . . . .	627
16.6	Conditional Event Control . . . . .	627
16.7	Disable Statement . . . . .	628
17	<b>Procedural Programming Statements</b> . . . . .	631
17.1	if-else-if Statements . . . . .	631
17.1.1	unique-if and unique0-if . . . . .	633
17.1.2	priority-if . . . . .	636
17.2	Case Statements . . . . .	637
17.2.1	casex, casez, and do not care . . . . .	640
17.2.2	Constant Expression in “case” Statement . . . . .	644
17.2.3	One-Hot State Machine Using Constant Case Expression . . . . .	646
17.2.4	unique-case, unique0-case, and priority-case . . . . .	648
17.3	Loop Statements . . . . .	652
17.3.1	The “for” Loop . . . . .	652
17.3.2	The “repeat” Loop . . . . .	654
17.3.3	The “foreach” Loop . . . . .	655
17.3.4	The “while” Loop . . . . .	658
17.3.5	The “do – while” Loop . . . . .	659
17.3.6	The “forever” Loop . . . . .	660
17.4	Jump Statements . . . . .	662
17.4.1	“break” and “continue” . . . . .	662
18	<b>Inter-process Synchronization, Semaphores and Mailboxes</b> . . . . .	665
18.1	Semaphores . . . . .	665
18.2	Mailboxes . . . . .	668
18.2.1	Parameterized Mailbox . . . . .	675
19	<b>Clocking Blocks</b> . . . . .	677
19.1	Clocking Blocks with Interfaces . . . . .	683
19.2	Global Clocking . . . . .	685
20	<b>Checkers</b> . . . . .	689
20.1	Nested Checkers . . . . .	694
20.2	Checkers: Legal Conditions . . . . .	695
20.3	Checkers: Illegal Conditions . . . . .	696
20.4	Checkers: Important Points . . . . .	698
20.5	Checkers: Instantiation Rules . . . . .	701

20.6	Checkers: Rules for “formal” and “actual” Arguments . . . . .	703
20.7	Checkers: In a Package . . . . .	704
<b>21</b>	<b>“let” Declarations . . . . .</b>	<b>705</b>
21.1	“let”: Local Scope . . . . .	706
21.2	“let”: With Parameters . . . . .	707
21.3	“let”: In Immediate and Concurrent Assertions . . . . .	709
<b>22</b>	<b>Tasks and Functions . . . . .</b>	<b>715</b>
22.1	Tasks . . . . .	716
22.1.1	Static and Automatic Tasks . . . . .	718
22.2	Functions . . . . .	724
22.2.1	Function Called as a Statement . . . . .	725
22.2.2	Function Name or “return” to Return a Value . . . . .	726
22.2.3	Void Functions . . . . .	727
22.2.4	Static and Automatic Functions . . . . .	728
22.3	Passing Arguments by Value or Reference to Tasks and Functions . . . . .	730
22.3.1	Pass by Value . . . . .	730
22.3.2	Pass by Reference . . . . .	732
22.4	Default Argument Values . . . . .	735
22.5	Argument Binding by Name . . . . .	737
22.6	Parameterized Tasks and Functions . . . . .	738
<b>23</b>	<b>Procedural and Continuous Assignments . . . . .</b>	<b>745</b>
23.1	Procedural Assignments . . . . .	746
23.1.1	Blocking Versus Non-Blocking Procedural Assignments . . . . .	746
23.1.2	Continuous Assignment . . . . .	751
23.2	Procedural Continuous Assignment: Assign and Deassign . . . . .	753
23.3	Procedural Continuous Assignment: Force-Release . . . . .	755
<b>24</b>	<b>Utility System Tasks and Functions . . . . .</b>	<b>759</b>
24.1	Simulation Control System Tasks . . . . .	759
24.2	Simulation Time System Functions . . . . .	760
24.3	Timescale System Tasks . . . . .	761
24.3.1	\$printtimescale . . . . .	761
24.3.2	\$timeformat . . . . .	761
24.4	Conversion Functions . . . . .	763
24.4.1	Conversion to/from Signed/Unsigned Expression . . . . .	765
24.5	\$bits: Expression Size System Function . . . . .	766
24.6	Array Querying System Functions . . . . .	766
24.7	Math Functions . . . . .	769
24.7.1	Integer Math Functions . . . . .	769
24.7.2	Real Math Functions . . . . .	769
24.8	Bit-Vector System Functions . . . . .	771
24.9	Severity System Tasks . . . . .	773

24.10 \$random and Probabilistic Distribution Functions .....	773
24.10.1 \$random .....	773
24.10.2 Probabilistic Distribution Functions .....	774
24.11 Queue Management Stochastic Analysis Tasks.....	775
24.11.1 \$q_initialize .....	776
24.11.2 \$q_add .....	776
24.11.3 \$q_remove .....	776
24.11.4 \$q_full .....	777
24.11.5 \$q_exam.....	777
24.11.6 Example of Queue Management Stochastic Analysis Tasks and Functions .....	778
<b>25 I/O System Tasks and Functions .....</b>	<b>785</b>
25.1 Display Tasks .....	785
25.2 Escape Identifiers.....	788
25.3 Format Specifications.....	789
25.4 File I/O System Tasks and Functions.....	792
25.4.1 \$fopen and \$fclose .....	792
25.4.2 \$fdisplay, \$fwrite, \$fmonitor, and \$fstroke .....	797
25.4.3 \$fwrite and \$format .....	797
25.5 Reading Data from a File.....	800
25.5.1 \$fgetc, \$ungetc, and \$fgets .....	800
25.5.2 \$fscanf and \$sscanf .....	802
25.5.3 \$ftell, \$fseek, and \$rewind.....	808
25.5.4 \$fread .....	808
25.5.5 \$readmemb and \$readmemh .....	809
25.5.6 \$writememb and \$writememh.....	813
25.6 \$test\$plusargs and \$value\$plusargs .....	814
25.6.1 \$test\$plusargs.....	815
25.6.2 \$value\$plusargs .....	816
25.7 Value Change Dump (VCD) File.....	817
25.7.1 \$dumpfile .....	817
25.7.2 \$dumpvars .....	818
25.7.3 \$dumpon/\$dumpoff .....	819
25.7.4 \$dumplimit.....	819
25.7.5 \$dumpflush.....	819
25.7.6 \$dumpall .....	819
25.7.7 \$dumpports .....	820
<b>26 GENERATE Constructs.....</b>	<b>821</b>
26.1 Generate: Loop Constructs .....	822
26.2 Generate : Conditional Construct.....	825
<b>27 Compiler Directives .....</b>	<b>831</b>
27.1 `define .....	831
27.1.1 `undef and `undefineall .....	836
27.2 `ifdef, `else, `elsif, `endif, and `ifndef .....	837

27.3 `timescale . . . . .	839
27.4 `default_nettype . . . . .	842
27.5 `resetall. . . . .	842
<b>Bibliography</b> . . . . .	843
<b>Index</b> . . . . .	845

# List of Figures

Fig. 1.1	SystemVerilog language.....	2
Fig. 1.2	SystemVerilog Assertions (SVA) evolution .....	4
Fig. 2.1	Tristate logic.....	14
Fig. 3.1	4-D unpacked array .....	67
Fig. 3.2	1-D packed and 3-D unpacked array.....	69
Fig. 3.3	2-D packed and 2-D unpacked array.....	70
Fig. 3.4	3-D packed and 1-D unpacked array.....	71
Fig. 4.1	“push” and “pop” of a queue.....	110
Fig. 5.1	Packed structure .....	124
Fig. 8.1	Inheritance memory allocation.....	168
Fig. 8.2	Class assignment .....	186
Fig. 8.3	Shallow copy .....	187
Fig. 8.4	Upcasting and downcasting.....	194
Fig. 12.1	Byte Array elements packed into an “int” .....	314
Fig. 12.2	Unpacking of bits .....	314
Fig. 13.1	Advantages of constrained random verification.....	326
Fig. 13.2	Constrained random verification (CRV) methodology .....	327
Fig. 14.1	SystemVerilog Assertions evolution.....	410
Fig. 14.2	A simple bus protocol design and its SVA property .....	411
Fig. 14.3	Verilog code for the simple bus protocol .....	412
Fig. 14.4	Assertions improve observability .....	413
Fig. 14.5	Assertions and OVL for different uses.....	415
Fig. 14.6	Assertions in formal and simulation .....	417
Fig. 14.7	A simple PCI read protocol.....	420
Fig. 14.8	Immediate assertion: basics.....	424
Fig. 14.9	Immediate assertions: finer points.....	426
Fig. 14.10	Concurrent assertion: basics.....	430

Fig. 14.11	Concurrent assertion: – sampling edge and action blocks .....	431
Fig. 14.12	Concurrent assertion: implication, antecedent, and consequent .....	432
Fig. 14.13	Property with an embedded sequence .....	433
Fig. 14.14	Implication operator: overlapping and non-overlapping.....	434
Fig. 14.15	Equivalence between overlapping and non-overlapping implication operators .....	436
Fig. 14.16	Clocking basics .....	437
Fig. 14.17	Clocking basics: clock in “assert,” “property,” and “sequence” .....	438
Fig. 14.18	Default clocking block.....	439
Fig. 14.19	“Clocking” and “default clocking” .....	440
Fig. 14.20	Assertion variable sampling and evaluation/execution in a simulation time tick.....	442
Fig. 14.21	Multi-threaded concurrent assertions.....	447
Fig. 14.22	Formal and actual arguments .....	448
Fig. 14.23	Event control as formal argument.....	449
Fig. 14.24	“disable iff” operator.....	450
Fig. 14.25	Severity levels for concurrent and immediate assertions.....	451
Fig. 14.26	Binding properties.....	452
Fig. 14.27	: Binding properties to design “module” internal signals (scope visibility).....	454
Fig. 14.28	Binding VHDL DUT to SystemVerilog Assertions module .....	456
Fig. 14.29	Sampled value functions \$rose and \$fell: basics .....	457
Fig. 14.30	\$rose: basics .....	458
Fig. 14.31	\$fell: basics .....	459
Fig. 14.32	Edge sensitive and performance Iiplication .....	460
Fig. 14.33	\$stable: basics .....	461
Fig. 14.34	\$past: basics .....	462
Fig. 14.35	\$past: gating expression.....	463
Fig. 14.36	\$past application .....	464
Fig. 14.37	##m clock delay: basics .....	467
Fig. 14.38	##[m:n] clock delay range .....	467
Fig. 14.39	##[m:n]: multiple threads.....	468
Fig. 14.40	[*m]: consecutive repetition operator – basics .....	475
Fig. 14.41	Consecutive repetition operator: application .....	475
Fig. 14.42	[*m:n] consecutive repetition range: basics.....	477
Fig. 14.43	[*m:n] consecutive repetition range: example .....	478
Fig. 14.44	[*m:n] consecutive repetition range: application.....	479
Fig. 14.45	[*m:n] consecutive repetition range: application.....	480
Fig. 14.46	[*m:n] consecutive repetition range: application.....	480
Fig. 14.47	Consecutive range application .....	481
Fig. 14.48	Simulation log for consecutive range application.....	482
Fig. 14.49	Non-consecutive repetition operator.....	483
Fig. 14.50	Non-consecutive repetition operator: example .....	484
Fig. 14.51	Non-consecutive range operator .....	485
Fig. 14.52	Non-consecutive repetition range operator: application .....	486

Fig. 14.53	GoTo non-consecutive repetition operator .....	487
Fig. 14.54	Non-consecutive repetition: example.....	488
Fig. 14.55	Difference between [=m:n] and [-> m:n] .....	488
Fig. 14.56	GoTo repetition non-consecutive operator: application .....	489
Fig. 14.57	Sig1 throughout seq1 .....	490
Fig. 14.58	Sig1 “throughout” seq1: application.....	490
Fig. 14.59	Sig1 “throughout” seq1: application simulation log .....	491
Fig. 14.60	Seq1 “within” seq2 .....	493
Fig. 14.61	Seq1 “within” seq2: application.....	494
Fig. 14.62	“within” operator: simulation log – pass cases .....	495
Fig. 14.63	“within” operator: simulation log – fail cases.....	496
Fig. 14.64	Seq1 “and” seq2: basics .....	497
Fig. 14.65	“and” operator: application .....	497
Fig. 14.66	“and” operator: application II .....	498
Fig. 14.67	“and” of expressions .....	499
Fig. 14.68	Seq1 “or” seq2: basics .....	499
Fig. 14.69	“or” operator: application .....	500
Fig. 14.70	“or” operator: application II.....	501
Fig. 14.71	Seq1 “intersect” seq2 .....	502
Fig. 14.72	Seq1 “intersect” seq2: application .....	503
Fig. 14.73	“intersect”: interesting application.....	503
Fig. 14.74	“first_match”: application .....	504
Fig. 14.75	“first_match” operator: application.....	506
Fig. 14.76	“not” operator: basics.....	507
Fig. 14.77	“not” operator: application.....	507
Fig. 14.78	If...else.....	508
Fig. 14.79	\$onehot and \$onehot0 .....	510
Fig. 14.80	\$isunknown .....	511
Fig. 14.81	\$countones .....	512
Fig. 14.82	\$asserton, \$assertoff, and \$assertkill.....	513
Fig. 14.83	Application: project-wide assertion control.....	514
Fig. 14.84	Multiply clocked sequences .....	515
Fig. 14.85	Multiply clocked properties: “and” operator between two different clocks.....	516
Fig. 14.86	Multiply clocked properties: “or” operator.....	517
Fig. 14.87	Multiply clocked properties: “not” operator .....	518
Fig. 14.88	Multiply clocked properties: clock resolution .....	518
Fig. 14.89	Multiply clocked properties: legal and illegal conditions .....	519
Fig. 14.90	Local variables: basics .....	520
Fig. 14.91	Local variables: do’s and do not’s.....	521
Fig. 14.92	Local variables: further nuances .....	522
Fig. 14.93	Local variables: application .....	525
Fig. 14.94	.triggered: end point of a sequence .....	526
Fig. 14.95	.matched: basics .....	529
Fig. 14.96	.matched: application .....	529

Fig. 14.97	“expect”: basics.....	530
Fig. 14.98	\$changed .....	531
Fig. 14.99	\$changed - pass and fail cases .....	531
Fig. 14.100	Embedding concurrent assertion in procedural block.....	550
Fig. 14.101	Concurrent assertion embedded in procedural block is non-blocking.....	550
Fig. 14.102	Nested implications in a property .....	551
Fig. 15.1	Comprehensive assertions- and functional coverage-based methodology .....	559
Fig. 15.2	Covergroup and coverpoint basics .....	560
Fig. 15.3	Coverpoint basics.....	561
Fig. 15.4	Covergroup/coverpoint example .....	562
Fig. 15.5	“bins”: basics .....	564
Fig. 15.6	Covergroup/coverpoint example with “bins”.....	566
Fig. 15.7	Covergroup: formal and actual arguments .....	567
Fig. 15.8	“cross” coverage: basics.....	573
Fig. 15.9	“cross”: example .....	573
Fig. 15.10	“cross” example: simulation log .....	574
Fig. 15.11	“bins” for transition coverage .....	577
Fig. 15.12	“bins” for transition: further nuances.....	578
Fig. 15.13	Example of PCI cycles transition coverage .....	578
Fig. 15.14	“wildcard bins” .....	580
Fig. 15.15	“ignore_bins” .....	581
Fig. 15.16	“illegal_bins” .....	584
Fig. 15.17	“binsof” and “intersect” .....	585
Fig. 15.18	Coverage options: instance-specific example .....	589
Fig. 15.19	Coverage options for “covergroup” type .....	590
Fig. 15.20	Predefined tasks, functions and methods for functional coverage ...	594
Fig 17.1	Finite-state machine .....	646
Fig. 19.1	Clocking block: sample and drive signals.....	678
Fig. 23.1	Non-blocking assignment shown as sequential flops.....	746

# List of Tables

Table 2.1	Integer data types .....	6
Table 2.2	“real” data types .....	10
Table 2.3	“real” data-type conversion functions .....	12
Table 2.4	Nets .....	13
Table 2.5	wire/tri truth table.....	15
Table 2.6	wand/triand truth table .....	17
Table 2.7	“wor”/”trior” truth table .....	17
Table 2.8	tri0 truth table.....	18
Table 2.9	tri1 truth table.....	18
Table 2.10	Variables.....	25
Table 2.11	Enumerated-type methods.....	33
Table 2.12	Enumeration element ranges.....	36
Table 2.13	String operators .....	42
Table 2.14	String methods .....	44
Table 3.1	Associative array methods .....	88
Table 3.2	Locator methods using the with clause .....	93
Table 3.3	Locator methods where with clause is optional .....	96
Table 3.4	Array ordering methods .....	100
Table 3.5	Array reduction methods.....	101
Table 4.1	Queue methods.....	110
Table 12.1	SystemVerilog operators .....	291
Table 12.2	Arithmetic operators.....	292
Table 12.3	Relational operators .....	297
Table 12.4	Equality operators .....	298
Table 12.5	Logical equality truth: 1-bit operands .....	299
Table 12.6	Case equality truth: 1-bit operands .....	299
Table 12.7	Logical equality: vector operands .....	300
Table 12.8	Case equality: vector operands.....	300
Table 12.9	Wildcard equality operators .....	301

Table 12.10	Logical operators.....	301
Table 12.11	Bitwise binary AND (&) operator.....	303
Table 12.12	Bitwise binary OR (l) operator .....	303
Table 12.13	Bitwise binary exclusive OR (^) operator .....	303
Table 12.14	Bitwise binary exclusive NOR operator.....	304
Table 12.15	Bitwise unary negation (~) operator.....	304
Table 12.16	Reduction unary AND operator .....	306
Table 12.17	Reduction unary OR operator .....	306
Table 12.18	Reduction unary exclusive OR operator .....	306
Table 14.1	PCI read protocol test plan by functional verification team.....	421
Table 14.2	PCI read protocol test plan by design team.....	421
Table 14.3	Conventions used in this chapter.....	423
Table 14.4	Concurrent assertion operators.....	465
Table 14.5	Concurrent assertion operators – contd.....	466
Table 15.1	Coverage options: instance-specific per-syntactic level.....	591
Table 15.2	Coverage group-type (static) options .....	592
Table 15.3	Instance-specific coverage options.....	593
Table 17.1	caseZ truth table .....	643
Table 17.2	caseX truth table.....	643
Table 24.1	\$timeformat units_number arguments .....	762
Table 24.2	\$timeformat default value of arguments .....	762
Table 24.3	Array querying system functions .....	767
Table 24.4	Real math functions.....	769
Table 24.5	Status code for queue system tasks .....	777
Table 24.6	q_stat_code and q_stat_value of task \$q_exam .....	778
Table 25.1	Display tasks .....	786
Table 25.2	Escape identifiers .....	789
Table 25.3	Format specifiers for display statements .....	790
Table 25.4	File types used with \$fopen.....	793

## About the Author

**Ashok B. Mehta** has worked in the ASIC/SoC design and verification field for over 30 years. He started his career at Digital Equipment Corporation (DEC) as a CPU design engineer. He then worked at Data General, Intel (first Pentium design team), and after a route through a couple of startups, worked at Applied Micro and TSMC.

He was a very early adopter of Verilog language and participated in Verilog, VHDL, iHDL (Intel HDL) and SDF (standard delay format) standards subcommittees. He has also been a proponent of ESL (Electronic System Level) designs, and at TSMC he architected two industry standard ESL Reference Flows that created a verification reuse platform from ESL to Gate level. Lately, he has been involved with 2.5D/3DIC stacked SoC design and verification.

Ashok earned an MSEE from the University of Missouri. He holds 19 US patents in the fields of SoC, 3DIC and ESL design and verification. In his spare time, he is an amateur photographer and likes to play drums on 70's rock music, driving his neighbors up the wall ☺

# Chapter 1

## Introduction



**Introduction** This chapter introduces the evolution of the IEEE standard SystemVerilog language. It describes how the SystemVerilog for Design and Verification, SystemVerilog Assertions, and SystemVerilog Functional Coverage language subsets fold into a single unified language.

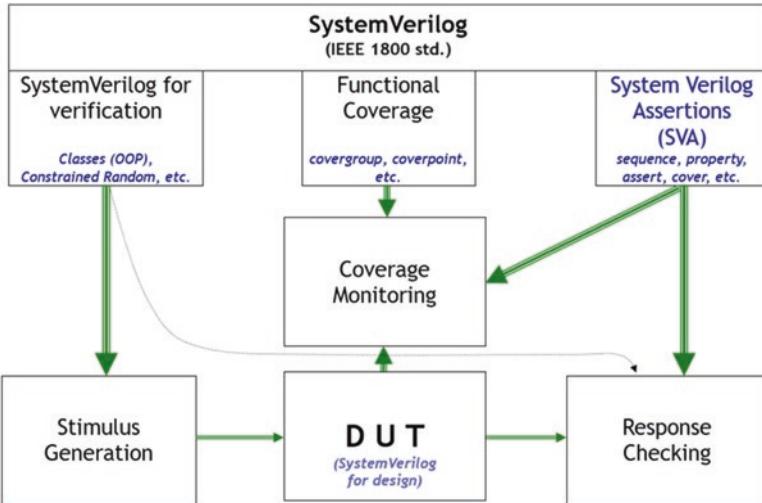
SystemVerilog is the IEEE standard 1800 unified language for functional design and verification. It has its roots in the Verilog language that was invented way back in the mid-1980s by Phil Moorby of Gateway Design Automation. This book covers the IEEE 1800–2017 version of the standard. The standard was developed to meet the increasing usage of the language in specification, design, and verification of hardware.

SystemVerilog language has four distinct languages under a single simulation kernel. They are:

1. SystemVerilog for Design (synthesizable subset)
2. SystemVerilog for Verification (which includes the OOP subset)
3. SystemVerilog Assertions
4. SystemVerilog Functional Coverage

The SystemVerilog for Design and Verification subsets, SystemVerilog Assertions (SVA) subsets, and SystemVerilog Functional Coverage subsets are orthogonal to each other. Different features have separate syntax, but they all share a common set of data types, expression operators, etc. And all these subsets work under a unified simulation kernel (simulation time tick).

Figure 1.1 shows the language subset for testbench constructs (class, constrained random, etc.), the subset for SystemVerilog Assertions (SVA), and the subset for Functional Coverage. Each has a unique role to play in the language. It is the combination of all these subsets that makes the language ever so powerful. No need to have a multi-language simulation environment (one language for design, one for



**Fig. 1.1** SystemVerilog language

verification, etc.). The unified language allows you to tackle the entire domain of design and verification of hardware:

- Design the DUT (SystemVerilog subset for design).
- Provide stimulus/check response from the testbench (SystemVerilog OOP subset for Verification).
- Check combinatorial/sequential logic response from the DUT (SystemVerilog Assertions).
- Measure Functional Coverage of the design (SystemVerilog Functional Coverage).

This integrated whole created by SystemVerilog greatly exceeds the sum of its individual components, creating a new type of engineering language, HDVL (hardware design and verification language). Using a single unified language enables a unified environment for the engineers to model large, complex designs and verify that these designs are functionally correct.

## 1.1 SystemVerilog Language Evolution

It all started with the Verilog language as the first generation of hardware design and verification language around 1985. It became an IEEE standard in 1995. The original Verilog was developed by Phil Moorby of Gateway Design Automation. It was geared more toward hardware design but had sufficient behavioral constructs to develop testbenches. Note that the designs in late 1980s were in 10 to 100,000 gates; range and verification entailed quite a bit of gate-level verification and some RTL

level. So, the language, at that time, had sufficient features to tackle both design and verification.

Verilog-2001 was a major update to the standard and added features such as multi-dimensional arrays, auto variables, etc. Higher-level constructs for design were added. By this time, the designs were already in multimillion gates, and it became evident that the language needed a major overhaul when it came to functional verification capabilities. Since the Verilog language lacked advanced constructs for verification, other languages popped up, namely, Vera and “e,” to augment Verilog with OOP (object-oriented programming) subsets. But now the user had to create a multi-language environment with support from different EDA vendors, which is a very cumbersome, error-prone and time-consuming task.

Enter SystemVerilog 3.0 (June of 2002). It added advanced Verilog and “C” data types. It was a step forward to making a robust language for both design and verification. It added extensions to synthesizable constructs of Verilog and enabled modeling hardware at higher levels of abstraction. But still it lacked language constructs that would allow for a reusable/modular code/environment for verification.

Enter SystemVerilog 3.1 (May of 2003). This version completely overhauled the language subset for verification. It added C++-style “class” construct with methods, properties, inheritance, etc. It also added features to allow constrained random verification. It also added the SystemVerilog Assertions (SVA) subset with enhanced semantics for sequential temporal domain expressions, sequence and property generation, etc. It also added the Functional Coverage subset to the language. It allowed one to objectively measure the functional coverage of a design using coverpoints, covergroups, bins, etc.

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

So far, SystemVerilog was not an IEEE standard. It was an Accellera standard. In June of 2004, after SystemVerilog 3.1a was released, Accellera donated SystemVerilog standard to IEEE standards association, which oversaw the Verilog 1364 standard. Accellera worked with the IEEE to form a new standards request, to review and standardize the SystemVerilog extensions to Verilog. IEEE assigned the project number 1800 to SystemVerilog, and hence SystemVerilog is IEEE 1800 standard.

SystemVerilog IEEE 1800 contains many extensions for the verification of large designs, integrating features from SUPERLOG, Vera, C, C++, and VHDL languages. The primary technology donations that make up SystemVerilog include:

- The SUPERLOG Extended Synthesizable Subset (SUPERLOG ESS), from Co-Design Automation:
  - SUPERLOG (Co-Design Automation) was the brainchild of Peter Flake, Phil Moorby, and Simon Davidmann. In 2001, Co-Design Automation (which was

acquired by Synopsys in 2002) donated to Accellera the SUPERLOG Extended Synthesizable Subset. This subset included a standard set of enhancements, including enhancements for verification. You can say, SystemVerilog started with the donation of the SUPERLOG language to [Accellera](#) in 2002 by the startup company Co-Design Automation.

- PSL Assertions (which began as a donation of Sugar language from IBM)
- OpenVera Assertions (OVA) from Synopsys:
  - OpenVera Assertions (OVA) and DirectC features were donated to Accellera. These donations significantly extended the verification capabilities of Verilog. The bulk of the verification functionality is based on the [OpenVera](#) language donated by [Synopsys](#).
- The DirectC and coverage API from Synopsys
- Tagged unions and high-level language features from Bluespec

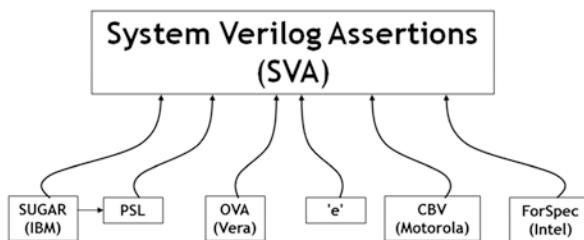
Then came the next revision of the standard SystemVerilog – 2012. And then SystemVerilog – 2017. This book is based on the 2017 LRM.

SystemVerilog Assertions (SVA) subset/sub-language is also influenced by many different languages as shown in Fig. 1.2.

SystemVerilog Assertions language is derived from many different languages. Features from these languages either influenced the language or were directly used as part of the language syntax/semantic.

Sugar from IBM led to PSL. Both contributed to SVA. The other languages that contributed are Vera, “e,” CBV from Motorola, and ForSpec from Intel.

In short, when we use SystemVerilog Assertions language, we have the benefit of using the latest evolution of an assertion language that benefited from many other robust assertion languages.



**Fig. 1.2** SystemVerilog Assertions (SVA) evolution

# Chapter 2

## Data Types



**Introduction** This chapter describes the rich set of data types that SystemVerilog offers. Integer datatypes and real datatypes are discussed. In addition, use-defined types; static, local, automatic, and global variables; enumerated types; string data types; and event data types are discussed. Each data type is explained with examples and simulation logs.

SystemVerilog offers a rich variety of integer and real data types and nets. Verilog used to provide “reg” and “wire,” but these were not sufficient especially for functional verification. Verilog 2001 introduced the concept of “variables” to emphasize that “reg” is a data type of a variable. SV takes it a step further in saying that a net is a signal with a logic data, and you can apply other data types to nets. SystemVerilog also offers object-oriented feature set that allows for modular and reusable verification environment. Many of the data types are geared toward supporting these new features.

The categories under which data types can be broadly summarized are the “integer” data type and the “real” data type and nets.

Note: We will use the term *integral* data type throughout the book. It refers to the data types that can represent a single basic integer data type, packed array, packed structure, packed union, enum variable, or time variable.

### 2.1 Integer Data Types

The integer data types are shown in Table 2.1. The integer data types can be broadly classified into two types, the 2-state vs. 4-state types and signed vs. unsigned types. 4-states are “0,” “1,” “x,” and “z” and 2-states are “0” and “1.”

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_2](https://doi.org/10.1007/978-3-030-71319-5_2)) contains supplementary material, which is available to authorized users.

**Table 2.1** Integer data types

Integer data type	Description
shortint	2-state data type, 16-bit signed integer
int	2-state data type, 32-bit signed integer
longint	2-state data type, 64-bit signed integer
byte	2-state data type, 8-bit signed integer or ASCII character
bit	2-state data type, user-defined vector size, unsigned
logic	4-state data type, user-defined vector size, unsigned
reg	4-state data type, user-defined vector size, unsigned
integer	4-state data type, 32-bit signed integer
time	4-state data type, 64-bit unsigned integer

### 2.1.1 Integer, int, longint, shortint, logic, byte, reg

The int, longint, and shortint are 2-state signed integer data types, while the “integer” is a 4-state signed integer data type. Legacy “reg” and the new “logic” types are identical. They are both 4-state unsigned.

Let us look at an example to see how these work:

```

module datatypel;
    integer a;          //4 state - 32 bit signed
    int b;             //2 state - 32 bit signed
    shortint c;        //2 state - 16 bit signed
    longint d;         //2 state - 64 bit signed
    logic [7:0] A1;   //4-state - unsigned 'logic'
    logic signed [7:0] s11; //4-state - signed 'logic'
    byte b11;          //2-state signed 'byte'
    reg [7:0] r1;     //4-state - unsigned 'reg'

initial
begin
    a = 'h xxzz_ffff; //integer - 4 state - 32 bit signed
    b = -1;           //int - 2 state - 32 bit signed
    c = 'h ffxfx;    //shortint - 2 state - 16 bit signed
    d = 'h ffff_xxxx_ffff_zzzz;
                    //longint - 2 state - 64 bit signed

    A1 = -1 ; //signed assignment to unsigned 'logic'
    s11 = -1; //signed assignment to signed 'logic'
    b11 = -1; //signed byte
    r1 = 8'b xxzx_0101; //'reg' - unsigned 4-state
end

```

```

initial
begin #10;
    $display("a = %h b = %h c = %h d = %h", a, b, c, d);
$display("A1 = %0d s11=%0d b11 = %0d r1 = %b",A1,s11,b11,r1);
    #10 $finish(2);
end
endmodule

```

In this example, we define a, b, c, and d to be of type integer, int, shortint, and longint, respectively. Then we define “A1” as an 8-bit unsigned logic type. Next, we define “s11” as of type “logic” – but – signed. Yes, you can take an unsigned type and make it a signed type by explicitly declaring it as a signed type. Finally, we define a signed byte “b11” and an unsigned 8-bit reg “r1.”

In the testbench, we assign different numbers to each of these variables. Some of these assigned values have “x” (unknown) in them to show how a 2-state vs. a 4-state variable treats an “x.” We also assign both positive and negative values to some of the variables to see how signed vs. unsigned values work:

*Simulation log:*

a = xxzzffff b = ffffffff c = f0f0 d = ffff0000ffff0000

A1 = 255 s11= -1 b11 = -1 r1 = xzxz0101

V C S   S i m u l a t i o n   R e p o r t

First, we assign a = ‘h xxzz\_ffff, where “a” is of type integer which is unsigned 4-state. So, the simulation log shows that “a” retains “x” as an “x” and “z” as “z” (4-states are 0, 1, X, and Z).

Next, we assign b = -1, where “b” is “int” signed 2-state type. We have displayed “b” as a hex value. Hence, “b” shows an assignment of “ffffffff” which is the equivalent of decimal -1. Point is that since “b” is signed, it retains its signed value assignment as signed.

Next, we assign c = ‘h ffx, where “c” is a shortint 2-state signed type. Hence, the display shows that assigned value to “c” is “f0f0.” This is because “x” is converted to “0” since there is no “x” state in a 2-state variable.

Next, we assign d = ‘h ffff\_xxxx\_ffff\_zzzz, where “d” is a longint 2-state signed type. Since “d” is a 2-state variable, it converts “x” to “0” as well as “z” to “0.” Hence the displayed value is “ffff0000ffff0000.”

Next, we assign A1 = -1, where “A1” is unsigned logic type. Note that “A1” is unsigned, but we are assigning a negative value to it. So, it will convert the negative 1 to positive 255 which is the equivalent of -1 in unsigned logic. Hence, the display shows A1 = 255.

Next, we assign s11 = -1, where “s11” is of type logic but is explicitly declared as “signed.” So, even though “logic” is of type unsigned, “s11” is converted to of type signed. Hence, the assigned value of “-1” remains as “-1” as shown in the display (s11 = -1).

Next, we assign b11 = -1, where b11 is a byte of type 2-state signed. Hence, it also retains the assigned negative value as negative, as shown in the simulation log.

Finally, we assign r1 = 8'b xzxz\_0101, where r1 is “reg” which is unsigned 4-state type. Since it is a 4-state type, it will retain “x” as an “x” and a “z” as a “z”

as shown in the simulation log. Note that unsigned 4-state “logic” and unsigned 4-state “reg” are equivalent. There is no difference between them. “reg” is kept around for legacy reasons.

Also, you can explicitly assign a signed number to a variable. So, for example, 1'sb1 is signed number assignment, while 1'b1 is unsigned assignment. Here is a simple example:

```
logic [7:0] L1; //unsigned logic type
L1 = 4'sb1001; //= 8'b11111001 //Sign extension
L1 = 1'sb1; //= 8'b1111_1111 //Sign extension
L1 = 8'sb1; //= 8'b0000_0001 //NO sign extension because of
//explicit width being same as vector declaration
L1 = 8'sbX; //=8'bxxxx_xxxx
```

### 2.1.2 Signed Types

Let us see how signed and unsigned variables work. You can explicitly describe a “logic” type as signed, as we saw in previous section. Let us further explore that in seeing how assignments to signed vs. unsigned variables are evaluated. Here is an example:

```
module top;
    logic [7:0] r1;
    logic signed [7:0] srl1;

    initial begin
        r1 = -2;
        $display($stime,,,"r1=%d",r1);

        srl1 = -2;
        $display($stime,,,"srl1=%d",srl1);

        r1 = r1+1;
        $display($stime,,,"r1=%d",r1);

        srl1 = srl1+1;
        $display($stime,,,"srl1=%d",srl1);
    end
endmodule
```

*Simulation log:*

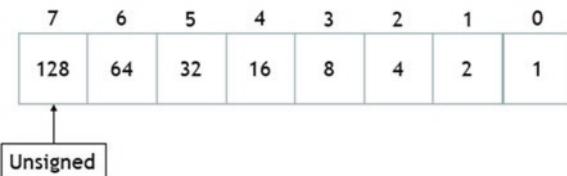
```
# run -all
# 0 r1=254
```

```
# 0 sr1= -2
# 0 r1=255
# 0 sr1= -1
# exit
```

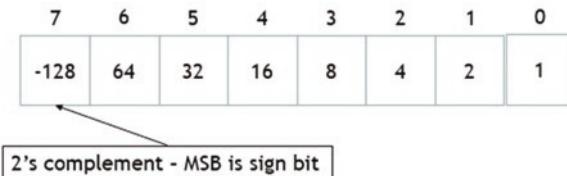
“r1” is declared as default unsigned 8-bit vector, while “sr1” is declared as signed 8-bit vector. When we assign  $r1 = -2$ , since “r1” is unsigned, it will have the value 254 (decimal equivalent of  $-2$ ). But “sr1” will evaluate as  $-2$  since it is signed. When we add a 1 to “r1,” it evaluates to 255 ( $254 + 1$ ). But when we add a 1 to “sr1,” it will be  $-1$  ( $-2 + 1$ ). This exemplifies how signed and unsigned variables are evaluated and interpreted.

Here’s how signed and unsigned variables are interpreted.

```
logic [7:0] r1;
```



```
logic signed [7:0] sr1;
```



You can also declare wires and ports as signed. Note that logic, reg, wire, inputs, and outputs are unsigned by default:

```
wire signed [7:0] w;
module sm (input signed [7:0] iBus, output logic signed [7:0] oBus);
```

Here are some more examples:

```
logic signed [3:0] sr = -1; ( sr = 4'sb1111)
logic signed [7:0] sr1 = 1; (sr1 = 8'sb00000001)
logic [7:0] adds = sr + sr1; ( adds = 8'b00000000)
adds = sr; (adds = 8'b11111111);

logic [7:0] usr = 1;
logic signed [7:0] s_add;
s_add = sr + usr;      (s_add = 15+1 = 8'sb00010000) (signed +
unsigned = unsigned; sr is treated as unsigned 15)
```

### 2.1.3 Bits vs. Bytes

As we know, a “bit” is unsigned, while a “byte” is signed. So, do you think the following two declarations are equivalent?

```
bit [7:0] aBit; // Note 'bit' is 2-state, unsigned
byte bByte; // Note 'byte' is 2-state, 8-bit signed integer
```

Answer is *no* because:

```
bit [7:0] aBit; // = 0 to 255
byte bByte; // = -128 to 127
```

So, you need to be careful in mixing bits with bytes because they have different polarities.

Similarly, do you think the following two statements are equivalent?

```
byte MEM_BYTES [256];
bit signed [7:0] MY_MEM_BYTES [256];
```

The answer is *yes*. This is because we explicitly declared “bit” to be signed. So, “bit signed [7:0]” is equivalent to a “byte.”

## 2.2 Real Data Types

The “real” data types are shown in Table 2.2. Variables of these three types are commonly called “real” variables. Note that the following is *not* allowed on these variables:

- Edge event controls (posedge, negedge, edge) applied to real variables
- Bit-select or part-select references of variables declared as real
- Real number index expressions of bit-select or part-select references of vectors

**Table 2.2** “real” data types

Real data type	Description
real	The “real” data type is 64-bit
shortreal	The “shortreal” data type is 32-bit
realtime	The “realtime” declarations is treated synonymously with “real” declarations and can be used interchangeably

Also, note the following when converting real to integer or vice versa.

Conversion from “real” to “integer”:

- Real numbers are converted to integers by truncating the real number to the nearest integer.

Conversion from “integer” to “real”:

- Individual bits that are “x” or “z” in the net or the variable are treated as zero upon conversion.

### 2.2.1 “real” Data-Type Conversion Functions

There are also system functions available that allow for conversion from “real” and “shortreal” to “integer,” “bit,” and vice versa. This is shown in Table 2.3. The table describes each conversion function.

Let us look at an example of how these functions work:

```
module datatype1;
    real real1, real2, real3;
    integer i1;
    bit [63:0] bit1;
    initial begin
        real1 = 123.45;
        i1 = $rtoi(real1);
        real2 = $itor(i1);
        bit1 = $realtobits ( real1);
        real3 = $bitstoreal(bit1);
    end
    initial begin
        #10;
        $display("real1 = %f real2 = %f i1=%0d",real1,real2,i1);
        $display("bit1 = %b real3=%f",bit1,real3);
        #10 $finish(2);
    end
endmodule
```

In this example, we assign 123.45 to “real1” which is of type “real.” Then we use conversion functions on that value.

Here is the *simulation log*:

```
real1 = 123.450000
i1=123
real2 = 123.000000
```

**Table 2.3** “real” data-type conversion functions

Conversion function	Description
integer \$rtoi ( real_val )	\$rtoi converts real values to an integer type by truncating the real value to the nearest integer (e.g., 123.45 becomes 123). But note that \$rtoi differs from casting a real value to an integer or other integral types, in that casting will perform rounding instead of truncation
real \$itor ( int_val )	\$itor converts integer values to real values (e.g., 123 becomes 123.0)
[63:0] \$realtobits ( real_val )	Converts values from a real type to a 64-bit vector representation of the real number
real \$bitstoreal ( bit_val )	Converts a bit pattern created by \$realtobits to a value of the real type
[31:0] \$shortrealtobits ( shortreal_val )	Converts values from a shortreal type to the 32-bit vector representation of the real number
shortreal \$bitstoshortreal ( bit_val )	Converts a bit pattern created by \$shortrealtobits to a value of the shortreal type

First, we take “real1 = 123.45” and convert it to an integer using the \$rtoi conversion function. Since \$rtoi truncates the real value to integer, we see in the simulation log that “i1” is equal to 123 (0.45 is truncated).

Next, we take the converted integer value and reconvert it to real (`real2 = $itor(i1);`). This produces `real2 = 123.000000`.

Next, we take the “real1” value and convert it to bits using “bit1 = \$realtobits (real1);.” This conversion function produces a 64-bit vector representation of the real value. This is shown in simulation log as “bit1 = 010000001011110110111001100110011001100110011001101.”

Next we take this 64-bit converted value and convert it back to real type (`real3 = $bitstoreal(bit1);`). This produces `real3 = 123.45`. So, no precision was lost when going from `$realtobits` and then back to `$bitstoreal`.

### 2.3 Nets

Nets are used to connect elements such as logic gates. As such, they do not store any value. They simply make a connection from the driving logic to the receiving logic or simply pull a net high or low- or high-impedance state.

Let us look at the truth table of each of this type that will explain their functionality. Table 2.4 shows net data types.

### 2.3.1 “wire” and “tri”

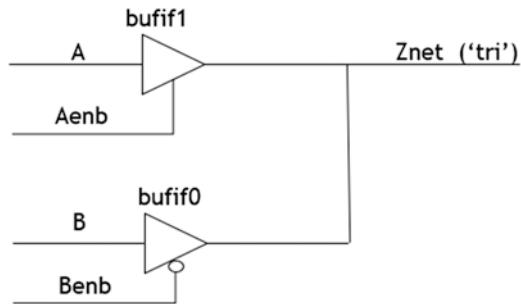
- The *wire* and *tri* nets connect elements. They do not store any value.
- The net types *wire* and *tri* are identical in their syntax and functions.
- Two names are provided so that the name of a net can indicate the purpose of the net in that model.
- A *wire* net can be used for nets that are driven by a single gate or continuous assignment. But note that this is simulator/methodology dependent. A “wire” that is driven by multiple nets may be caught by a lint type tool to catch a multiply driven “wire.” Most methodologies restrict “wire” to have a single driver.
- The *tri* net type can be used where multiple drivers drive a net.

Here is a simple example showing how tristate logic works.

In Fig. 2.1, we show a typical use of “tri” net. There are two drivers, namely, buif0 and bufif1. They both drive a single “tri” net called Znet. Bufif1 drives the net “A” when Aenb is “1,” and bufif0 drives the net “B” when Benb is “0.” So, when “A” = 1 and “B” = 0, there will be conflict on the net Znet, and the result will be “X” (unknown). But when “A” = 0 and “B” = 1, both drivers will be disabled and Znet will be equal to “Z.” In other words, Znet is a resolved type net. It will resolve the driven values by two (or more) drives and produce the correct result. This is in

**Table 2.4** Nets

Net	Description
wire	A high impedance net; multi-driver net
tri	A high impedance net; multi-driver net
tri0	Resistive pulldown net
tri1	Resistive pullup net
trior	Same as “wor”; “1” wins in all cases; multi-driver net
triand	Same as “wand”; “0” wins in all cases; multi-driver net
trireg	Models charge storage node
uwire	Unresolved type; allows only one driver on the net
wand	Same as “triand”; “0” wins in all cases
wor	Same as trior; “1” wins in all cases
supply0	Net with supply strength to model “gnd”
supply1	Net with supply strength to model “power”

**Fig. 2.1** Tristate logic

contrast to “reg” or “logic” type which are unresolved type (they will cause a race condition when multiple drivers drive from concurrently running logic processes).

Here is the SystemVerilog code and testbench for the circuit in Fig. 2.1:

```

module trinet;
    logic A, B, Aenb, Benb;
    tri Znet;

    bufif1 if1(Znet, A, Aenb); //bufif1 driver
    bufif0 if0(Znet, B, Benb); //bufif0 driver

    initial begin
        A = 1; Aenb = 1; //Drive bufif1 with 1
        B = 0; Benb = 0; //Drive bufif0 with 0
        #10;
        Aenb = 0; //disable bufif1
        Benb = 1; //disable bufif0
    end

    initial begin
        $monitor ($stime ,,"A=%0d Aenb=%0d B=%0d Benb = %0d
Znet=%0d", A, Aenb, B, Benb, Znet);
    end

endmodule
  
```

*Simulation log:*

```

0 A=1 Aenb=1 B=0 Benb = 0 Znet=x // drivers with opposite values
                                         // hence Znet='x'

10 A=1 Aenb=0 B=0 Benb = 1 Znet=z //drivers disabled - hence Znet='z'
V C S S i m u l a t i o n R e p o r t
  
```

As you notice from the code, when both bufif0 and bufif1 drive opposite values, Znet = “x.” But when both are disabled, Znet = “z.”

*Note that in the preceding example, you can replace “tri” with “wire,” and you will get the same results. But, from a methodology point of view, you want to use “wire” only for singly driven nets.*

Here is the truth table for “wire/tri.” They both have the same functionality when they are driven by multiple drivers. A strongly driven “0” or “1” will always win over a high impedance “z” state, and whenever there are multiple conflicting value drivers, the result will be an “x.” “x” always wins over any other driven value. The truth table is shown in Table 2.5.

### 2.3.2 Unresolved “wire” Type: “uwire”

As we saw, both “wire” and “tri” allow multiple drivers. But what if you want to restrict “wire” to only single driver? What if you want to see an error when multiple drivers drive a “wire”? Well, that is where unresolved “uwire” type comes into picture.

The keyword to declare an unresolved wire is “uwire.” It is an unresolved or unidriver wire and is used to model nets that allow only a single driver. It is an error to connect any bit of a uwire net to more than one driver. It is also an error to connect a uwire net to a bidirectional terminal of a bidirectional pass switch.

Here is a simple example:

```
module init;
    uwire w1;
    logic enable, a, b;

    assign w1 = enable ? a:1'bz;
    assign w1 = !enable ? b:1'bz;
endmodule
```

In this example, uwire “w1” is driven by multiple drivers. If “w1” was declared as “wire,” this would be perfectly normal and accepted. But since “w1” is declared as uwire, you will get the following error from the simulator:

**Table 2.5** wire/tri truth table

wire/tri	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

```
# SLP: Elaboration phase ...
# SLP: Error: testbench.sv (5): Net 'w1' of the uwire type cannot be driven by multiple drivers.
# SLP: Fatal Error: Cannot continue elaboration due to previous errors
```

“uwire” is an excellent way to enforce a single driver “wire” methodology:

### 2.3.3 Resolved vs. Unresolved Type

Also, note that, in contrast to net “wire,” variables are neither resolved nor unresolved, and net types are resolved (except when you declare a “wire” as a “uwire” as we just saw). Here is a simple example to show how the code will behave differently for a variable vs. a net. In short, a variable will cause a race condition when driven from more than one procedural block, while a “wire” will behave with correct net resolution. Here is a simple code snippet:

```
module init;
    wire w1; //resolved type
    logic enable, a, b, varC; //unresolved type

    assign w1 = enable ? a:1'bz; //w1 is resolved type
    assign w1 = enable ? b:1'bz;

    initial begin
        a = 1'b1; b=1'b0;
        enable = 1'b1;
    end

    always @(enable or b) begin
        if (enable) varC = b; //race condition
        else varC = 1'bz;
    end
    always @* begin
        if (enable) varC = a; //race condition
        else varC = 1'bz;
    end
endmodule
```

As you notice from the above example, “varC” is driven from two procedural always blocks. There is no guarantee which assignment will win! This is considered a race condition and should be avoided at all cost. *Different simulators will give you different results and your logic will be unstable.* “w1” is also driven by two nets, but it will resolve according to Table 2.5.

### 2.3.4 “wand” and “triand”

The wand and triand nets will create *wired and* configurations so that if any driver is 0, the value of the net is 0.

Table 2.6 shows the truth table for “wand”/“triand.”

### 2.3.5 “wor” and “trior”

The wor and trior nets will create *wired or* configurations so that when any of the drivers is 1, the resulting value of the net is 1. Table 2.7 shows the truth table for “wor”/“trior.”

### 2.3.6 “tri0” and “tri1”

The tri0 and tri1 nets model nets with resistive *pulldown* and resistive *pullup* devices on them:

- When no driver drives a tri0 net, its value is 0 with strength pull.
- When no driver drives a tri1 net, its value is 1 with strength pull.

Following truth tables model multiple drivers of strength strong on tri0 (Table 2.8) and tri1 (Table 2.9) nets. The resulting value on the net has strength strong, *unless both drivers are z, in which case the net has strength pull.*

Let us take the example that we discussed before but replace “tri” with “tri0.” The results will be as per the truth table in Table 2.8:

**Table 2.6** wand/triand truth table

wand/triand	0	1	X	Z
0	0	0	0	0
1	0	0	X	1
X	0	X	X	X
Z	0	1	X	Z

**Table 2.7** “wor”/“trior” truth table

wor/trior	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z

**Table 2.8** tri0 truth table

<i>tri0</i>	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	<b>0 (pull strength)</b>

**Table 2.9** tri1 truth table

<i>tri1</i>	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	<b>1 (pull strength)</b>

```

module trinet;
    logic A, B, Aenb, Benb;
    tri0 Znet; //Note Znet is declared as 'tri0'

    bufif1 if1(Znet, A, Aenb);
    bufif0 if0(Znet, B, Benb);

    initial begin
        A = 1; Aenb = 1;
        B = 0; Benb = 0;
        #10;
        Aenb = 0; //disable bufif1
        Benb = 1; //disable bufif0
    end
    initial begin
        $monitor ($stime ,,"A=%0d Aenb=%0d B=%0d Benb = %0d
Znet=%0d", A, Aenb, B, Benb, Znet);
    end
endmodule

```

*Simulation log:*

0 A=1 Aenb=1 B=0 Benb = 0 Znet=x

10 A=1 Aenb=0 B=0 Benb = 1 Znet=0

V C S S i m u l a t i o n R e p o r t

As you notice, at time 10, when both drivers are disabled (meaning they both drive “z” on Znet), the resultant value on Znet is “0” (and not “z” as we saw when Znet was declared as “tri”).

## 2.4 Drive Strengths

The nets and primitives can be assigned drive strengths. They are defined as follows:

Level	Keyword		Context
7	supply0	supply1	Primitive/assign
6 (default)	strong0	strong1	Primitive/assign
5	pull0	pull1	Primitive/assign
4	large		trireg
3	weak0	weak1	Primitive/assign
2	medium		trireg
1	small		trireg
0	highz0	highz1	Primitive/assign

The strength with Level 7 has the highest strength and then in decreasing order. Strong0 and strong1 are the default strengths. So, for example, you have the following assign driving the same net “wr1”:

```
assign (strong0, weak1) wr1 = a;
assign (pull0, pull1) wr1 = b;
```

If “a” is 0 and “b” is 1, then strong0 (in the first assignment) will win over pull1 (in the second assignment), and the final result value on “wr1” will be strong0 since strong0 has higher strength than pull1.

Here is an example:

```
module top ();
    tri0 t0;
    tri t2;
    trireg (large) #(0,0,25) trg;
    wand w1; //same as triand
    wire w2;
    supply0 s1;
    supply1 s2;
    reg a, b, c, d, e, f, enb;

    buf b1 (w1, a);
    buf b2 (w1, b);
```

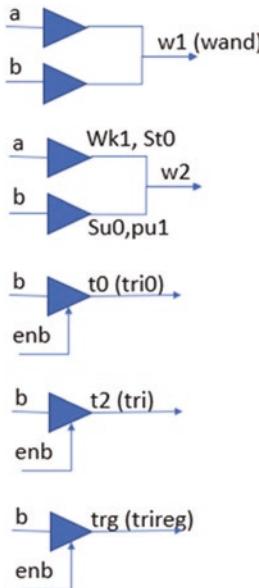
```

buf (weak1, strong0) (w2, a);
buf (supply0, pull1) (w2, b);

bufif1 (t0, b, enb);
bufif1 (t2, b, enb);
bufif1 (trg, b, enb);
initial begin
  a = 0; b = 1; enb=0;
  #10; enb = 1;
  #10; enb=0;
  #10;
end
initial $monitor($stime ,,"a=%b b=%b enb=%b wand-w1=%v wor-
w2=%v tri0-t0=%v tri-t2=%v trg=%v s1=%v s2=%v",a,b,enb,w1,w2,t0,t
2,trg,s1,s2);
endmodule

```

Here is the pictorial representation of the above model.



#### *Simulation log:*

# run -all

# 0 a=0 b=1 enb=0 wand-w1=St0 w2=St0 tri0-t0=Pu0 tri-t2=HiZ trg=LaX  
**s1=Su0 s2=Su1**

```
# 10 a=0 b=1 enb=1 wand-w1=St0 w2=St0 tri0-t0=St1 tri-t2=St1 trg=St1
    s1=Su0 s2=Su1

# 20 a=0 b=1 enb=0 wand-w1=St0 w2=St0 tri0-t0=Pu0 tri-t2=HiZ trg=La1
    s1=Su0 s2=Su1

# *** Warning: (vsim-3466) [DECAY] - Charge on node '/top/trg' has decayed.
# Time: 45 ns Iteration: 0 Instance: /top File: testbench.sv Line: 5

# 45 a=0 b=1 enb=0 wand-w1=St0 w2=St0 tri0-t0=Pu0 tri-t2=HiZ trg=LaX
    s1=Su0 s2=Su1
```

“w1” is declared as “wand” (which is the same as “triand”). We drive “w1” as follows:

```
buf b1 (w1, a);
buf b2 (w1, b);
```

“a” is driven with 0 and “b” is driven with 1. Since this is a wand, 0 will win over 1, and “w1” will be St0 as shown in the simulation log at time 0. Note that it is St0 which is the default strength.

Then we drive “w2” as follows:

```
buf (weak1, strong0) (w2, a);
buf (supply0, pull1) (w2, b);
```

and again a = 0 and b = 1. So, strong0 wins over pull1 and the resultant value on “w2” will be St0.

There are three bufif1 in the model and their corresponding nets are declared as follows:

```
bufif1 (t0, b, enb);
bufif1 (t2, b, enb);
bufif1 (trg, b, enb);
tri0 t0;
tri t2;
trireg (large) #(0,0,25) trg;
```

First we drive enb = 1, meaning the tristate buffers are enabled. Since “b” (input to tristate buffers) is 1, t0 will be St1. t2 will also be “St1” and trg will also be St1.

Then, we disable the tristate buffers (enb = 0). Now, t0 will be Pu0 since “tri0” when disabled will pull down the net with a strength of pull. t2 will be HighZ because it is a tristate net, and trg will be La1 since it is a capacitor with “large” strength.

Note that for “trg” the charge decay time is 25. So, once “trg” is driven to La1, it will decay out in 25 time units. That is shown as a warning in the simulation log.

After 25 time units, “trg” will be driven to an unknown state with strength of “large” (LaX).

## 2.5 Variable vs. Net

There is a subtle difference between variables and nets. Variables are assigned through procedural blocks such as “initial,” “always,” “task,” and “function” or through continuous assignments. They are the left-hand side of the assignment as shown below:

```
logic [31:0] bus;
bit dataEnb;
always @(dataEnb)
begin
    bus = 32'h0;
end
```

Here, “bus” is a variable of type “logic.” It can only be assigned through a procedural block (“always” block in this case) or as the left-hand side of a continuous assignment. Variables can be written by one or more procedural statements, including procedural continuous assignments. The last write determines the value. You can also write a variable from different procedural blocks. *But* be careful not to write to the same variable from multiple procedural blocks, you may end up creating a race condition.

The rule is that it is an error to have multiple continuous assignments or a mixture of procedural and continuous assignments writing to a variable. Again, you will create a race condition.

In contrast, a net is a wire and can only be driven by continuous assignment or as output of gates (i.e., they are connecting nets between gates, output connected to input). *Nets cannot be driven from a procedural block.* A net can be written by one or more continuous assignments, by primitive outputs, or through module ports. The resultant value of multiple drivers is determined by the resolution function of the net (we will see that in upcoming sections). If a net on one side of a port is driven by a variable on the other side, a continuous assignment is implied. Note that a force statement can override the value of a net from a procedural block. When released, the net returns to the resolved value. *I highly recommend that you do – NOT – use a “force” for any kind of assignment in your code (nets or variables). Believe me, it will come back to haunt you.*

The following is driven by continuous assignment:

```
wire [31:0] wBus;
assign wBus = wBusInput; //continuous assignment driving a net
```

OR as output of gate logic:

```
wire y;
and g1 (y, a, b); //net type driven by output of a gate
```

Note that multiple continuous assignments to the same net are illegal. For example, the following is illegal:

```
assign C = sel ? p1 : p2;
assign C = sel ? p3 : p4;
```

## 2.6 “var”

Note that there is also a keyword “var” available when declaring variables. If a “var” is declared, the data type is optional and will be implicitly declared as of type “logic.” For example:

```
var byte vByte; // same as "byte vByte"
var A; //same as "var logic A"
      // "var logic A" is recommended coding practice
var [31:0] address; //same as "var logic [31:0] address"
```

## 2.7 Variable and Net Initialization

This section described how variables and nets behave at time “0” with their default values as well as assigned values in their declaration.

Here are the default initial values of variables and nets:

Type	Default initial value
4-state variable	‘x
2-state variable	‘0
real, shortreal	‘0.0
wire/tri	‘z

You must be careful on how these default initial values affect your testbench logic. Let us look at an example:

```

module init;
    logic clk1;          //uninitialized = 'x
    wire w1 = '0;        //Continuous assignment

    initial begin
        clk1 = 1;   //NO 'x to '1 transition at time 0
        @(posedge clk1);
        $display("posedge clk1 detected");
    end

    initial begin
        @(negedge w1); //'z to '0 transition detected
        $display("negedge w1 detected");
    end
endmodule

```

*Simulation log:*

```

ncsim> run
negedge w1 detected

```

ncsim: \*W,RNQUIE: Simulation is complete.

Here is what is going on.

We have not initialized `clk1` with “`logic clk1`” declaration – so it takes on the default value of ‘x. With ‘`wire w1 = “0”`, we have initialized “`wire w1`” to an initial value of “0.” Now, in the procedural code, let us try to see how these initial values trigger (or not) an event at time 0.

In the first initial block, we assign a “1” to `clk1`. This is to see if we get a posedge event at time 0 – because `clk1` would transition from ‘x to ‘1 at time 0. In other words, is there a time 0 posedge event on `clk1`? The answer is *no*. The simulator does not detect a posedge event even though we may think that `clk1` goes from default ‘x to initial value ‘1. This is shown in the simulation log. Note that “`clk1`” is uninitialized. What would happen if it was explicitly initialized to value “0”? Will you get posedge event on “`clk1`”? Try it out.

Now, “`wire w1`” is initialized to “0.” The default value of a “`wire`” is ‘z. So, do we see a transition from ‘z to ‘0 at time 0? The answer is *yes*. The nets seem to behave differently from the variables when it comes to time 0 event. So, keep this in mind in order to understand initial time events.

*Disclaimer: You may get different results from different simulators for time 0 events.*

*The above simulation was done using Cadence’s NCSIM. So, take my example with a grain of salt.*

Static variable initializations happen at time 0 before any processes begin (initial/always/continuous assignments). A net declaration assignment is just a shortcut for a declaration and a separate continuous assignment.

Unlike nets, a variable cannot have an implicit continuous assignment as part of its declaration. *An assignment as part of the declaration of a variable is a variable initialization, not a continuous assignment.*

For example:

```
wire w = a & b; // net with a continuous assignment
logic v = consta & constb; // variable with initialization
logic vw; // no initial assignment
assign vw = vara & varb; // continuous assignment to a variable
```

## 2.8 Static, Automatic, and Local Variables

This section is mainly to focus on different forms of variables available in the language.

Fundamentally, three forms of variable are available, namely, static, automatic, and local. These are described in Table 2.10.

### 2.8.1 Static vs. Local Variables

Let us see the difference between a static variable and a local one. Here is an example to illustrate that:

**Table 2.10** Variables

Variable	Description
Static variable	Allocated and initialized at time 0 Exists for the entire simulation Variables declared outside a module Variables in program, interface, checker, task, or function are local to the compilation unit and have a static lifetime
Automatic variable	Needed for recursive and reentrant tasks and functions Reallocated and initialized each time reentering the block Automatic variables cannot be written with non-blocking, continuous, or procedural continuous assignments References to automatic variables are limited to procedural blocks
Local variable	Accessible at the scope where they are defined and scopes below Accessible from outside the scope with hierarchical names

```

static int n; // Static variable - outside 'module' -
              // globally declared
              //visible to all modules/scopes that follow.

module vars;
int n;
      //Local variable - Module level - visible to all
scopes below

initial begin
  n = 2;
  $display("module level 'n' = %0d",n);
end

initial begin : init2
  int n; //Local - Block level
  n = 3;
  $display("block level 'n' = %0d",n);

  $unit::n = 4; //Static Global
  $display("Statically declared 'n' = %0d",$unit::n);
end

initial begin //hierarchical reference to local variable
  $display("init2.n = %0d", init2.n);
end
endmodule

module next;
  //Static variable 'n' is visible in the module 'next'
  initial begin
    $display("Statically declared 'n' in module 'next' =
%0d",$unit::n);
  end
endmodule

```

*Simulation log:*

```

module level 'n' = 2
block level 'n' = 3
Statically declared 'n' = 4
init2.n = 3
Statically declared 'n' in module 'next' = 4
V C S  S i m u l a t i o n  R e p o r t

```

First, we declare a static var. “int n” outside of the module ‘vars’.

*This makes it visible to all the modules that follow (note that when I use the term “global” it does not mean a global variable. It is just a static variable visible to all the modules that follow. It is a static variable declared outside of a module. There are no global variables in SystemVerilog). You can have variables declared in the compilation unit scope \$unit, but there can be multiple \$units. You can never access things in one compilation unit from another.*

Then, we declare another variable “int n” inside the module “vars” and assign it a value 2. This makes it a module-level var., visible to all the blocks under the module scope. Then we declare another variable “int n” within the named block “init2” and assign it the value 3. This makes it local only to that named block (except for hierarchical reference to it). Then we display the module level “n” vs. the block (“init2” block)-level “n.” We get the following in simulation log. The module-level and block-level “n” are in different scopes, and one does not clobber/override another:

```
module level n = 2
block level n = 3
```

In order to assign a value to the statically declared “n,” we need to use \$unit. This is done using “\$unit :: n = 4;.” *Note that this assignment will now be visible to “all” modules that follow the declaration of the statically declared variable.*

First, we display this variable using \$unit :: and get the required result “Global n = 4” as shown in the simulation log.

Then, we show that the “init2” block-level “n” can indeed be accessed hierarchically from other procedural blocks. This is shown by the following code:

```
initial begin //hierarchical reference to local variable
    $display("init2.n = %0d", init2.n);
end
```

And the simulation log shows this display as “init2.n = 3.”

Finally, we declare another module “next.” This is to show that the globally declared “n” (preceding the module “vars”) is indeed accessible across module boundaries. When we display “n” from module “next,” we get the desired result (i.e., the value assigned to \$unit :: n in module “vars”). This is shown in the simulation log as

Statically declared 'n' in module 'next' = 4

### 2.8.2 Automatic vs. Static Variable

Automatic variables are reallocated and initialized each time entering the block. This allows for reentrancy in SystemVerilog. In contrast, static variables are allocated and initialized once and do not reset to their value each time you enter a block. Here is an example to illustrate this point:

```

module autovars;

initial begin
    for (int i=0; i<2; i++) begin
        automatic int loop3 = 0; // executes every loop
        for (int j=0; j<2; j++) begin
            loop3++;
            $display("loop3=%0d",loop3);
        end
    end // loop3 = 1 2 1 2

    for (int i=0; i<2; i++) begin
        static int loop2 = 0; // executes once at time zero
        for (int j=0; j<2; j++) begin
            loop2++;
            $display("loop2=%0d",loop2);
        end
    end // loop2 = 1 2 3 4
end
endmodule : autovars

```

In this example, there are two *for* loops. The first *for* loop declares an “automatic” variable called “loop3” initialized to 0. Note the keyword *automatic* in its declaration. The second *for* loop declares a “static” variable called *loop2* initialized to 0. Note the keyword “*static*” in its declaration.

Here is the simulation log. I will explain it right after the log:

```

loop3=1
loop3=2
loop3=1
loop3=2
loop2=1
loop2=2
loop2=3
loop2=4

```

In the first for loop, the automatic variable will reinitialize itself every time you reenter the loop. It is automatic and allows for reentrant code. The variable will reinitialize to 0 every time you enter the external for loop. This way when *loop3* count is incremented, it will start from 0 and increment by 1 and 2. It will not clobber the initial value. That is why you see the following in the simulation log:

```

loop3=1
loop3=2

```

```
loop3=1
loop3=2
```

In contrast, the second *for* loop's static variable will – not – reinitialize itself every time you enter the *for* loop. It's initialized once and then increment successively until both *for* loops are over. Hence, it will increment as follows:

```
loop2=1
loop2=2
loop2=3
loop2=4
```

This example illustrates the difference between an automatic and a static variable. I will further explain the nuances of SystemVerilog's reentrant (or lack thereof) behavior when it comes to reentering the procedural “always” block. Is an “always” block reentrant (my favorite interview question)? Hold on to your hats.

Note also that we have not yet discussed “automatic” tasks or functions. That is another topic. But in brief, variables declared in an automatic task, function, or block are local in scope, are default to the lifetime of the call or block, and are initialized on each entry to the call or block. An automatic block is one in which declarations are automatic by default. Also, specific variables within an automatic task, function, or block can be explicitly declared as static. We will see examples on how automatic tasks and functions work in later sections.

### 2.8.3 Variable Lifetimes

SystemVerilog has three different kinds of variable lifetimes:

- *Static* – exists for the entire life of the simulation. Initialized once at time 0. Can be referenced from outside the scope of where it is declared.
- *Automatic* – a new instance gets created and initialized for each entry to the scope where it is declared (must be a procedural scope). Its lifetime ends when exiting the scope, and all nested scopes exit (to handle fork/join\_none). Can only be referenced from within the scope where it is declared.
- *Dynamic* – created by the execution of a procedural statement. Its lifetime can end a number of ways but normally by executing a procedural statement.

Dynamically sized arrays have a compound concept of lifetimes. Individual elements have dynamic lifetimes, but the array as a whole aggregate can have any of the above lifetimes. If we consider the array as an aggregate, it means that whenever the lifetime of an array variable ends, all the dynamically allocated elements are reclaimed.

Please refer to Chap. 8 on class to better understand the following statement.

Class objects have dynamic lifetimes, but the class variables that hold handles referencing class objects can have any of the above lifetimes. But since more than

one class variable can reference the same class object, the lifetime of class object ends when there are no more class variables referencing that object. So, if that class object contains dynamic array variables, those variables lifetime end when the object lifetime ends.

SystemVerilog does not specify how garbage collection works. When the lifetime of something ends, you can no longer access it. There is no way to know when the memory actually gets reclaimed.

## 2.9 Enumerated Types

An enumerated type defines a set of named values. Enumerated variables can be declared without either a data type or data value(s). In the absence of a data-type declaration, the default data type will be *int*.

Let us look at some examples:

```
enum {red, green, blue} light1, light2;
```

In this example, no data type is specified, so the default “int” data type is assumed. In this example, light1 and light2 are defined to be variables of the default type *int* that includes three members: red, green, and blue. No values are specified. So, red = 0, green = 1, and blue = 2. *Values are assigned in an ascending order*:

```
enum integer {IDLE, XX='x, S1='b01, S2='b10} state, next;
```

Here the data type is “integer” which means you can assign “x” and “z” values to the enum members. Note that when values are assigned to the members of an enum type, *they must be in ascending order*. Here, since XX = ‘x, you have to explicitly assign values to S1 and S2 since the number system does not follow an “x” (you can’t go from “x” to “0” or “1”).

Here is an example that will cause an error since after XX = ‘x, you did not assign explicitly incrementing number to S1 (or S2). What values can follow “x”?

```
enum integer {IDLE, XX='x, S1, S2} state, next;
```

Simulator will give you following type syntax error (Synopsys – VCS):

*Syntax ERROR: Values assigned must be in ascending order (or assign explicit value to each member).*

Let us look at more examples:

```
enum {bronze=3, silver, gold} medal;
```

In this example, we start off with bronze = 3 as the starting value. Hence, silver will be equal to 4 and gold will be equal to 5:

```
enum {a=3, b=7, c} alphabet;
```

In this example we are assigning explicit values to “a” and “b.” But we do not assign any value to “c.” So, “c” will take the value that follows  $b = 7$ . So, “c” will be equal to 8:

```
enum {a=0, b=7, c, d=8} alphabet;
```

This will cause an *error* since “c” will take on value 8 because it follows  $b = 7$ . But then you assign  $d = 8$ . So, both “c” and “d” will have value 8. You cannot have duplicate values and hence the error. Here is the error report by Synopsys – VCS:

*Error-[ENUMDUP]* Duplicate labels in enum

The enum label 'd' has the value 4'd8 which is duplicate of enum label 'c' in the declared enum.

```
enum {a, b=7, c} alphabet;
```

In this example, “a” is the first member and it does not have a value. So, it will start with “0.” So, the values are  $a = 0$ ,  $b = 7$ , and  $c = 8$ :

```
enum bit [3:0] {red='h13, green, blue} color;
```

This will give an *error* because the enum is of type “bit [3:0]” and the value assigned to red ( $= 'h13$ ) is outside the range allowed by “bit [3:0].” Here is the error report by Synopsys – VCS:

*Error-[ENUMRANGE]* Enum label outside value range

The enum label 'red' has the value 'h00000013 which is outside the range of the base type of the declared enum, which is 4 bit unsigned.

```
enum bit [3:0] {red='d13, green, blue} color;
```

Similar to above example, but all the values fall in range of “bit [3:0].” So,  $red = 13$ ,  $green = 14$ , and  $blue = 15$ :

```
enum bit [3:0] {bronze=10, silver, gold=5} medal4;
```

Here,  $bronze = 10$ ,  $silver = 11$ , and  $gold = 5$ .

Point in this example is that when you explicitly assign values, they may not be in an ascending order:

```
enum bit [3:0] {red, green, blue=3'h5} color;
```

Here,  $red = 0$ ,  $green = 1$ , and  $blue = 5$ :

```
enum {color[4]} color_set;
```

Here, color0 = 0, color1 = 1, color2 = 2, and color3 = 3:

```
enum {color[3] = 5} color_set;
```

Here, color0 = 5, color1 = 6, and color2 = 7:

```
enum {color[3:5]} color_set;
```

Here, color3 = 0, color4 = 1, and color5 = 2.

Enums can be especially useful in design of finite-state machines. For example, you can define an enum as follows to describe the states of a state machine:

```
enum logic [1:0] { IDLE = 2'b00,
                   READ = 2'b01,
                   WRITE = 2'b10,
                   RMW = 2'b11,
                   ILLEGAL = 'x } current_state, next_state;
```

Since the type is “logic,” you can assign an unknown ('x) value to an enum member. Such 'x assignment is very useful for simulation debug, and synthesis doesn’t care optimization. This enum can then be used in state machine coding as follows:

```
always @ (posedge clk, negedge reset)
    if (!reset) current_state <= IDLE;
    else current_state = next_state;
always @* begin
    ...
    case (current_state)
        IDLE : if (rdy) next_state = READ;
        READ : if (go) next_state = WRITE;
    ...
endcase
end
```

### 2.9.1 Enumerated-Type Methods

Enumerated-type methods are offered by the language which facilitates extracting values of the members of the enum type. Table 2.11 describes these methods.

Let us look at an example on how these methods work:

```

module enum_methods;

    typedef enum { red, green, blue, yellow } Colors;
    Colors c;

    initial begin
        $display("Number of members in Colors = %0d",c.num);

        c = c.first();
        $display("First member # = %0d",c);

        c = c.next(2);
        $display("c = %0d",c);

        c = c.last();
        $display("Last member # = %0d",c);

        $display( "%s : %0d", c.name, c );
    end
endmodule

```

*Simulation log:*

```

run -all;
# KERNEL: Number of members in Colors = 4
# KERNEL: First member # = 0
# KERNEL: c = 2

```

**Table 2.11** Enumerated-type methods

Method	Description
first()	The first() method returns the value of the first member of the enumeration: <b>function enum first();</b>
last()	The last() method returns the value of the last member of the enumeration: <b>function enum last();</b>
next()	The next() method returns the nth next enumeration value (default is the next one) starting from the current value of the given variable. A wrap to the start of the enumeration occurs when the end of the enumeration is reached: <b>function enum next(int unsigned N = 1);</b>
prev()	The prev() method returns the nth previous enumeration value (default is the previous one) starting from the current value of the given variable: <b>function enum prev( int unsigned N = 1 );</b>
num()	The num() method returns the number of elements in the given enumeration: <b>function int num();</b>
name()	The name() method returns the string representation of the given enumeration value: <b>function string name();</b>

```
# KERNEL: Last member # = 3
# KERNEL: yellow : 3
# KERNEL: Simulation has finished. There are no more test vectors to simulate.
exit
```

In this example, we define a “typedef enum { red, green, blue, yellow } Colors;” It has four members, with values, red = 0, green = 1, blue = 2, and yellow = 3.

We first display the number of members in the enum using the method “num().” There are four members in the enum. So, the display log shows “Number of members in Colors = 4.”

We then get the value of the first member using the method “first().” As we see that the first member is “red” with a value 0, we see that in the display “First member = 0.”

We then get the “next” value of the enum. But we are using “next(2).” We are at the first member “red” which has a value of 0. So, the next(2) will give us the value 2, as shown in the simulation log.

Next, we get the value of the last member of the enum. That is value 3, as shown in the simulation log “Last member # = 3.”

Now, we are at the last member “yellow.” So, when we display “c.name,” we get the name of the member as “yellow” and its value as 3, as shown in the simulation log “yellow : 3.”

Here is another simple example. Comments describe the functionality:

```
typedef enum { red, green, blue, yellow, white, black } Colors;
Colors col;
integer a, b;
a = blue * 3; // a = 6 (since blue is in 3rd position; blue= 2)
               // (2*3=6)
a = yellow;    //a = 3;
b = a + green; //b = 3 + 1 = 4
```

One more example:

```
module datatype1;

    enum bit [3:0] {red, green, blue=5} color;
    typedef enum { read=10, write[5], intr[6:8] } E1;

    E1 e1 = write0; //initialize with a member of the enum

    int i1;

    initial
```

```

begin

    i1 = green;
    $display("i1 = %0d",i1);

    $display ("e1.name=%s",e1.name);

    e1 = e1.last ( );
    $display ("e1.last=%0d e1.name=%s",e1, e1.name);

    $display ("color.name = %s", color.name);

    $display ("red=%s green=%d blue=%d",color.
name,green,blue); //OK
    $display ("red=%d green=%d blue=%d",red,green,blue); //OK

    $display ("write0=%0d write1=%0d write2=%0d write3=%0d
write4=%0d", write0, writel, write2, write3, write4);

end
endmodule

```

*Simulation log:*

```

i1 = 1
e1.name=write0
e1.last =18 e1.name=intr8
color.name = red
red=red green= 1 blue= 5
red= 0 green= 1 blue= 5
write0=11 write1=12 write2=13 write3=14 write4=15
      V C S   S i m u l a t i o n   R e p o r t

```

A few things to note. We can initialize an enum type with a member of the enum, as in:

```
E1 e1 = write0; //initialize with a member of the enum
```

Then, when we display the name of the enum, we'll get the initialized value as the name:

```
$display ("e1.name=%s",e1.name);
```

Simulation log shows:

e1.name=write0

Let us look at the following code:

```
typedef enum { read=10, write[5], intr[6:8] } E1;
e1 = e1.last ( );
$display ("e1.last=%0d e1.name=%s",e1,e1.name);
```

The value of the last member of the enum E1 is 18. That's because the first member "read = 10" is followed by "write[5]" (five elements); so,  $10 + 5 = 15$ . Then, "intr[6:8]" (three elements), so  $15 + 3 = 18$ . That is shown in the simulation log. And the last member is "intr8."

In the simulation log:

e1.last =18 e1.name=intr8

Study the rest of the simulation log carefully to see what is going on.

## 2.9.2 Enumerated Type with Ranges

Table 2.12 shows that you can specify a range with a member of the enumerated type. Instead of declaring each member of a range separately, you can simply specify the required range with the member name. The "name" in Table 2.12 means the

**Table 2.12** Enumeration element ranges

Member name of the enumerated type	Description
name	Associates the next consecutive number with name
name = C	Associates the constant C to name
name[N]	Generates N named constants in the sequence: name0, name1,..., nameN–1. N will be a Positive integral number
name[N] = C	Optionally, a constant can be assigned to the generated named constants to associate that constant to the first generated named constant; subsequent generated named constants are associated consecutive values N has to be a positive integral number
name[N:M]	Creates a sequence of named constants starting with nameN and incrementing or decrementing until reaching named constant nameM N and M have to be nonnegative integral numbers
name[N:M] = C	Optionally, a constant can be assigned to the generated named constants to associate that constant to the first generated named constants; subsequent generated named constants are associated consecutive values N and M have to be nonnegative integral numbers

name of a member of the enumerated type. We will understand this better with an example right after we declare the table.

Here is an example of how these ranges with enumerated member work. We have so far seen member-assigned constants without any range:

```
module datatype1;
    typedef enum { read=10, write[5], intr[6:8] } cycle;
    enum { readreg[2] = 1, writereg[2:4] = 10 } reg0;

    initial
        begin
            $display ("read=%0d\n", read);
            $display ("write0=%0d write1=%0d write2=%0d write3=%0d
write4=%0d\n", write0,writel1,write2,write3,write4);
            $display ("intr6=%0d, intr7=%0d intr8=%0d\n",intr6,
intr7, intr8);

            $display ("readreg0=%0d readreg1=%0d\n",readreg0,
readreg1);
            $display ("writereg2=%0d writereg3=%0d writereg4=%0d\
n",writereg2, writereg3, writereg4);

        end
    endmodule
```

Here is how to interpret the example:

```
typedef enum { read=10, write[5], intr[6:8] } cycle;
```

defines an enum “cycle” which has three members. First “read”; then a range of five writes, namely, “write0,” “write1,” “write2,” “write3,” and “write4”; and then a range of three “intr,” namely, “intr6,” “intr7,” and “intr8.”

Similarly:

```
enum { readreg[2] = 1, writereg[2:4] = 10 } reg0;
```

defines an enum “reg0” with two members: first, readreg0 and readreg1 and second writereg2, writereg3, and writereg4.

Here is the simulation log that shows how the values are assigned to members with a range.

*Simulation log:*

read=10

write0=11 write1=12 write2=13 write3=14 write4=15

```

intr6=16, intr7=17 intr8=18
readreg0=1 readreg1=2
writereg2=10 writereg3=11 writereg4=12
V C S S i m u l a t i o n R e p o r t

```

The first line in simulation shows that read = 10 which is an explicit constant we assigned to “read.”

The next line shows the five writes (write[5]). As you notice, the numbers for five writes start with 11 because the previous member “read” was assigned constant 10. So, write0, write1, write2, write3, and write4 will be 11, 12, 13, 14, and 15, respectively.

The next line shows three “intr”s (intr[6:8]). Since the last value assigned was 15 (to write4), the numbers for “intr” will start with 16. Intr6, intr7, and intr8 will get the values 16, 17, and 18, respectively.

The next line shows two “readreg” (readreg[2]). Since we assigned a constant = 1 (readreg[2] = 1), to “readreg,” the values for readreg0 and readreg1 will be 1 and 2, respectively.

The last line shows three “writereg” (writereg[2:4]). Since we assigned a constant = 10 (writereg[2:4] = 10), the values for writereg2, writereg3, and writereg4 will be 10, 11, and 12, respectively.

## 2.10 User-Defined Type: Typedef

“typedef” simply means that a type name can be given so that the same type can be used in many places. Here is its definition:

```
typedef existing_type mytype;
```

For example:

```
typedef enum {NO, YES} boolean;
boolean myvar1, myvar2; // user-defined type
```

Essentially, if you want to define an enum type once and then use that type for many different vars, you use a typedef. Here is a simple example:

```

module tdef;
    typedef integer unsigned u_integer;
    typedef enum {RED, GREEN, BLUE} rgb;
    typedef bit [7:0] ubyte;

    u_integer uI = 32'h face_cafe;
    u_integer uI1 = 32'h cafe_face;
    rgb rgb_i = GREEN;

```

```

rgb rgb_i1 = BLUE;
ubyte cnt = 8'hFF;

initial begin
$display ("rgb_i=%s rgb_i1=%s uI=0x%0h uI1=0x%0h cnt=%0d",
rgb_i.name( ), rgb_i1.name( ), uI, uI1, cnt);
end
endmodule

```

*Simulation log:*

```

rgb_i=GREEN rgb_i1=BLUE uI=0xfacface uI1=0xcafeface cnt=255
VCS Simulation Report

```

Three different “typedef”s are declared in the model.

First typedef is “typedef integer unsigned u\_integer;.” We then declare two variables u1 and uI1 of type “u\_integer” and initialize them in their declaration:

```

u_integer uI = 32'h face_cafe;
u_integer uI1 = 32'h cafe_face;

```

Simulation log shows the values assigned to “u1” and “uI1.”

Second typedef is “typedef enum {RED, GREEN, BLUE} rgb;.” We declare two variables “rgb\_i” and “rgb\_i1” and initialize them with different enum members:

```

rgb rgb_i = GREEN;
rgb rgb_i1 = BLUE;

```

Simulation log shows the names assigned to “rgb\_i” and “rgb\_i1.”

Lastly, we have the typedef “typedef bit [7:0] ubyte;” and declare “cnt” of type “ubyte.” We assign it a value of 8'hff and display it.

Note that typedef is not restricted to enum types only. For example, you can declare the following.

Example 1:

```

typedef struct {
    bit [31:0] opcode;
    bit R_W;
    logic byteEnb;
    integer data;
    integer addr;
} read_cycle;
read_cycle rC;

```

Example 2:

```
typedef int data_t [3:0][7:0];
data_t a;
```

Example 3:

```
typedef int Qint[$];
Qint DynamicQ[ ]; // same as int DynamicQ[ ][ $];
```

Example 4:

```
typedef struct packed {
    bit [3:0] s1;
    bit s2;
} myStruct;

typedef union {
    logic [7:0] u1;
    myStruct b2;
} mUnionT;
mUnionT Union1;
```

*typedef* is used in many examples in the book.

## 2.11 String Data Type

The string data type is an ordered collection of characters. The length of a string variable is the number of characters in the collection. Variables of type string are dynamic as their length may vary during simulation. A single character of a string variable is of type byte.

Syntax:

**string** variable\_name [= initial\_value];

If an initial value is not specified in the declaration, the variable is initialized to “”, an empty string. An empty string has zero length.

Let us look at an example to see how strings work:

```

module datatype1;
    string s1 = "hello";
    string s2 = "hello world";
    string s3 = "hello\0world"; // \0 is ignored
    string s4, s5, s6;

initial
begin
    s4 = "later";
    s5 = "";           //empty string
    s6 = {"hi", s5}; //concatenation
#10;
$display ("s1=%s s2=%s s3=%s s4=%s s5=%s s6=%s", s1, s2,
s3, s4, s5, s6);
#100 $finish;
end
endmodule

```

In this example, we are declaring six strings, s1 through s6. s1, s2, and s3 are initialized to various string values. Note the escape character (\0) in string s3. Simulator will simply ignore the escape character and consider it an empty character. Then in the “initial” block, we assign s4 to a string “later” and s5 to an empty string and concatenate s5 with “hi” and assign it to string s6. Let us see what simulation results we get.

Here is the *simulation log*:

```

ncsim> run
s1=hello s2=hello world s3=helloworld s4=later s5= s6=hi
Simulation complete

```

In the simulation log, you see “s1 = hello” and “s2 = hello world” which are the strings assigned to them. But notice s3. The escape character is ignored, and the display shows “helloworld” as a single string with escape character becoming an empty string. S4 displays its assigned string “later,” while s5 displays an empty string. The concatenation of “hi” with s5 displays s6 = hi (since s5 is an empty string).

### 2.11.1 String Operators

String data type also offers operators that work on strings. These operations are shown in Table 2.13.

Let us look at an example to see how these operators work:

**Table 2.13** String operators

Operator	Description
Str1 == Str2	Equality. Checks whether the two string operands are equal. Result is 1 if they are equal and 0 if they are not
Str1 != Str2	Inequality
Str1 < Str2	Comparison: Relational operators return 1 if the corresponding condition is true using the lexicographic ordering of the two strings Str1 and Str2
Str1 <= Str2	
Str1 > Str2	
Str1 >= Str2	
{Str1,Str2,...,Strn}	Concatenation
Str[index]	<i>Indexing.</i> Returns a byte, the ASCII code at the given index. Indices range from 0 to N-1, where N is the number of characters in the string. If given an index out of range, returns 0
{multiplier{str}}	Replication

```

module datatype1;
    string s2 = "hello world";
    string s3 = "hello\0world";
    string s4, s5;
    string s6 = "compare";
    string s7 = "compare";
    string s8;
    integer s2len, s3len, s2c;

    initial begin
        #10; $display ("s2=%s s3=%s",s2,s3);
        #100 $finish;
    end

    initial begin
        #15;
        s2len = s2.len( ); $display("String Length s2 =
%0d",s2len);
        s3len = s3.len( ); $display("String Length s3 =
%0d",s3len);

        if (s2 == s3) $display("s2 = s3"); else $display("s2
!= s3");
        if (s6 == s7) $display("s6 = s7"); else $display("s6
!= s7");

        s4 = s2.substr(1,6);
    end

```

```

$display("s4 = %s", s4);

s5 = "later ";
s8 = {3{s5}};
$display("s8 = %s", s8);

s2c = s2[0];
$display("s2c = %s", s2c);
end
endmodule

```

In this example, we declare strings s2 through s8 and assign various character strings to these string data types. Note that strings s6 and s7 are assigned the same string for comparison purpose in the code. First, we assign string length (.len( )) to integers s2len and s3len. Then, we compare strings s2 and s3 and also strings s6 and s7. Then we extract a substring of s2 and assign it to s4. In the end we replicate s5 three times and assign it to s8. Let us look at simulation log to see how these assignments work.

*Simulation log:*

```

s2=hello world s3=helloworld
String Length s2 = 11
String Length s3 = 10
s2 != s3
s6 = s7
s4 = ello w
s8 = later later later
s2c = h

$finish at simulation time          110
V C S   S i m u l a t i o n   R e p o r t

```

The simulation log displays s2 and s3 as expected. The length of string s2 (“hello world”) is 11 as shown in the log. But length of s3 is 10, since it has an escape character which is ignored. We then compare s2 with s3 and see that they are different, and log displays that. S6 and s7 are the same and the display shows that. We extract a substring of s2 starting at position 1 and ending at position 6. So, the resultant string is “ello w.” Note that the first character in s2 string (“hello world”) is considered as position 0. Finally, we replicate s5 (“later”) three times and assign it to s8, which results in the string (“later later later”). Finally, we extract a character at string s2’s index 0 (s2c = s2[0];) which is character “h” as shown in the display.

## 2.11.2 String Methods

There are also several methods available to work with strings. We saw a couple of those in the preceding example (`.len()` and `.substr()`). The following describes other such methods (Table 2.14).

Here is an example that shows the usage of these methods:

**Table 2.14** String methods

Method	Description
<code>.len()</code>	<code>str.len()</code> – returns the length of the string: <i>function int len();</i>
<code>.putc(int i, byte c);</code>	<code>str.putc(i, c)</code> replaces the <i>i'th</i> character in “str” with the given integral value. Does not change the size of the string, simply replaces a character. Similar to <code>str[i] = c;</code> <i>function void putc(int i, byte c);</i>
<code>.getc(int i);</code>	<code>str.getc(i)</code> returns the ASCII code of the <i>i'th</i> character in “str.” Usage: <code>x = str.getc(i);</code> same as <code>x = str[i];</code> <i>function byte getc(int i);</i>
<code>toupper()</code>	<code>str.toupper()</code> converts characters in “str” to uppercase: <i>function string toupper();</i>
<code>tolower()</code>	<code>str.tolower()</code> converts characters in “str” to lowercase: <i>function string tolower();</i>
<code>compare()</code>	<code>str1.compare(str2)</code> compares str1 and str2 with regard to lexical ordering and return value: <i>function int compare(string s);</i>
<code>icompare()</code>	<code>str1.icompare(str2)</code> comparison is the same as for “ <code>compare()</code> ” but is case sensitive: <i>function int icompare(string s);</i>
<code>substr()</code>	<code>str.substr(i, j)</code> returns a new string that is a substring formed by characters in position <i>i</i> to <i>j</i> : <i>function string substr(int i, int j);</i>
<code>atoi(), atohex(), atooct(), atobin()</code>	<code>str atoi()</code> interprets the string as ASCII decimal string and returns the integer corresponding to the decimal representation in “str” <i>function integer atoi();</i> The function scans all leading digits and underscore ( <code>_</code> ) character, until it encounters any other character. Returns 0 if no digits are found. Similarly, <code>atohex()</code> interprets the string as hexadecimal: <i>function integer atohex();</i> <code>atooct</code> interprets the string as octal: <i>function integer atooct();</i> <code>atobin</code> interprets the string as binary: <i>function integer atobin();</i> An example later in the section clarifies these methods
<code>atoreal()</code>	<code>str.atoreal()</code> returns the real number corresponding the ASCII decimal representation in “str”: <i>function real atoreal();</i>

(Continue)

**Table 2.14** (Continue)

Method	Description
itoa ( ); hextoa ( ); octtoa ( ); bintoa ( ); realtoa ( );	Inverse of atoi ( ). str.itoa(i) stores the ASCII decimal representation of i into “str”: <i>function void itoa (integer i);</i> hextoa ( ) is inverse of atohex ( ). <i>function void hextoa (integer i);</i> octtoa ( ) is inverse of atooct ( ). <i>function void octtoa (integer i);</i> bintoa ( ) is inverse of atobin ( ). <i>function void bintoa (integer i);</i> realtoa ( ) is inverse of atoreal ( ). <i>function void realtoa ( real r);</i>

```

module sMethods;

    string s1 = "hello";
    string s2 = "hello world";
    string s4;
    string s5 = "GOODBYE";
    string s6 = "123_456_CC";
    string s7 = "fff_000_bb";
    byte x;
    integer s2len, s3len, i1, i2;

initial
begin
    #15;
    s2len = s2.len( );
    $display("String Length s2 = %0d",s2len);

    s1.putc(0,"C"); //replace 0'th character with 'C'
    $display("String s1 = %s",s1);

    x = s1.getc(0); //get 0'th character of string s1.
    $display("0'th character of string 'Hello' = %s",x);

    s4 = s2.toupper( ); //convert string 's2' to upper case
    $display("Upper Case of 'hello world' = %s",s4);

    s4 = s5.toLowerCase ( );
    $display("Lower case of 'GOODBYE' = %s",s4);

    i1 = s6.Atoi ( );
    $display("s6.Atoi of string s6 '123_456_CC' = %0d",i1);

```

```

s6.itoa (i1);
$display("s6.itoa = %s",s6);

i2 = s7.atohex ( );
$display("s7.atohex of string s7 'ffff_000_bb' = %h",i2);
end
endmodule

```

*Simulation log:*

```

String Length s2 = 11
String s1 = Cello
0'th character of string 'Cello' = C
Upper Case of 'hello world' = HELLO WORLD
Lower case of 'GOODBYE' = goodbye
s6.atoi of string s6 '123_456_CC' = 123456
s6.itoa = 123456
s7.atohex of string s7 '0xffff_000_bb' = fff000bb
V C S   S i m u l a t i o n   R e p o r t

```

In this example, we see usage of various string-based methods. First, we declare a few “string” variables. In the “initial” block, we first get the length of the string “s2” (string s2 = “hello world”;) which is 11, as shown in the simulation log.

Then we use the “putc” method to replace the 0’th character of string “s1” (s1 = “hello”;) with the character “C”:

```
s1.putc(0,"C");
```

Since s1 string was “hello,” it will now become “Cello” as shown in the simulation log.

Then, we get the 0’th character of the string s1 (which is now “Cello”):

```
x = s1.getc(0);
```

This will give use the character “C” since it is the 0’th character of “Cello.” This is shown in the simulation log.

Then we convert “s2” string (s2 = “hello world”) to all uppercase using the .toupper method and store the result in string “s4”:

```
s4 = s2.toupper();
```

Since s2 = “hello world,” we get “HELLO WORLD” as the resultant string, as shown in the simulation log.

Then we convert “s5” string (s5 = “GOODBYE”) to all lowercase using the .tolower method and store the result in string “s4.” So, s4 = goodbye, as shown in the simulation log.

Next, we use the method “atoi” on string “s6” (s6 = “123\_456\_CC”):

```
i1 = s6.atoi ( );
```

Recall that s6.atoi () interprets the string as ASCII decimal string and returns the integer corresponding to the decimal representation in “str.” So, we get “i1” = 123,456. Note that only the decimal digits are returned. So, underscore ( \_) is ignored between 123\_456. And when it encountered the second underscore followed by non-decimal characters, it simply ignored those and stopped the conversion. Hence, the final result is i1 = 123,456.

We then use the inverse method “itoa” on the result of the previous operation:

```
s6.itoa (i1);
```

Recall that itoa is inverse of atoi (). str.itoa(i) stores the ASCII decimal representation of i into “str.” So, since i1 = 123,456, the result of s6.itoa(i1) is = 123,456. We passed the method the integer “i1” and got the ASCII decimal representation of “i1” into the string s6.

Finally, we use the method “atohex” that interprets the string as hexadecimal:

```
i2 = s7.atohex ( );
```

where s7 = “fff\_000\_bb”. So, “a2hex” will look for the hexadecimal string in the string “s7” and return the value to integer “i2.” Hence, i2 = fff000bb. If there were non-hexadecimal digits in “s7,” you will get different results. Also, you cannot use “z” or “x” as part of the string. So, let us say if you have s7 defined as:

```
s7 = "000_fff_zzz"
```

you will get the following warning (Synopsys – VCS):

Warning-[INV-CHAR-ATOHEX] Invalid input character in atohex  
testbench.sv, 40

X(x) and Z(z) are invalid input characters for atohex routine

We are only supporting strings starting with "0x" as input strings inside atohex.

## 2.12 Event Data Type

This is a powerful feature of the language. It allows you to synchronize among your concurrently executing “always” blocks, tasks, etc. Basically, you define an event type and then use it to trigger other processes. It allows for communication between and synchronization of two or more concurrently active processes.

The identifier declared as an event data type is called *named event*.

You can wait for a named event to be triggered using the edge-sensitive @ operator or the level-sensitive “wait” construct. Named events are triggered using the “->” operator. Named events triggered via the -> operator unblock all processes currently waiting on that event. When triggered they behave like a one shot. Here is the syntax:

```
event_trigger ::= -> hierarchical_event_identifier
```

Let us look at an example:

```
module et;
    event etrig;

    initial begin
        #10;
        -> etrig; //trigger named event 'etrig'
        #10;
        -> etrig; //trigger named event 'etrig'
    end

    always @(etrig) //execute when event etrig is triggered
        //edge sensitive
        $display("@ etrig occurred at time %0t",$time);

    initial begin
        wait (etrig.triggered) ; //level sensitive 'wait' on the
                                //event 'etrig'
        $display("'wait' etrig occurred at time %0t",$time);
    end
endmodule
```

#### *Simulation log:*

```
@ etrig occurred at time 10
'wait' etrig occurred at time 10
@ etrig occurred at time 20
```

#### V C S   S i m u l a t i o n   R e p o r t

In the module “et,” we declare a named event called “etrig.” Note that the data type is called *event*. This is an event that can be explicitly triggered whenever you want. In this example, we trigger it first at time 10 and then at time 20. The “always” block is simply waiting for the trigger on “etrig” and executes whenever “etrig” is triggered. This is like waiting for “posedge” or “negedge,” but here we have explicit control over when the “always” block is triggered. The simulation log shows that the “always” block triggered twice since we triggered the named event “etrig” twice.

The module also shows how level-sensitive “wait” can be triggered with the named event “etrig.” Using this mechanism, an event trigger will unblock the waiting process whether the wait executes before or at the same simulation time as the trigger operation. Note that in order to trigger a level-sensitive “wait” block, you must use the built-in method *triggered* as shown below:

```
wait (eventName.triggered) ;
```

The initial block waits for the “wait” block to trigger and displays that it has been triggered. The first event “etrig” occurs at time 10, and hence we get the display from the “wait” block at time 10. The second event “etrig” will not fire the “wait” again, because “wait” is level sensitive, and it was over with the first trigger of “etrig.”

Here is another example, showing that you can have trigger multiple named events concurrently. This allows for good control when you want to fire multiple concurrently running processes:

```
module et;
    event e1, e2, e3;

    initial begin
        #10;
        fork
            -> e1;
            -> e2;
            -> e3;
        join
    end

    always @(e1)
        $display("event e1 triggered at time %0t",$time);

    always @(e2)
        $display("event e2 triggered at time %0t",$time);

    always @(e3)
        $display("event e3 triggered at time %0t",$time);
endmodule
```

#### *Simulation log:*

event e1 triggered at time 10  
 event e2 triggered at time 10  
 event e3 triggered at time 10

Note that once you trigger the event (e.g., `- > e1`), the control passes on to the next executable statement. In our case, the control directly passes from `- > e1` to `- > e2` to `- > e3`.

Three events were triggered simultaneously as you see from the simulation log.

### 2.12.1 Event Sequencing: `wait_order()`

You can wait for named events to execute in a certain order specified by `wait_order` construct. `wait_order( )` suspends the calling process until all of the specified events are triggered in a specified order.

For example, `wait_order(eveA, eveB, eveC);` will execute only if the event `eveA` occurs first and then the event `eveB` and then the event `eveC`.

Here is an example:

```
module et;
    event etrig1, etrig2;

    initial begin
        #10;
        -> etrig1; //trigger named event 'etrig1'
        #10;
        -> etrig2; //trigger named event 'etrig2'
        #10;
        -> etrig1; //trigger named event 'etrig1'
        #100;
    end

    initial begin
        //wait_order (etrig2 , etrig1); //FATAL ERROR
        // $display("'wait' etrig2, etrig1 occurred at time
        %0t",$time);

        wait_order (etrig1 , etrig2); //OK
        $display("'wait' etrig1, etrig2 occurred at time
        %0t",$time);

        wait_order (etrig2 , etrig1); //OK
        $display("'wait' etrig2, etrig1 occurred at time
        %0t",$time);
    end
endmodule
```

*Simulation log:*

```
'wait' etrig1, etrig2 occurred at time 20
'wait' etrig2, etrig1 occurred at time 30
```

**V C S   S i m u l a t i o n   R e p o r t**

In this example, we trigger etrig1 at time 10, followed by etrig2 at time 20, and then etrig1 again at time 30. So, the order of events is etrig1- > etrig2- > etrig1.

In the initial block, we first wait\_order(etrig2, etrig1). But this will give a run time *fatal error* because the *first* order of events was etrig1- > etrig2. We first fire etrig1 at time 10 and then etrig2 at time 20. This is reverse from what wait\_order(etrig2, etrig1) was expecting. Here is the fatal error that Synopsys – VCS gives:

```
Fatal Error: Got an out of order event in wait_order
expecting event "etrig2(0)", received event "etrig1(1)"
```

**V C S   S i m u l a t i o n   R e p o r t**

If we comment that line and continue with wait\_order(etrig1, etrig2), that will work because etrig1 occurred at time 10 and then etrig2 at time 20. And you will get the display ““wait’ etrig1, etrig2 occurred at time 20” as shown in the simulation log.

And the last statement “wait\_order(etrig2, etrig1) will also now work because there is indeed a *second* order of events from etrig2 -> etrig1 at times 20 and 30. The first order was etrig1 -> etrig2. And the second order is etrig2 -> etrig1. Hence, no fatal for the second etrig2 -> etrig1.

## 2.13 Static Casting

Verilog was a loosely typed language and type compatibility was not checked. That is not the case with SystemVerilog. SystemVerilog has complex data types, and you need to use static cast to convert one data type to another data type (even though this is not necessarily the case when two variables are assignment compatible. For example, you can assign a variable of type *bit[8:0]* to one whose type is *logic[31:0]* or, *int* can be interchanged with *bit signed [31:0]* wherever it is syntactically legal to do so. You can use static casts to show intent or change the default implicit casting rules.

During value or variable assignment to a variable, it is required to assign value or variable of the same data type. Casting converts one data type to another compatible data type (e.g., *real* to *int*). Note that only certain types need casting, and not all types can be statically cast (we will discuss where casting needs to be used and where it cannot be used, in a later section).

Static casting is checked during compilation and not during run time. The static casting operator is apostrophe ('). A data type can be changed by using a cast (' ) operation. The expression to be cast is enclosed in parentheses that are prefixed with the casting type and an apostrophe. Here is the syntax:

```
constant_cast :: casting_type ' (constant_expression)
cast :: casting_type ' (expression)
```

For example:

```
real R;
int I;
R = 1.1 * 2.2;
I = int ' (R);
```

Without casting, assignment of different data types results in compile time errors. Here is an example of static casting:

```
module Cast;

    real rNum;
    int iNum;
    string s1;
    logic [7:0] A1;
    logic signed [7:0] A2;

    initial begin

        rNum = (2.3 * 3.2);

        //real to int casting
        $display("\n real to int casting");

        iNum = int'(rNum); // or iNum = int'(2.3 * 3.2);
                           //lose precision
        $display("real value is %f", rNum);
        $display("int value is %0d",iNum);

        $display("\n Signed casting");
        iNum = -10;
        rNum = signed'(iNum);
        $display("real value is %f", rNum);
```

```

$display("int value is %0d", iNum);

$display("\n Unsigned casting");
iNum = -10;
rNum = unsigned'(iNum);
$display("real value is %f", rNum);
$display("int value is %0d", iNum);

A1 [0 +: 8] = unsigned'(-4); //A1 = A1[7:0]
A2 = signed'(4'b1100);
$display("\n A1 = %b A2 = %0d", A1, A2);

//String to int casting
$display("\n String to int casting");
iNum = int'(s1);
$display("String s1 = %0p int iNum=%0d", s1, iNum);

end
endmodule

```

*Simulation log:*

real to int casting  
 real value is 7.360000  
 int value is 7

Signed casting  
 real value is -10.000000  
 int value is -10

Unsigned casting  
 real value is 4294967286.000000  
 int value is -10  
 A1 = 11111100 A2 = -4

String to int casting  
 String s1 = "hello" int iNum=1701604463  
 V C S S i m u l a t i o n R e p o r t

The example declares three variables of type real, int, and string. The first casting is done when we want to assign a real variable (rNum) to an int variable (iNum). They are of different types and hence casting is required. That is done using “iNum = int'(rNum).” The simulation log shows the rNum (real) value and the int value iNum:

real to int casting  
 real value is 7.360000  
 int value is 7

We then do a signed casting:

```
iNum = -10;
rNum = signed'(iNum);
```

iNum is signed and rNum is not signed. And we get the following simulation log:

Signed casting  
real value is -10.000000  
int value is -10

We then do unsigned casting:

```
iNum = -10;
rNum = unsigned'(iNum);
```

iNum is negative. Its unsigned counterpart is assigned to rNum. Hence, we get the following result:

Unsigned casting  
real value is 4294967286.000000  
int value is -10

Then, we cast “A1” and “A2” as follows:

```
A1 [0 +: 8] = unsigned'(-4); //A1 == A1[7:0]
A2 = signed'(4'b1100);
$display("\n A1 = %b A2 = %0d", A1, A2);
```

“A1” gets the unsigned casting value of a signed number. And “A2” get the value of casting an unsigned number to signed. So, unsigned –4 is equal to 11,111,100 and signed 1100 is –4:

A1 = 11111100 A2 = -4

Then, we do string to int casting:

```
iNum = int'(s1);
```

The result is as follows:

String to int casting  
String s1 = "hello" int iNum=1701604463

Note that the \$signed() and \$unsigned() system functions return the same results as signed'() and unsigned'(), respectively. The functions \$itor, \$rtoi, \$bitstoreal, and \$realtobits can also be used to perform type conversions. See Sect. 2.2.1.

When a **shortreal** is converted to an **int** or to 32-bits using either casting or assignment, its value is rounded. Therefore, the conversion can lose information. To convert a **shortreal** to its underlying bit representation without a loss of information, use \$shortrealtobits. To convert from the bit representation of a shortreal value into a **shortreal**, use \$bitstoshortreal.

Also, there is the constant cast (const'(x)). When casting an expression as a constant, the type of the expression to be cast will pass through unchanged. The only effect is to treat the value as though it had been used to define a “const” variable of the type of the expression.

So, where should you be using static casts?

There are basically three places where you should be using static cast:

- (1) You can use static cast between types where implicit casts are already defined, for example, between real and integers. Take the following:

```
real rNum;
int iNum;
byte Abyte
rNum = 2.3;
Abyte = 'b 00000110;

iNum = Abyte/rNum;
$display("rNum = %f iNum = %0d",rNum, iNum);
```

*Simulation log:*

rNum = 2.300000 iNum = 3

Abyte is implicitly cast to a real before doing real division (a single real type in an expression means all operands get converted to real); then the result is implicitly cast back to an int, with rounding to an integral value.

But you could also write this as:

```
iNum = int'(real'(Abyte)/rNum);
```

This is now evaluated as integral division which truncates instead of rounds. The explicit cast to **int** is just documenting what implicitly happens, and it shows your intention that you know what you are doing. Here is the code and simulation log:

```
iNum = int'(real'(Abyte)/rNum);
$display("rNum = %f iNum = %0d",rNum, iNum);
```

*Simulation log:*

```
rNum = 2.300000 iNum = 3
```

- (2) You must use a static cast to assign to an enum type from any other integral type.
- (3) Bit-stream casting (discussed next). This is between unpacked types made up of exclusively integral types and strings, as long as the total number of bits between source and destination can be the same.

And here is where static casting is not allowed:

- Class (there are a few exceptions, but it is not recommended to use a class with a static cast).
- Chandle
- Event
- Virtual interfaces
- Unpacked types containing nonintegral types

### **2.13.1 Bit-Stream Casting**

Bit-stream casting temporarily converts an unpacked array to a stream of bits in vector form. This is helpful when you want to assign packed arrays to unpacked arrays, unpacked arrays to packed arrays, structure to a packed or unpacked array, a fixed or dynamically sized array to a dynamically sized array, and a structure to another structure with different layout or copying arrays and structures. Bit-stream casting can be used to convert between different aggregate types, such as two structure types, or a structure and an array or queue type. An example of where this would be useful is when you want to model packet data transmission over serial communication streams.

During bit-streaming the identity of separate elements of an array is lost, and the temporary vector is represented simply as a stream of bits. This temporary vector can then be assigned to another array, which can be either a packed array or an unpacked array. *The total number of bits represented by the source and destination arrays must be the same* (but the size of each element in the two arrays can be different).

Bit-stream casting uses the static cast operator. For example, (Sutherland, SystemVerilog for Design, Second Edition):

```
typedef int data_t [3:0][7:0]; //Total bits (4*8) = 32
data_t a;
int b [1:0][3:0][3:0]; //Total bits (2*4*4) = 32
a = data_t'(b);
```

In this example, we are assigning an unpacked array to another unpacked array of a different layout. Note that the total number of bits is the same for both arrays. The cast operation is performed by converting the source array into a temporary

vector representation (a stream of bits) and then assigning groups of bits to each element of the destination array. The assignment is made from left to right, such that the left-most bits of the source bit-stream are assigned to the first element of the destination array, the next left-most bits to the second element, and so forth.

## 2.14 Dynamic Casting

We learned about static casting in Sect. 2.13. As we discussed, SystemVerilog casting means the conversion of one data type to another data type. During value or variable assignment to a variable, it is required to assign value or variable of the same data type. Static casting is done using an apostrophe ('') syntax.

There is also the concept of dynamic casting. SystemVerilog provides the \$cast system task/function to assign values to variables that may not be valid because of different data types. When values need to be assigned between two different data-type variables, ordinary assignment might not be valid, and instead a system task called \$cast should be used.

Dynamic casting is most used when resolving handles to classes in inheritance. A base class variable can be used to refer to a derived class object but not vice versa. We will discuss this further in Sect. 8.11.

Dynamic casting is checked during run time. \$cast can be called either as a function or as a task. I prefer \$cast as a function because you can check to see if the \$cast succeeded or not. If it does not succeed, it returns a 0, and you can take some action on that failure. But when \$cast as a task fails, it is a runtime error, and your simulation will simply abort.

Here is the syntax:

```
function int $cast (target_var, source_exp);
task $cast (target_var, source_exp);
```

Here, target\_var is the target variable, and source\_exp is the source expression that should be evaluated and assigned to the target variable. When called as a function, it returns a 1 if the cast is legal. But if the \$cast function fails and returns a 0, it does not make the assignment (and no run-time error). When called as a task, it will attempt to assign the source expression to the target variable, and if it is invalid, a run-time error will occur, and the target variable will remain unchanged.

First, a simple example shows type incompatibility and how \$cast resolves it:

```
module tb;
    typedef enum { soccer=2, cricket=4, football=10 } sports;
    sports    mS;
    int i;

    initial begin
        i = 10;
```

```

mS = i; //Synopsys-VCS - WARNING - incompatible types
        //Mentor Questa/Aldec-Riviera - run time ERROR

$cast(mS, i); // $cast as a task - match types
$display ("Sports=%s", mS.name( ));

i = mS; //No Warning or Error
$display("i=%0d", i);

i=50;
// $cast (mS, i); //ERROR - 50 is not a valid value for enum

if ($cast (mS, i)) // $cast as a function
    $display ("Cast passed");
else
    $display ("Cast failed");
end
endmodule

```

*Simulation log:*

Sports=football

i=10

Cast failed

### V C S   S i m u l a t i o n   R e p o r t

In the module “tb,” we define “int i” and define an enum “sports” and declare “mS” of type “sports.” The first thing we do is assign an int to the enum-type “mS”:

```
mS = i;
```

But init “i” is not of type enum and there is type incompatibility. We will get the following warning from Synopsys – VCS:

```
Warning-[ENUMASSIGN] Illegal assignment to enum variable
testbench.sv, 11
tb, "mS = i;"
```

Only expressions of the enum type can be assigned to an enum variable.

The type int is incompatible with the enum 'sports'

For the same code, Aldec-Riviera will give a run-time error:

ERROR VCP2694 "Assignment to enum variable from expression of different type."

And the following is an error from Mentor’s Questa:

```
** Error (suppressible): testbench.sv(9): (vlog-8386) An enum variable 'mS' of type
'sports' may only be assigned the same enum typed variable or one of its values.
Variable i requires an explicit cast.
```

So, to correct this type incompatibility, we use \$cast to cast “i” to enum “mS”:

```
$cast(mS, i);
```

This dynamic cast will make the int “i” and enum “mS” type compatible and the simulation will pass. Since  $i = 10$ , mS now gets the value 10 and “football” is at value 10 in enum mS. So, simulation log shows “sports = football.”

However, note that the following assignment is perfectly ok:

```
i = mS;
```

An enum can be assigned to an int but not vice versa.

Then we assign an incorrect enum value in the dynamic cast. Since the value  $i = 50$  is outside, the range of enum “sports,” we will get a run time error:

```
i=50;  
$cast(mS, i);
```

Here is the error that we will get from Synopsys – VCS:

```
Error-[STASKE_DCF] Dynamic cast failed  
testbench.sv, 22
```

Dynamic cast using '\$cast' failed. The source expression is not yielding a valid value for the destination variable.

Lastly, we use \$cast as a function. When used as a function, it will return a “1” if the cast succeeds or a “0” if it fails. In our case, we are purposely assigning  $i = 50$  (a value outside the range of enum “sports”). This will cause the \$cast function to fail, and we will get the failure message (“Cast failed”) as shown in the simulation log.

# Chapter 3

## Arrays



**Introduction** This chapter introduces arrays offered by the language. Specifically, packed, unpacked, associative, and dynamic arrays are discussed. Array assignment, indexing, slicing, array manipulation methods, and array ordering methods are discussed.

An array is a collection of variables, all of the same type, and accessed using the same name plus one or more indices. For example:

```
bit uP [3:0]; //1-D unpacked
bit [3:0] p; //1-D packed
logic [31:0] v1 [7:0]; //1-D packed & 1-D unpacked
integer da [ ]; //dynamic array 'da' of type integer
logic myArray[ integer ]; //Associative array
```

Let us discuss each in detail. Array assignment, indexing, slicing, array manipulation methods, and array ordering methods are also discussed.

### 3.1 Packed and Unpacked Arrays

The main difference between an unpacked array and a packed array is that an unpacked array is *not* guaranteed to be represented as a contiguous set of bits, while a packed array is guaranteed to be represented as a contiguous set of bits. Other way to look at the difference is that when a packed array appears as a primary, it is treated as a single vector.

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_3](https://doi.org/10.1007/978-3-030-71319-5_3)) contains supplementary material, which is available to authorized users.

The term *unpacked array* is used to refer to the dimensions declared *after* the data identifier name. An unpacked array can be formed from any data type, including other packed or unpacked arrays.

The term *packed array* is used to refer to the dimensions declared *before* the data identifier name. In other words, a packed array is a mechanism for subdividing a vector into subfields, which can be conveniently accessed as array elements. Note that the packed arrays can be made of only the single bit data types (bit, logic, reg), enumerated types, and recursively other packed arrays and packed structures.

Here is an example of an unpacked array:

```
bit uP [3:0]; //1-D unpacked
//unpacked dimensions declared after the data identified name.
```

This unpacked array can be represented as shown below (compiler dependent):

unused	uP0
unused	uP1
unused	uP2
unused	uP3

As you notice, up0 through up3 are dispersed over multiple words. In other words, they are not contiguous or packed.

Here is an example of a packed array:

```
bit [3:0] p; //1-D packed
//packed dimensions declared before the data identifier name
```

This packed array can be represented as shown below:

unused	P3	P2	P1	P0
--------	----	----	----	----

As you notice, p3 through p0 are contiguous, i.e., packed.

We will study many different dimensions of packed and unpacked arrays. A 1-D packed array simply acts as a vector (as shown above) and is something we are familiar with. Let us start with a 2-D packed array.

### 3.1.1 2-D Packed Array

Let us look at a working example:

```
module tb;
// 2-D packed array
```

```

// 4 entries(rows) of 8 bits(columns) each
// Total packed dimension (contiguous bits) = 4*8 = 32 bits
bit [3:0][7:0]    m_data;

initial begin
  m_data = 32'h0102_0304;//Assign to 32 contiguous bits

  //display 2-d packed array as a contiguous set of bits
  $display ("m_data = 0x%h", m_data);

  //display 1 byte each stored at m_data[0]...m_data[3]
  for (int i = 0; i < 4; i++) begin
    $display ("m_data[%0d] = 0x%h", i, m_data[i]);
  end
end
endmodule

```

*Simulation log:*

```

m_data = 0x01020304
m_data[0] = 0x04
m_data[1] = 0x03
m_data[2] = 0x02
m_data[3] = 0x01

```

**V C S   S i m u l a t i o n   R e p o r t**

In this example, we declare a 2-D packed array called “m\_data.” Note that the dimensions are on the left side of the array name. The way to read this array is that it has four locations/entries (rows) of 8-bits (columns) each row. The [7:0] represents a byte and [3:0] represents four locations where these bytes are stored. And since this is packed array, the entire array is treated as a single contiguous set of bits. Since there are four entries (8-bits each), the total dimension is  $4*8 = 32$  bits. Hence, we can assign the entire array with a single value (32'h 0102\_0304). When we display “m\_data,” it comes out as a contiguous set of bits as shown in the simulation log. We then display each of the four locations and see 8-bits stored at each of these locations, in the simulation log.

### 3.1.2 3-D Packed Array

Let us now look at a 3-D packed array. The idea is the same as that of 2-D packed array. Let us look at an example:

```

module tb;
  bit [2:0][1:0][7:0]    m_data;    // 3-D packed array

```

```

initial begin
    // Assign 16-bits ([1:0][7:0]) at each of the three
    //([2:0])locations
    m_data[0] = 16'h0102;
    m_data[1] = 16'h0304;
    m_data[2] = 16'h0506;

    // m_data as a single packed value
    $display ("m_data = 0x%h", m_data);

    //Assign the entire array with a single value
    m_data = 48'hcafe_face_0708;

    // m_data as a single packed value
    $display("m_data = 0x%h", m_data);

    foreach (m_data[i]) begin
        $display ("m_data[%0d] = 0x%h", i, m_data[i]);
        foreach (m_data[, j]) begin
            $display ("m_data[%0d][%0d] = 0x%h", i, j, m_
data[i][j]);
        end
    end
    end
endmodule

```

*Simulation log:*

```

m_data = 0x050603040102
m_data = 0xcafeface0708
m_data[2] = 0xcafe
m_data[2][1] = 0xca
m_data[2][0] = 0xfe
m_data[1] = 0xface
m_data[1][1] = 0xfa
m_data[1][0] = 0xce
m_data[0] = 0x0708
m_data[0][1] = 0x07
m_data[0][0] = 0x08

```

V C S   S i m u l a t i o n   R e p o r t

In this example, we declare a three-dimensional packed array, named “m\_data.” The way to read the dimensions [2:0][1:0][7:0] is that there are 48 contiguous bits ( $3 \times 2 \times 8$ ). First dimension [2:0] represents three locations at each of which a 16-bit value is stored. Second dimension [1:0] represents two locations at each of which an 8-bit value is stored. Since this is packed array, we can assign a contiguous (as in

vector) value to the array, or we can assign a 16-bit value to each of the three locations. Either way, the value is stored as a contiguous set of bits, as shown in the display of “m\_data” in the simulation log. We then display each of the location as shown in the simulation log.

### 3.1.3 1-D Packed and 1-D Unpacked Array

Let us look at the following example which shows how a 1-D packed and 1-D unpacked array works:

```
module PU;
    logic [31:0] v1 [7:0]; //1-D packed & 1-D unpacked
(memory)

    initial begin
        //Array Index 7 of unpacked
        v1[7] = 'h FF_FF_FF_FF; //equivalent to v1[7][31:0]
        $display(v1);

        //Array Index 6 of unpacked; 31:0 of packed
        v1[6][31:0] = 'h 11_11_11_11;
        $display(v1);

        //Array Index 5 of unpacked; 15:0 of packed
        v1[5][15:0] = 'h aa_aa;
        $display(v1);

        //Array Index 4 of unpacked; 0th bit of packed
        v1[4][0] = 1;
        $display(v1);
    end
endmodule
```

Here is the *simulation log*. I will describe how the code works along with the simulation log:

```
'{ 'h ffffffff, 'hxxxxxxxx, 'hxxxxxxxx, 'hxxxxxxxx, 'hxxxxxxxx, 'h
     xxxxxxxxx, 'hxxxxxxxx }
'{ 'h ffffffff, 'h 11111111, 'hxxxxxxxx, 'hxxxxxxxx, 'hxxxxxxxx, 'h
     xxxxxxxx, 'hxxxxxxxx }
'{ 'h ffffffff, 'h 11111111, 'h xxxxaaaa, 'hxxxxxxxx, 'hxxxxxxxx, 'h
     xxxxxxxx, 'hxxxxxxxx }
```

```
'{ 'h ffffffff, 'h 11111111, 'h xxxxaaaa, 'h xxxxxxxx1, 'h xxxxxxxx, 'h xxxxxxxx, 'h
xxxxxxxx, 'h xxxxxxxx }
```

## V C S S i m u l a t i o n R e p o r t

Here is what is going on. We declare a 1-D packed/1-D unpacked array named “v1.” It is both packed and unpacked in one dimension and acts like a typical memory. There are 32 contiguous (packed) bits that form a word and 8 unpacked slots each of which holds a 32-bit variable. The packed bits act like a vector, and its part selects can be accessed. You cannot do that with the unpacked dimension. Think of this as an 8-deep memory of 32-bit words. 32-bit words are stored at address #0 to address #7. We will see plenty more examples of packed and unpacked arrays in coming sections.

First, we assign v1[7] = ‘h FF\_FF\_FF\_FF;. This means 32-bit word ‘h FF\_FF\_FF\_FF is assigned to array location 7 in “v1.” As you see in the simulation log, the seventh element (since it’s declared [7:0]) of the array is ‘h FF\_FF\_FF\_FF. The rest of the bits are unknown (“x”) since the array is of type “logic” and uninitialized bits of “logic” type remain unknown.

Next, we do the same thing, except that we explicitly assign the 32-bits in array location 6: v1[6][31:0] = ‘h 11\_11\_11\_11;. The simulation log shows this value in the sixth position of the array.

Next, we assign a part-select of the 32-bit packed word and put it in unpacked array location 5: v1[5][15:0] = ‘h aa\_aa;. The simulation log shows the value ‘h xxxxaaaa in position number 5 of the unpacked dimension of the array.

Lastly, we assign “1” to a single bit of the 32-bit words at location 4 of the array. The simulation log shows the single bit assignment at position 4 – ‘hxxxxxx1.

### 3.1.3.1 Multidimensional Arrays

Array in this example is a multi-dimensional array. Such arrays are declared by including multiple dimensions in a single declaration. The dimension preceding the identifier set the packed dimensions, and the dimensions following the identifier set the unpacked dimensions. In a multi-dimensional array, the dimensions declared before the name ([31:0] in our example) vary more rapidly than the dimensions following the name ([7:0] in our example). In other words, a packed dimension varies more rapidly than an unpacked dimension. In a list of dimensions, the rightmost varies most rapidly, but first the packed dimensions vary more rapidly than the unpacked one. For example:

```
bit [1:5][1:6] uP [1:7] [1:8]
```

Here, 1:6 varies most rapidly, followed by 1:5, then 1:8 and 1:7.

As noted above, the 1-D packed and 1-D unpacked array acts like a memory where we store words at different address locations. Here is an example to look at this array as a memory:

```

int data;
logic [7:0] ram [0:1023]; //declare a memory of 1024
locations
                                         //which store bytes
ram [10] = 8'h ff; //store a byte at address location 10
data = ram [10]; //read from address location 10

```

Now let us look at arrays with different dimensions and see how their locations can be accessed.

### 3.1.4 4-D Unpacked Array

This representation is derived from Cummings (Sunburst Design, n.d.). We declare a 4-D unpacked array as follows. Note that all the dimensions of the declaration are on the right side of the variable “uP”:

```
logic uP [3:0][2:0][1:0][7:0];
```

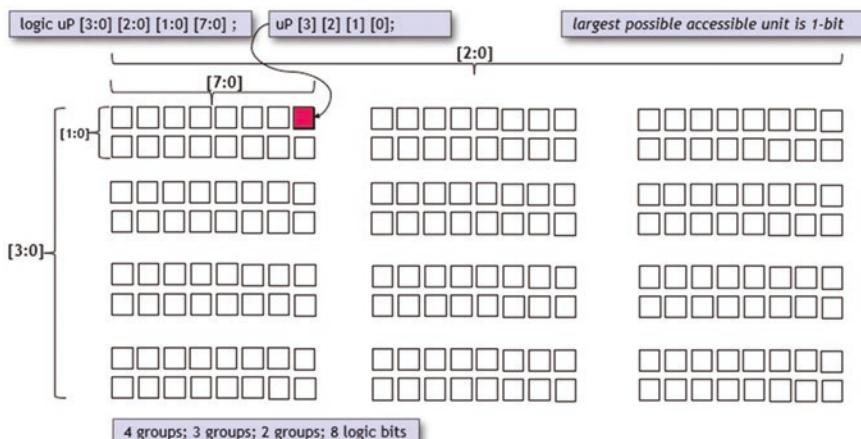
Figure 3.1 shows the graphical representation of this array.

The entire array is unpacked. The order in which array dimensions are read is:

1            2            3            4

```
logic uP [3:0][2:0][1:0][7:0];
```

One way to understand the graphical representation is to break down the dimensions as follows:



**Fig. 3.1** 4-D unpacked array

- [3:0] – Unpacked vertical dimension – *row*
- [2:0] – Unpacked horizontal dimension – *column*
- [1:0] – Unpacked vertical dimension – *row*
- [7:0] – Unpacked final horizontal dimension – *column*

Note that the final dimension [7:0] is also unpacked, i.e., they are not contiguous bits. In other words, only a single bit will be accessible as the largest possible accessible unit. We will see how this contrasts when the final dimension [7:0] is packed (Sect. 3.1.5).

The example shows which position of the array will be accessed by  $\text{uP}[3][2][1][0]$ . We first select row #3 of the unpacked dimension [3:0], then column #2 of [2:0], then row #1 of [1:0], and finally the 0th bit of [7:0]. The final bit accessed is shown in red.

Study this section carefully to see how such multi-dimensional arrays work.

### 3.1.5 1-D Packed and 3-D Unpacked Array

We now see what happens when one of the dimensions is packed, while others are unpacked. We take a similar example to that in Sect. 3.1.4 but convert the final dimension [7:0] from unpacked to packed. Here is the array definition, as an example:

```
logic [7:0] uP [3:0][2:0][1:0];
```

Figure 3.2 shows the graphical representation.

The graphical representation is similar to that shown in Fig. 3.1, except that the final dimension [7:0] is packed which means they are stored as contiguous bits. In other words, the largest possible accessible unit is 8-bits wide. This does not mean that you can only access [7:0] as a unit. You can also reach a single bit or a part-select of this 8-bit-wide contiguous set of bits. But you cannot access more than 8-bits as a unit in this array.

The order in which array dimensions are read is:

4	1	2	3
---	---	---	---

```
logic [7:0] uP [3:0][2:0][1:0];
```

The example shows which bit will be accessed by:

```
uP[0][0][0][0]
```

We follow row- > column- > row- > column analogy as described in the previous section.

Note that we always go from unpacked dimension to packed dimension.

So, select row 0 of the [3:0] dimension. Then select column 0 of [2:0]. Then select row 0 of [1:0]. And finally select bit 0 of [7:0]. This is shown as the red bit in the lower right side of the diagram.

Now, let us see what gets accessed by:

```
uP[3][1][1]
```

Again, start with row - > column transitions. First select row 3 of [3:0], then column 1 of [2:0], and then row 1 of [1:0]. Now, we have not specified any bits or part-select of the final dimension [7:0] which means we access the entire byte since that is the largest accessible unit. This is shown as the contiguous red bits in the upper middle of the diagram.

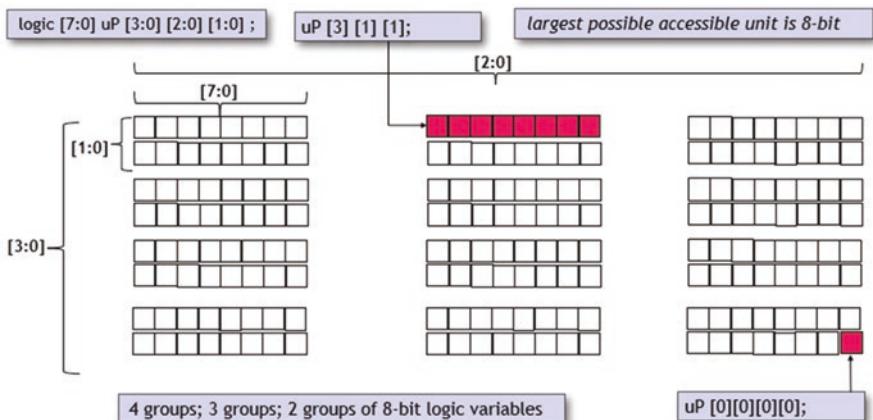
### 3.1.6 2-D Packed and 2D-Unpacked Array

Let us now see how the data is arranged and accessible with a 2-D packed and 2-D unpacked array. Let us take the following dimensions as an example:

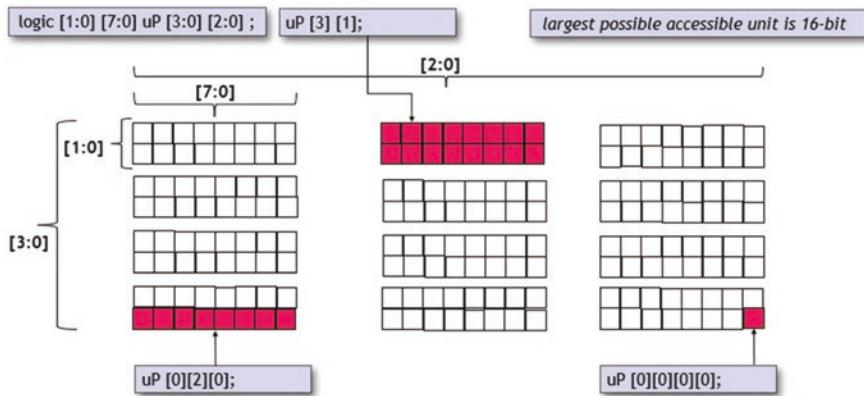
```
3      4      1      2
```

```
logic [1:0] [7:0] uP[3:0] [2:0];
```

The graphical representation of this array is shown in Fig. 3.3.



**Fig. 3.2** 1-D packed and 3-D unpacked array



**Fig. 3.3** 2-D packed and 2-D unpacked array

Note that `[1:0] [7:0]` are on the left-hand side of the variable declaration. This means that they are packed contiguous bits. `[1:0]` means two rows and `[7:0]` means eight columns. Since they are packed, these  $2 \times 8 = 16$ -bits will act as the largest accessible unit. Again, single largest accessible unit does not mean that you can access only 16-bits at a time. You can indeed select a part-select or a bit of this 16-bit unit. But you cannot access more than 16-bits as a unit in this array.

Let us take some examples to see how bits or part-selects can be accessed in this array:

`uP [3] [1]`

Again, going with row and column analogy (unpacked to packed), we first access row 3 of the unpacked side of the array (`[3:0]`) and then column 1 of `[2:0]`. We have not specified what needs to be accessed from the packed side of the array (viz., `[1:0] [7:0]`). This means that we will be accessing the entire packed contiguous set of 16-bits. This is shown in the upper left corner (red bits) of Fig. 3.3.

Another example:

`uP [0] [2] [0]`

We start with row 0 of unpacked side of the array (`[3:0]`) and then column 2 of `[2:0]`. So, we are done with the unpacked part of the array. The last index is `[0]` which points to the packed side of the array (we always go from unpacked to packed dimensions). So, row 0 of `[1:0]` is accessed. But we have not specified the last part of the packed dimension, which means the entire byte (`[7:0]`) will be accessed. This is shown in the lower left of the figure (red bits).

One more example:

`uP [0] [0] [0] [0]`

Similar to the above analogy, we start with the unpacked dimension of the array and move to the packed dimension. So, first select row 0 of the unpacked dimension [3:0], then select column 0 of [2:0], then moving on to the packed side, select row 0 of [1:0], and finally select bit 0 of [7:0]. This is shown as a red bit on the lower right side of Fig. 3.3.

### 3.1.7 3-D Packed and 1-D Unpacked Array

Let us now see how the data is arranged and accessible with a 3-D packed and 1-D unpacked array. Let us take the following dimensions as an example:

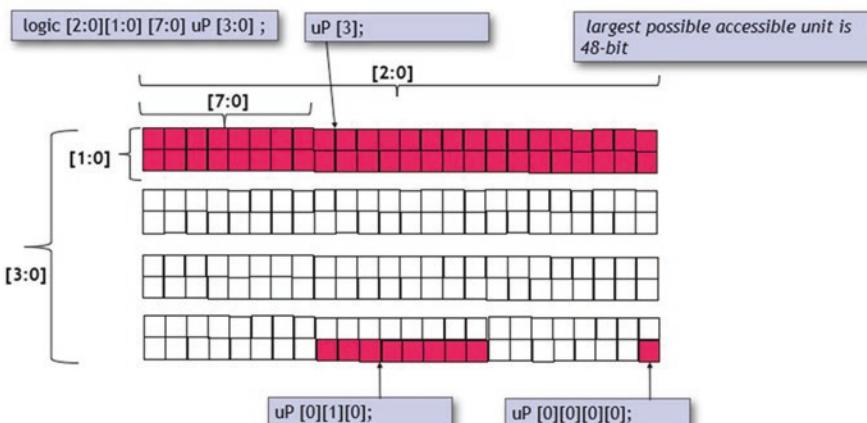
2      3      4      1

```
logic [2:0][1:0][7:0] uP [3:0];
```

The graphical representation of this array is shown in Fig. 3.4.

Note that [2:0] [1:0] [7:0] are on the left-hand side of the variable declaration. This means that they are packed contiguous bits. First the unpacked side of the array [3:0] rows and then on the packed side of the array – [2:0] means three columns, [1:0] means two rows, and [7:0] means eight columns. Since [2:0][1:0][7:0] are packed, these  $3 \times 2 \times 8 = 48$ -bits will act as the largest accessible unit. Again, single largest accessible unit does not mean that you can access only 48-bits at a time. You can indeed select a part-select or a bit of this 48-bit unit. But you cannot access more than 48-bits as a unit in this array.

Let us take some examples to see how bits or part-selects can be accessed in this array:



**Fig. 3.4** 3-D packed and 1-D unpacked array

```
uP [3];
```

Continuing with the analogy of rows and columns, this dimension means to access row 3 of the unpacked dimension [3:0]. Since we have not specified any other dimension on the packed side, the [2:0][1:0][7:0] give us the contiguous 48-bits as a unit to be accessed. This is shown as the 48 contiguous red bits on the upper side of Fig. 3.4.

Another example:

```
uP[0][1][0];
```

Here we first access row 0 of the unpacked side of the array ([3:0]), then column 1 of the packed side ([2:0]), and then row 0 of [1:0] on the packed side. Since the last dimension is not given, we access the [7:0] side of the packed array as a unit of 8-bits. This is shown as red bits in the lower middle of Fig. 3.4.

Finally, let us see how the following gets accessed:

```
uP[0][0][0][0]
```

First we access row 0 of the unpacked array dimension ([3:0]); then on the packed side, column 0 of [2:0]; then row 0 of [1:0]; and finally the bit 0 of [7:0]. This is shown as the singular red bit on the lower right side of Fig. 3.4.

## 3.2 Assigning, Indexing, and Slicing of Arrays

Let us see how the above discussion ties into a real example where we assign to array and sub-arrays; index these arrays and their slices.

Here is an example. The description in the source code and in the simulation log explains what is going on:

```
module PU;
    int A[2:0][3:0][4:0], B[2:0][3:0][4:0], C[5:0][4:0];

initial
begin
    A[0][2][4] = 1024; //row 0, column 2, element #4

    //display index #4 (i.e., 5th element)
    $display("A[0][2][4]=",A[0][2][4]);

    //display 5 elements of row 0, column 2
    $display("A[0][2]=",A[0][2]);
```

```

//display row 0 (4 columns; 5 elements each)
$display("A[0]=",A[0]);

//display 3 rows * 4 columns of 5 elements each
$display("A=",A);

$display("\n");
B[1][1][1]=512; //row 1; column 1; element #1
// assign a subarray composed of five ints
A[2][3] = B[1][1];

//display 5 elements of row 2, column 3
$display("A[2][3]=",A[2][3]);

B[0][0][0]=128; //Assign only to the last unpacked element
A[1] = B[0];
$display("\n");
$display("A[1]=",A[1]); //display row 1 (4 columns; 5
elements each)

C[5][4]=64;
A[0][1] = C[5];
$display("\n");
$display("C[5]=",C[5]);
$display("A[0][1]=",A[0][1]);
end
endmodule

```

*Simulation log:*

```

A[0][2][4]=      1024 //index #4 (i.e., 5th element)
A[0][2]='{1024, 0, 0, 0, 0} //5 elements of row 0, column 2
A[0]=''{0, 0, 0, 0, 0}, '{1024, 0, 0, 0, 0}, '{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 0}
//4 columns of row 0 with value assigned to column 2, element #4 (5th position)

A=''{'{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 0},
  '{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 0}, '{1024, 0, 0,
  0, 0}, '{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 0}}
//Entire 3 rows*4 columns (12 entries – 5 elements each with value assigned to
column 2, element #5)

A[2][3]='{0, 0, 0, 512, 0} // display 5 elements of row 2, column 3

A[1]=''{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 0}, '{0, 0, 0, 0, 128}
// display row 1 (4 columns; 5 elements each)

C[5]='{64, 0, 0, 0, 0} //Row 5, 5 elements with index 4 assigned

```

```
A[0][1]={64, 0, 0, 0, 0} //Row 0, column 1 of 5 elements
V C S S i m u l a t i o n R e p o r t
```

### 3.2.1 *Packed and Unpacked Arrays as Arguments to Subroutines*

Arrays can be passed as arguments to subroutines. As with any arguments when they are passed as values, a copy of the array is passed to the subroutine.

Here is an example of do's and don'ts of the formal of an array and its actual to the array as arguments to subroutines.

Assume the following SystemVerilog task where we define a formal of type 2-D unpacked array:

```
task trial (int a[3:1][3:1]); // 'a' is a two-dimensional array
                           // (2-D unpacked)
```

Following are examples of actuals that are passed to this task. It highlights where you may end up making mistakes:

```
int b[3:1][3:1]; // OK: same type, dimension, and size
int b[1:3][0:2]; // OK: same type, dimension, & size
                  // (different ranges)
logic b[3:1][3:1]; // ERROR: incompatible element type
                    // (logic vs. int)
event b[3:1][3:1]; // ERROR: incompatible type (event
vs. int)
int b[3:1]; // ERROR: incompatible number of dimensions
int b[3:1][4:1]; // ERROR: incompatible size (3 vs. 4)
```

## 3.3 Dynamic Arrays

A dynamic array is an unpacked array whose size can be set or changed at run time. You need to instantiate a dynamic array with the “new” operator. During instantiation, the size of a dynamic array is set. The default size is set by the “new” constructor or array assignment.

Dynamic array dimensions are denoted in the array declaration by [ ]. Syntax is:

```
data_type array_name [ ];
```

Let us look at a simple example:

```

module darray;
    integer da [ ]; //dynamic array 'da' of type integer
initial
begin
    da = new[4]; //construct and allocate a size of 4 elements
    $display($stime,,, "da size = %0d",da.size);

    da.delete( ); //delete elements of an array
    $display($stime,,, "da size = %0d",da.size);

    da = '{1,2,3,4}; //array initialization
    $display($stime,,, "da = ",da);
end
endmodule

```

*Simulation log:*

```

0 da size = 4
0 da size = 0
0 da ='{1, 2, 3, 4}
V C S  Simulation Report

```

The model declares a dynamic array named “da[ ]” of type integer. Then applies different methods to get its size, initialization values, etc.

“da = new [4];” constructs the array “da” and allocates a size of four elements. This is an unpacked array.

Then we display its size. As you see from the simulation log, its size comes out to be 4, as expected. The size( ) built-in method returns the current size of the array.

Then we delete this array to see how its size changes. Does it delete the entire array? Well, the array is still there but its elements are cleared. Hence, its size comes out as 0, as shown in the simulation log. The delete( ) built-in method clears all the elements yielding an empty array (zero size).

Then we initialize the array “da = ‘{1,2,3,4};.” Since the array was sized with four elements, we give it four values, each for each element. When we display the array “da,” we see these elements displayed.

In this example, we saw how to declare a dynamic array, how to construct and allocate a size to it, and how to delete the elements of the entire array. And we saw different methods available.

### 3.3.1 Dynamic Arrays – Resizing

Let us look at an example showing how to add elements and resize a dynamic array:

```

module darray;
    bit [7:0] d_array1[ ];
initial begin
    //memory allocation
    d_array1 = new[2];
$display($stime,,, "d_array1 size = %0d",d_array1.size);

    $display("\n");
    //array assignment - changes the size of the array
    d_array1 = {2,3,4}; //add 1 more element to the array
$display($stime,,, "d_array1 size = %0d",d_array1.size);
    $display($stime,,, "d_array1=",d_array1);

    $display("\n");
    d_array1[2]=5;
    $display($stime,,, "d_array1[0]=",d_array1[0]);
    $display($stime,,, "d_array1[1]=",d_array1[1]);
    $display($stime,,, "d_array1[2]=",d_array1[2]);
    $display($stime,,, "d_array1=",d_array1);

    $display("\n");
    d_array1[3]=6; //will not change the size of the array or
                    //add a new element - Warning
$display($stime,,, "d_array1 size = %0d",d_array1.size);
    $display($stime,,, "d_array1=",d_array1);

    $display("\n");
    d_array1 = {2,3,4,6};
$display($stime,,, "d_array1 size = %0d",d_array1.size);
    $display($stime,,, "d_array1=",d_array1);

    $display("\n");
    //increase the size of d_array1
    d_array1 = new [d_array1.size( ) + 1] (d_array1) ;
$display($stime,,, "d_array1 size = %0d",d_array1.size);
    $display($stime,,, "d_array1=",d_array1);
end
endmodule

```

*Simulation log:*

```

0  d_array1 size = 2
0  d_array1 size = 3
0  d_array1={'h2, 'h3, 'h4}

```

```

0 d_array1[0]= 2
0 d_array1[1]= 3
0 d_array1[2]= 5
0 d_array1={'h2, 'h3, 'h5}

0 d_array1 size = 3
0 d_array1={'h2, 'h3, 'h5}

0 d_array1 size = 4
0 d_array1={'h2, 'h3, 'h4, 'h6}

0 d_array1 size = 5
0 d_array1={'h2, 'h3, 'h4, 'h6, 'h0}

```

V C S   S i m u l a t i o n   R e p o r t

Let us go through the code line by line.

First we declare a dynamic array called d\_array1. Then we allocate two elements to it (0,1) and display the size of the array as shown below:

```

d_array1 = new[2];
$display($stime,,, "d_array1 size = %0d",d_array1.size);

```

0 d\_array1 size = 2

Then initialize the array and add one more element. As we have noted, this is a dynamic array, and its size can be changed dynamically. Hence the size now becomes 3, as shown below:

```

//array assignment - changes the size of the array
d_array1 = {2,3,4}; //add 1 more element to the array
$display($stime,,, "d_array1 size = %0d",d_array1.size);
$display($stime,,, "d_array1=",d_array1);

```

Simulation log displays:

0 d\_array1 size = 3  
0 d\_array1={'h2, 'h3, 'h4}

Then, we change the value of the second element and display all the elements. This is to show how to override a value of the array element:

```

d_array1[2]=5; //Change the value of 2nd element
$display($stime,,, "d_array1[0]=",d_array1[0]);
$display($stime,,, "d_array1[1]=",d_array1[1]);
$display($stime,,, "d_array1[2]=",d_array1[2]);
$display($stime,,, "d_array1=",d_array1);

```

Simulation log shows:

```
0 d_array1[0]= 2
0 d_array1[1]= 3
0 d_array1[2]= 5
0 d_array1={'h2, 'h3, 'h5}
```

Next we do something interesting. The current size of the array is 3 (i.e., index 0,1,2). But we try to add a third element to the array (i.e., at index 3). What will happen? Nothing. The size of the array will not change, and the value 6 assigned to d\_array1[3] will be ignored. Here is the code:

```
d_array1[3]=6; //will not change the size of the array -
               //Warning
$display($stime,,, "d_array1 size = %0d",d_array1.size);
$display($stime,,, "d_array1=",d_array1);
```

As we see from the simulation log, the size does not change and the values of the elements in the array also do not change (some simulators will give an “out of bound” warning). Here is the simulation log:

```
0 d_array1 size = 3
0 d_array1={'h2, 'h3, 'h5}
```

Next, I am showing two ways to change the size of the array. First, simply reinitialize the array with an extra element, as shown in the code below:

```
d_array1 = {2,3,4,6};
$display($stime,,, "d_array1 size = %0d",d_array1.size);
$display($stime,,, "d_array1=",d_array1);
```

So, now the size will become 4 as shown in the log below:

```
0 d_array1 size = 4
0 d_array1={'h2, 'h3, 'h4, 'h6}
```

Another way to resize an array is shown below:

```
d_array1 = new [d_array1.size( ) + 1] (d_array1);
               //increase the size of
d_array1
$display($stime,,, "d_array1 size = %0d",d_array1.size);
$display($stime,,, "d_array1=",d_array1);
```

So, now the array size will become 5 as shown in the log below. Note that here we are not only increasing the size by 1 but also initializing it with d\_array1 (which was d\_array1 = {'h2, 'h3, 'h4, 'h6}). So, the new array will be as shown in the simulation log:

```
0 d_array1 size = 5  
0 d_array1={'h2, 'h3, 'h4, 'h6, 'h0}
```

In this example we saw how to add elements in an array and how to resize it.

### 3.3.2 *Copying of Dynamic Arrays*

You can copy one dynamic array to another dynamic array. This allows for flexibility in reusing an array for creating another array. Here is an example:

```
module darray; //copying of arrays  
int oarray [ ];  
int carray [ ];  
  
initial begin  
    // Allocate 5 memory locations to "oarray" and  
    // initialize with values  
    oarray = new [5];  
    oarray = '{1, 2, 3, 4, 5};  
  
    carray = oarray; // copy "oarray" to "carray"  
    $display ("carray = %p", carray);  
  
    // Grow size by 1 and copy existing elements to the  
    "carray"  
    carray = new [carray.size( ) + 1] (carray);  
    $display("carray size = %0d",carray.size);  
  
    // Assign value 6 to the newly added location [index 5]  
    carray [carray.size( ) - 1] = 6;  
    $display("carray[5]=%0d",carray[5]);  
  
    // Display contents of new "carray"  
    $display ("carray = %p", carray);  
  
    oarray = carray; //copy carray to oarray  
    $display ("oarray = %p", oarray);  
  
    // Display size of both arrays  
    $display ("oarray.size( ) = %0d, carray.size( ) = %0d",  
oarray.size( ), carray.size( ));  
end  
endmodule
```

We declare two dynamic arrays, namely, “oarray” and “carray.” We initialize “oarray” and then copy it into “carray.” Then we increase the size of “carray” and copy it back into “oarray.” I will explain the workings after the simulation log.

*Simulation log:*

```
carray = '{1, 2, 3, 4, 5}
carray size = 5
carray[5]=6
carray = '{1, 2, 3, 4, 5, 6}
oarray = '{1, 2, 3, 4, 5, 6}
oarray.size( ) = 6, carray.size( ) = 6
V C S   S i m u l a t i o n   R e p o r t
```

First, we allocate five elements to “oarray” and initialize it. And then we copy “oarray” to “carray.” Then display the newly created “carray”:

```
carray = '{1, 2, 3, 4, 5}
```

Then, we increase the size of “carray” by 1. The original size of “oarray” was 5. So, now the size of newly created “carray” will be 6:

```
carray size = 6
```

Then we assign a value to this element 6 (i.e., index #5 of “carray”). And display this newly assigned value. And also display the entire “carray”:

```
carray[5]=6
carray = '{1, 2, 3, 4, 5, 6}
```

Now, we copy “carray” back to “oarray” and display “oarray.” It should be the same as “carray”:

```
oarray = '{1, 2, 3, 4, 5, 6}
```

Finally, we display the size of both the original “oarray” and the copied “carray.” They should be the same:

```
oarray.size( ) = 6, carray.size( ) = 6
```

### 3.3.3 Dynamic Array of Arrays

So, far we have array with single dimension (logic array1[ ]). But you can also have arrays with multiple dimensions – a.k.a. array with sub-arrays. We need to understand how to allocate memory and initialize such arrays. Let us see an example of how such arrays work:

```

module darray;
    int abc[ ][ ]; //array of arrays

    initial begin
        abc = new[3]; //sub array still not created
        $display("abc = ",abc);

        //Create sub-arrays
        foreach (abc[i]) begin
            abc[i] = new[4];
            $display("abc[%0d] = %p", i, abc[i]);
        end
        $display("abc = ",abc);

        //assign values to array and sub-array
        foreach(abc[i , j]) begin
            abc[i][j] = (j+1)+i;
        end

        //display
        foreach (abc[i , j]) begin
            $display("abc[%0d][%0d] = %0d", i, j, abc[i][j]);
        end
        $display("abc = ",abc);
    end
endmodule

```

I will explain how the code works after the simulation log.

*Simulation log:*

```

abc ='{}','{}','{}'
abc[0] ='{0, 0, 0, 0}
abc[1] ='{0, 0, 0, 0}
abc[2] ='{0, 0, 0, 0}

abc ='{'0, 0, 0, 0} ,'{0, 0, 0, 0} ,'{0, 0, 0, 0} }

abc[0][0] = 1
abc[0][1] = 2
abc[0][2] = 3
abc[0][3] = 4

abc[1][0] = 2
abc[1][1] = 3
abc[1][2] = 4
abc[1][3] = 5

```

```
abc[2][0] = 3
abc[2][1] = 4
abc[2][2] = 5
abc[2][3] = 6
```

```
abc = '{1, 2, 3, 4} , {2, 3, 4, 5} , {3, 4, 5, 6} }
```

### V C S S i m u l a t i o n R e p o r t

First we declare a two-dimensional dynamic array named “abc.” Then we allocate memory for it to have three elements (`abc = new[3]`). Note that only the first dimension is constructed, the sub-array is still not allocated. So, when we display “abc,” we get the following which shows that the sub-arrays are empty:

```
abc = '{} , {} , {}'
```

Next, for each of the allocated array, we allocate elements for each of the sub-array. This is done in a for loop that goes through array numbers 1, 2, and 3 and allocates four elements for each of these three arrays. So, when we display these sub-arrays, we see the following. Note that the values are “0” because the arrays are of type “int” and “int” is a two-state type with default value “0”:

```
abc[0] = {0, 0, 0, 0}
abc[1] = {0, 0, 0, 0}
abc[2] = {0, 0, 0, 0}
```

Now that we have both the main array and sub-arrays allocated, we simply go through each element of these arrays and assign some value. Once we assign the values, we display each of the elements of these arrays. We get the following. Each array (0 to 3) has four sub-array elements. That is what is displayed here:

```
abc[0][0] = 1
abc[0][1] = 2
abc[0][2] = 3
abc[0][3] = 4

abc[1][0] = 2
abc[1][1] = 3
abc[1][2] = 4
abc[1][3] = 5

abc[2][0] = 3
abc[2][1] = 4
abc[2][2] = 5
abc[2][3] = 6
```

Finally, we display the entire dynamic array “abc.” We get the following:

```
abc = '{1, 2, 3, 4} , {2, 3, 4, 5} , {3, 4, 5, 6} }
```

So, that is a simple example of showing how to create an array with sub-arrays and how to initialize them.

### 3.4 Associative Arrays

As we saw, dynamic arrays are useful at dealing with a collection of variables (contiguous) whose number can change dynamically (resize, allocate new elements, etc.). In contrast, an associative array is kind of a lookup table. The memory is not allocated until it is used, and the data are stored at the so-called indices. You look up data by using an index which acts like a key to that location. This is like a tag memory in a cache subsystem where the cache lines are sparsely located at indices which are known as tags. You store data at a given key (index) and look it up using the same key. This way you can take advantage of limited sparse memory. When the size of the collection of variables is unknown or the data space is sparse, an associative array is a better option over a dynamic array.

Array elements in associative arrays are allocated dynamically. Associative array elements are unpacked. The associative array maintains the entries that have been assigned values and their relative order according to the index data type. “real” or “shortreal” data types, or a type containing a real or shortreal, is an illegal index type.

Here is the syntax:

```
data_type  array_id [index_type];
```

where `data_type` is the data type of the array elements. `array_id` is the name of the array. And `index_type` is the data type used as the index (or key).

Some examples of associative arrays. We will see each one in detail:

```

//wildcard index. Can be indexed by any integral type
int myArray[ * ];

//Array that stores 'bit [31:0]' at string type index.
bit [31:0] myArray[ string ];

//Array that stores 'string' at string type index.
string myArray [ string ];

// Array that stores 'int' at Class type index
int myArray [ class ];

//Array that stores 'logic' type at integer type index
logic myArray[ integer ];

typedef bit signed [7:0] mByte;
int myArray [mByte]; //bit signed' index

```

Here are different types of indices.

### 3.4.1 Wild Card Index

*Important Note: I recommend against using wildcard index. They cannot be used in a foreach loop or with most of the array methods that need a defined index type. This is only in SV left over from Vera which had no other index types.*

Having said that, here it is for the sake of completeness:

```

int array_name[*]; //Wildcard index. can be indexed by any
//integral datatype.

```

The [\*] index is considered a wild card index. It means that the index can be of any integral type (recall that integral data type refers to the data types that can represent a single basic integer data type, packed array, packed structure, packed union, enum variable, or time variable).

There are a few rules to follow for wildcard index:

- Nonintegral index values are illegal and result in an error.
- A four-state index value containing X or Z is invalid.
- Indexing expressions are treated as unsigned.
- Associative arrays that specify a wildcard index type cannot be used in a foreach loop.

An example follows later in the section.

### 3.4.2 String Index

```
int array_name [ string ]; // String index
```

String index uses a string data type as the index to store and retrieve data. Here are some rules for string type index:

- Indices can be strings or string literals of any length.
- An empty string ““index is valid.
- The ordering is lexicographical (lesser to greater).

An example follows in Sect. 3.4.4.

### 3.4.3 Class Index

```
int array_name [className]; //Class index
```

This is an associative array that uses SystemVerilog Class as its index. The indices can be objects of that particular type or derived from that type. A null index is valid.

Here is a simple example:

```
module assoc_arr;
class AB;
    int a;
    int b;
endclass

int arr[AB]; //Associative array 'arr' with class 'AB'
as index
AB obj, obj1;

initial begin
    obj = new();
    obj1= new();

arr[obj]=20; //Store 20 at the object handle index 'obj'
$display("%0d",arr[obj]);

arr[obj1]=10; //Store 10 at the object handle index 'obj1'
$display("%0d",arr[obj1]);
end
endmodule
```

*Simulation log:*

```
20
10
```

### V C S S i m u l a t i o n R e p o r t

We declare class “AB” and use that as the class index in the associative array “arr” (int arr[AB]). We declare two variables of type AB (obj and obj1) and instantiate the class to create object handles obj and obj1. Now, to store a value at the class index, we need to provide it with the object handle of that class to the associative array. We do that with arr[obj] = 20; and arr[obj1] = 10;. This is the way we use a class index to store values in an associative array. The display shows the values stored in the array at object handles.

#### 3.4.4 *String Index – Example*

Here is a simple example to explain storing and retrieving data from an associative array using string index:

```
module assoc_arr;
  integer St [string] = '{"Peter":26, "Paul":24, "Mary":22};
  integer data;

initial
begin
  $display("St=",St);
  $display("data stored at Peter = %0d",St["Peter"]);
  $display("data stored at Paul = %0d",St["Paul"]);
  $display("data stored at Mary = %0d",St["Mary"]);

  St["mike"] = 20;    //new data stored at new index "mike"
  data = St["mike"]; //retrieve data stored at index "mike"
  $display("data stored at mike = %0d",data);

  $display("St=",St);
end
endmodule
```

*Simulation log:*

```
run -all;
```

```
# KERNEL: St={'Mary':22, "Paul":24, "Peter":26}
```

```
# KERNEL: data stored at Peter = 26
```

```
# KERNEL: data stored at Paul = 24
```

```
# KERNEL: data stored at Mary = 22
# KERNEL: data stored at mike = 20
# KERNEL: St='{"Mary":22, "Paul":24, "Peter":26, "mike":20}
# KERNEL: Simulation has finished.
```

Here is how the code works. First, we declare the associative array “St” and initialize it:

```
integer St [string] = '{"Peter":26, "Paul":24, "Mary":22};
```

Initialization means data 26 is stored at index “Peter,” data 24 is stored at index “Paul,” and data 22 is stored at index “Mary.”

Then, we display this array to see what data is stored at each string index. That is what the first line of the simulation log shows. The ordering is lexicographical (lesser to greater).

Then we display the data stored at each of the string index. As we see, correct values of what is stored at each index are displayed in the next three lines of the simulation log.

Then, we add new data at a new index called “mike.” Note that new memory is allocated only when we added data at this new string index. An entry for a nonexistent associative array element is allocated when it is used as the target of an assignment. Array elements in associative arrays are allocated dynamically. When we display “data,” we see that value 20 was stored at index “mike.”

In the end we display the entire array and see that the new data at index “mike” is part of the array.

### 3.4.5 *Associative Array Methods*

Here are some methods afforded by associative arrays. We will see an example, right after Table 3.1 on the usage of these methods.

Here is an example that shows the usage of these methods:

```
module assoc_arr;
int temp, imem[int];
integer St [string] = '{"Peter":20, "Paul":22, "Mary":23};

initial
begin
if(St.exists( "Peter" ) ) $display(" Index Peter exists ");

//Assign data to imem[int]
imem[ 2'd3 ] = 1;
imem[ 16'hffff ] = 2;
```

**Table 3.1** Associative array methods

Method	Description
num( ) and size( )	<b>function int</b> num( ); <b>function int</b> size( ); The num( ) and size( ) methods return the number of entries in the associative array. If the array is empty, they return 0
delete( )	<b>function void</b> delete ( [input index] ); If the index is specified, then the delete( ) method removes the entry at the specified index If the entry to be deleted does not exist, the method issues no warning <i>If the index is not specified, then the delete ( ) method removes all the elements in the array</i>
exists ( )	<b>function int</b> exists( input index ); The exists( ) function checks whether an element exists at the specified index within the given array. It returns 1 if the element exists; otherwise, it returns 0
first ( )	<b>function int</b> first( ref index ); The first( ) method assigns to the given index variable the value of the first (smallest) index in the associative array. It returns 0 if the array is empty; otherwise, it returns 1
last ( )	<b>function int</b> last( ref index ); The last( ) method assigns to the given index variable the value of the last (largest) index in the associative array. It returns 0 if the array is empty; otherwise, it returns 1
next ( )	<b>function int</b> next( ref index ); The next( ) method finds the smallest index whose value is greater than the given index argument If there is a next entry, the index variable is assigned the index of the next entry, and the function returns 1. Otherwise, the index is unchanged, and the function returns 0
prev ( )	<b>function int</b> prev( ref index ); The prev( ) function finds the largest index whose value is smaller than the given index argument. If there is a previous entry, the index variable is assigned the index of the previous entry, and the function returns 1. Otherwise, the index is unchanged, and the function returns 0

```

imem[ 4'b1000 ] = 3;
$display( " imem has %0d entries", imem.num );

if(imem.exists( 4'b1000)) $display("Index 4b'1000
exist");

imem.delete(4'b1000);

if(imem.exists( 4'b1000)) $display("Index 4b'1000
exists");
else $display(" Index 4b'1000 does not exist");

imem[ 4'b1000 ] = 3;

```

```

        if(imem.first(temp)) $display(" First entry is at index
%0d ",temp);
        if(imem.next(temp)) $display(" Next entry is at index
%0b ",temp);
        if(imem.last(temp)) $display(" Last entry is at index
%0h",temp);

        imem.delete( ); //delete all entries
        $display(" imem has %0d entries", imem.num );
        $display(" imem = %p", imem);
    end
endmodule

```

*Simulation log:*

Index Peter exists

imem has 3 entries

Index 4b'1000 exists

Index 4b'1000 does not exist

First entry is at index 3

Next entry is at index 1000

Last entry is at index ffff

imem has 0 entries

imem = '{}'

V C S   S i m u l a t i o n   R e p o r t

Here is what is going on. We declare two associative arrays, namely, “St” (string index) and “imem” (wildcard index). We initialize “St” in its declaration and see if the index “Peter” exists in the array.

Since, “Peter” does exist as an index in the array “St,” we get the following display:

Index Peter exists

Next, we assign three different data values to the wildcard array “imem” at different indices. Since we stored values at three indices, the display shows how many entries are in “imem”:

imem has 3 entries

Then, we check to see if index “4'b1000” exists in “imem.” Since, it does exist, we get the display:

Index 4b'1000 exists

Then, we delete index 4'b1000 and again check to see if it exists in “imem.” Well, it does not anymore, so we get the following display. Note that if the index were not

specified with the `delete( )` method, all entries of the array would have been deleted. So, `imem.delete( )` would delete all entries of the array `imem`:

Index 4'b1000 does not exist

Then, we use three methods; `first( )`, `next( )`, and `last( )` to query “`imem`.” The `first( )` returns the value of the smallest index in “`imem`,” which is 3. Then, `next( )` returns the next smallest index after the first one that we just queried which is 4'b1000. And `last( )` returns the largest index in the array which is “`ffff`.” These three queries are displayed as follows:

First entry is at index 3

Next entry is at index 1000

Last entry is at index `ffff`

Then we delete all entries of “`imem`” by using “`imem.delete( );`” And we display the # of entries in “`imem`” (which should now be 0) and the “`imem`” array itself which should be empty:

`imem` has 0 entries

`imem = '{ }`

### 3.4.6 Associative Array – Default Value

You can assign a default value to an associative array during its declaration. Associative array literals use the ‘{index:value} syntax with an optional default index.

Here is a simple example:

```
module assoc_arr;
    string words [int] = '{default: "hello"};
```

```
initial begin
    $display("words = %p", words['hffff]); //default
    $display("words = %p", words[0]); //default
    $display("words = %p", words[1000]); //default

    words['hffff] = "goodbye";
    $display("words = %p", words);
    $display("words = %p", words[100]); //default

    end
endmodule
```

*Simulation log:*

`words = "hello"`

```
words = "hello"
words = "hello"
words = '{0xffff:"goodbye"}'
words = "hello"
V C S S i m u l a t i o n R e p o r t
```

Array “words” have been initialized with a default value of “hello.” This means that for any location that has *not* been assigned a value will store “hello” in it. The simulation log shows that when we accessed indices “hffff” or “0” or “1000,” the array shows the default stored value “hello.” This is because we have not stored anything at these indices. Then we assign index ‘hffff’ a string value “goodbye.” The array “words” displays the new stored value “goodbye” – and it does *not* show the default value. But if you access a location other than the assigned one at “hffff,” you will still get the default value “hello” as shown in the last line of simulation log.

### 3.4.7 Creating a Dynamic Array of Associative Arrays

This is bit esoteric to show that associative arrays can be stored as elements of dynamic arrays. Here is a simple example:

```
module assoc_arr;
    //Create a dynamic array whose elements are associative arrays
    int games [ ] [string];

    initial begin
        //Create a dynamic array with size of 3 elements
        games = new [3];

        //Initialize the associative array inside each dynamic
        //array element
        games [0] = '{ "football" : 10, "baseball" : 20, "hututu":70 };
        games [1] = '{ "cricket" : 30, "ice hockey" : 40 };
        games [2] = '{ "soccer" : 50, "rugby" : 60 };

        // Iterate through each element of dynamic array
        foreach (games[element])

            // Iterate through each index of the current element in
            // dynamic array
            foreach (games[element][index])
                $display ("games[%0d][%s] = %0d", element, index,
                games[element][index]);
```

```

    end
endmodule

```

*Simulation log:*

```

games[0][baseball] = 20
games[0][football] = 10
games[0][hututu] = 70
games[1][cricket] = 30
games[1][ice hockey] = 40
games[2][rugby] = 60
games[2][soccer] = 50

```

V C S   S i m u l a t i o n   R e p o r t

Here is how the code works. We declare a dynamic array “int games [ ]” whose elements are associative arrays “int games [ ] [string].” The associative array indices are of type string and will store int type values.

First, we instantiate the dynamic array of three elements. So, you have now created three elements of the dynamic array, games[0], games[1], and games[2], each one storing an associative array. We then store int type values at associative array string indices, for each of the dynamic array element. Then we display the entire dynamic array comprising of associative arrays.

## 3.5 Array Manipulation Methods

SystemVerilog provides several built-in methods to facilitate array searching, ordering, locating, and reduction. They are useful for finding and indexing elements of an array. The return type of these methods is a queue of “int” except for the associative arrays which return a queue of the same type as the associative array index type. Associative arrays with wildcard index are not allowed for these methods.

You can search array elements or indices with the *with* expressions. What is a *with* clause? When array manipulation methods iterate over the array elements, these methods are used to evaluate an expression specified with the *with* clause. The *with* clause accepts an expression enclosed in parenthesis. *With* clause is mandatory for some methods and optional for others.

### 3.5.1 Array Locator Methods

The array locator methods in (Table 3.2) require the “with” clause. It is mandatory. These methods are used to filter out certain elements from an existing array based on a given expression. All such elements that satisfy the given expression are returned in a queue.

Let us look at example that uses these methods:

**Table 3.2** Locator methods using the *with* clause

Locator method ( <i>with</i> clause mandatory)	Description
find ()	Returns index of the first element satisfying <i>with</i> expression
find_index ()	Returns the indices of all the elements satisfying the given expression
find_first ()	Returns the first element satisfying the given expression
find_first_index ()	Returns the index of the first element satisfying the given expression
find_last ()	Returns the last element satisfying the given expression
find_last_index ()	Returns the index of the last element satisfying the given expression

```

module arrayLocator;
    string str[5] = {"bob", "kim", "derek", "bob", "kim"};
    string ques[$]; //queue of string type
    int intA[int]; //associative array
    int quei[$]; //queue of int type
    int x;

    initial begin
        intA[1] = 3;
        intA[2] = 2;
        intA[3] = 6;
        intA[4] = 7;
        intA[5] = 3;

        //returns all elements stratifying the 'with' expression
        quei = intA.find( x ) with ( x > 5 );
        $display("find(x)::quei=%0p",quei);

        //returns all elements stratifying the 'with' expression
        quei = intA.find( x ) with ( x < 5 );
        $display("find(x)::quei=%0p",quei);

        //returns the indices of all elements
        //that satisfy the 'with' expression
        //quei = intA.find_index with (item == 3);
        quei = intA.find_index with (item > 1);
        $display("find_index::quei=%0p",quei);

        //returns the first element satisfying 'with' expression
        quei = intA.find_first with (item > 3);
        $display("find_first::quei=%0p",quei);

        //returns the first element satisfying 'with' expression

```

```

ques = str.find_first with (item > "bob");
$display("find_first::ques=%0p",ques);

//returns the last element satisfying 'with' expression
ques = str.find_last with (item < "kim");
$display("find_last::ques=%0p",ques);

//returns index of the first element satisfying 'with'
//expression
quei = intA.find_first_index (s) with (s<5);
$display("find_first_index::quei=%0p",quei);

//returns index of the last element satisfying 'with'
//expression
quei = intA.find_last_index (s) with (s<5);
$display("find_last_index::quei=%0p",quei);

end
endmodule

```

*Simulation log:*

```

find(x)::quei='{6, 7}
find(x)::quei='{3, 2, 3}
find_index::quei='{1, 2, 3, 4, 5}
find_first::quei='{6}
find_first::ques={"kim"}
find_last::ques={"bob"}
find_first_index::quei='{1}
find_last_index::quei='{5}

```

V C S   S i m u l a t i o n   R e p o r t

Let us dissect the code. We first declare various string and int type arrays as well as queues of string and int type. These queues are required because return values from methods can only be stored in queues.

Then we assign values to the elements of the int array “intA.”

We first use *find* locator method as follows:

```

quei = intA.find( x ) with ( x > 5 );
$display("find(x)::quei=%0p",quei);

```

*find* will return all elements that are  $>5$ . The elements of the intA array are 3,2,6,7,3. So, those greater than 5 are 6 and 7. That is shown in the simulation log as follows:

```
find(x)::quei='{6, 7}
```

Next, we again use *find*, but this time we look for elements <5:

```
quei = intA.find( x ) with ( x < 5 );
$display("find(x)::quei=%0p",quei);
```

These elements in “intA” array are 3,2,3. This is shown in simulation log as follows:

```
find(x)::quei='{3, 2, 3}
```

Next, we use “*find\_index*,” which will return the indices of all elements (of array “intA”) that satisfy the *with* expression. The code is:

```
quei = intA.find_index with (item > 1);
$display("find_index::quei=%0p",quei);
```

The elements in “intA” array are 3,2,6,7,3. All array indices have values >1. Hence, the simulation log shows:

```
find_index::quei='{1, 2, 3, 4, 5}
```

Next, we look for the first element using the *with* clause. Here is the code for that:

```
quei = intA.find_first with (item > 3);
$display("find_first::quei=%0p",quei);
```

The first element where item >3 is 6 (from the values 3,2,6,7,3) as shown in the simulation log:

```
find_first::quei='{6}
```

Next, we search the string array for the first element with “item > bob.” Note that the string array is defined as “string str[5] = {'bob”, “kim”, “derek”, “bob”, “kim”};”:

```
ques = str.find_first with (item > "bob");
$display("find_first::ques=%0p",ques);
```

The first element where the string is > “bob” is “kim” and that is shown in the simulation log:

```
find_first::ques='{"kim"}
```

Similarly, we do *find\_last* with “item < kim”:

```
ques = str.find_last with (item < "kim");
$display("find_last::ques=%0p",ques);
```

shows the following in simulation log:

```
find_last::ques='{"bob"}'
```

Next we search for the index of the first element in “intA” array where the index is  $<5$ :

```
quei = intA.find_first_index (s) with (s<5);
$display("find_first_index::quei=%0p",quei);
```

The elements in “intA” array are 3,2,6,7,3. The first element that is less than 5 in the “intA” array is at index # 1. Hence the simulation log shows the index as index # 1:

```
find_first_index::quei='{1}'
```

Lastly, we search for the last index in “intA” array where  $s < 5$ :

```
quei = intA.find_last_index (s) with (s<5);
$display("find_last_index::quei=%0p",quei);
```

The elements in “intA” array are 3,2,6,7,3. So the last index where  $s < 5$  is at index # 5. That is what is shown in the simulation log:

```
find_last_index::quei='{5}'
```

Now let us look at other locator methods where the *with* clause is optional. They are shown in Table 3.3.

Nothing explains better than an example. So, here is one:

**Table 3.3** Locator methods where *with* clause is optional

Locator method ( <i>with</i> clause is optional)	Description
min ()	Returns the element with the minimum value or whose expression evaluates to a minimum
max ()	Returns the element with the maximum value or whose expression evaluates to a maximum
unique ()	Returns all elements with unique values or whose expression evaluates to a unique value
unique_index ()	Returns the indices of all elements with unique values or whose expression evaluates to a unique value

```

module arrayLocator;
    string str[5] = {"bob", "kim", "Derek", "bob", "kim"};
    string ques[$]; //queue of strings
    int intA[int]; //associative array
    int quei[$]; //queue of int
    int x;

initial begin
    intA[1] = 3;
    intA[2] = 2;
    intA[3] = 6;
    intA[4] = 7;
    intA[5] = 3;

    // Find smallest item
    quei = intA.min;
    $display("quei=%p", quei);

    // Find string with largest numerical value in 'str'
    ques = str.max;
    $display("ques=%p", ques);

    // Find all unique string elements in 'str'
    ques = str.unique;
    $display("ques=%p", ques);

    // Find all unique indices in 'intA'
    quei = intA.unique_index;
    $display("quei=%p", quei);

end
endmodule

```

*Simulation log:*

```

quei={2}
ques={"kim"}
ques={"bob", "kim", "Derek"}
quei={1, 2, 3, 4}

```

Just as in previous example, we declare string and int data types as well as queues. We then initialize first five elements of the array “intA.” Then we use the method “min” to search for the element with the smallest value in the “intA” array:

```
// Find smallest item
quei = intA.min;
$display("quei=%p", quei);
```

The elements of “intA” are 3,2,6,7,3. So, the smallest value is 2, and that is what we see in the simulation log:

```
quei='{2}
```

Next, we use the method max to search for the element with a maximum numerical value in the string array “str”:

```
ques = str.max;
$display("ques=%p", ques);
```

“str” has the values “bob,” “kim,” “Derek,” “bob,” and “kim.” The string with the largest numerical value is “kim,” and that is what we see in the simulation log:

```
ques='{"kim"}
```

Next, we look for all the unique elements in the string “str”:

```
ques = str.unique;
$display("ques=%p", ques);
```

“str” has the values “bob,” “kim,” “Derek,” “bob,” and “kim.” Since, “bob” and “kim” are repeated, they are not unique. Hence, we see the following in simulation log:

```
ques={"bob", "kim", "Derek"}
```

Finally, we search for all unique indices in the array “intA”:

```
quei = intA.unique_index;
$display("quei=%p", quei);
```

What this means is that search “intA” and find those *indices* which has unique values. The elements of “intA” are 3,2,6,7,3. So, the values at indices 1,2,3,4 are unique. The last value 3 at index 5 is a repeat and hence is not unique. Hence, we see the following in simulation log:

```
quei='{1, 2, 3, 4}
```

### 3.5.2 *Array Ordering Methods*

Array ordering methods reorder the elements of any unpacked array (fixed or dynamically sized) except for associative arrays. These methods are described in Table 3.4.

Here is an example:

```

module arrayOrder;
    string str[5] = {"bob", "george", "ringo", "john",
"paul"};
    int intA[8] = {3,2,1,6,8,7,4,9};

initial begin

$display("BEFORE 'str' reverse: str=%p", str);
//str.reverse (x) with (x > 5);
//Compile ERROR - can't use 'with' clause
str.reverse;
$display("AFTER 'str' reverse: str=%p", str);

$display("BEFORE 'intA' sort: intA=%p", intA);
//intA.sort (x) with (x > 6); //OK - 'with' clause is ok
intA.sort;
$display("AFTER 'intA' sort: intA=%p", intA);

$display("BEFORE 'intA' rsort: intA=%p", intA);
//intA.rsort (x) with (x > 3); //OK - 'with' clause is ok
intA.rsort;
$display("AFTER 'intA' rsort: intA=%p", intA);

$display("BEFORE 'intA' shuffle: intA=%p", intA);
//intA.shuffle (x) with (x < 5); //Compile ERROR -
//cannot use 'with' clause
intA.shuffle;
$display("AFTER 'intA' shuffle: intA=%p", intA);

end
endmodule

```

*Simulation log:*

BEFORE 'str' reverse: str='{"bob", "george", "ringo", "john", "paul"}'

AFTER 'str' reverse: str='{"paul", "john", "ringo", "george", "bob"}'

BEFORE 'intA' sort: str='{3, 2, 1, 6, 8, 7, 4, 9}'

**Table 3.4** Array ordering methods

Array ordering method	Description
reverse ()	Reverses the order of the elements in the array. Note that specifying a <i>with</i> clause will result in a compile error
sort ()	Sorts the array in ascending order, optionally using the expression in the <i>with</i> clause
rsort ()	Sorts the array in descending order, optionally using the expression in the <i>with</i> clause
shuffle ()	Randomizes the order of the elements in the array. Note that specifying a <i>with</i> clause will result in a compile error

```

AFTER 'intA' sort: intA='{1, 2, 3, 4, 6, 7, 8, 9}
BEFORE 'intA' rsort: intA='{1, 2, 3, 4, 6, 7, 8, 9}
AFTER 'intA' rsort: intA='{9, 8, 7, 6, 4, 3, 2, 1}
BEFORE 'intA' shuffle: intA='{9, 8, 7, 6, 4, 3, 2, 1}
AFTER 'intA' shuffle: intA='{2, 4, 1, 6, 7, 3, 9, 8}
```

### V C S   S i m u l a t i o n   R e p o r t

The main point to note is that you cannot use the *with* clause with *reverse* and *shuffle* methods. You will get compile time errors that will look like the following (Synopsys – VCS):

Error-[IAMC] Invalid array manipulation method call  
testbench.sv, 11

Wrong usage of array manipulation method 'reverse' as shuffle/reverse methods with 'with' clause not allowed.

The *with* clause is ok to use with *sort* and *rsort* methods.

### 3.5.3 *Array Reduction Methods*

Array reduction methods are applied to any unpacked array of integral values to reduce the array to a single value. You can use an optional *with* clause to specify the values used in the reduction method. The methods return a single value of the same type as the array element type. Table 3.5 describes the array reduction methods.

Here is an example:

**Table 3.5** Array reduction methods

Reduction method	Description
sum ()	Returns the sum of all the array elements or, if a <i>with</i> clause is specified, returns the sum of the values yielded by evaluating the expression for each array element
product ()	Returns the product of all the array elements or, if a <i>with</i> clause is specified, returns the product of the values yielded by evaluating the expression for each array element
and ()	Returns the bitwise AND (&) of all the array elements or, if a <i>with</i> clause is specified, returns the bitwise AND of the values yielded by evaluating the expression for each array element
or ()	Returns the bitwise OR (!) of all the array elements or, if a <i>with</i> clause is specified, returns the bitwise OR of the values yielded by evaluating the expression for each array element
xor ()	Returns the bitwise XOR (^) of all the array elements or, if a <i>with</i> clause is specified, returns the bitwise XOR of the values yielded by evaluating the expression for each array element

```

module arrayReduction;

int intA[4] = '{4,3,2,1};
logic [7:0] intB [2][2] = '{ '{1,2}, '{3,4} };
int y;

initial begin

y = intA.sum;
$display("intA.sum = %0d",y); //sum = 10 (4+3+2+1)

y = intA.sum with ( item + 1);
$display("intA.sum = %0d",y); //sum=14 (5+4+3+2)

//y = intB.sum; //Compile ERROR
//y = intB.sum with (item.sum); //OK
y = intB.sum with (item.sum with (item)); //OK
$display("intB.sum = %0d",y); //sum = 10 (1+2+3+4)

//y = intB.xor; //Compile Error
//y = intB.xor(item) with (item > 0); //Compile Error
y = intB.xor(item) with (item.xor); //OK
$display("intB.xor = %0h",y); //xor = 4 (1^2^3^4)

```

```

y = intA.product;
$display("intA.product = %0d",y); //product = 24 (4*3*2*1)

y = intA.product(item) with (item + 1);
$display("intA.product = %0d",y); //product = 120 (5*4*3*2)

y = intA.and;
$display("intA.and = %0h",y); //'and' = 0 (4&3&2&1)

y = intA.or;
$display("intA.or = %0h",y); //'or' = 7 (4 || 3 || 2 || 1)

end
endmodule

```

*Simulation log:*

```

intA.sum = 10
intA.sum = 14
intB.sum = 10
intB.xor = 4
intA.product = 24
intA.product = 120
intA.and = 0
intA.or = 7

```

### V C S   S i m u l a t i o n   R e p o r t

Note that we are using the reduction methods both on 1-D unpacked array (int intA[4]) and 1-D packed 2-D unpacked array (logic [7:0] intB [2][2]). There are differences in how you apply reduction methods to these arrays. The issue here is that the reduction operators only work on one dimension of an array. If you tried to use the sum reduction method on a 2-D array, you would be attempting to sum elements that were 1-D arrays.

With array “intA,” you may or may not use the *with* clause. With array “intB,” you must use the *with* clause. Else, you will get a compile error. So, the following statement will give a compile error:

```
y = intB.sum;
```

The error (from Synopsys – VCS) is as follows:

```
Error-[IMDARMC] Illegal MDA reduction method call
testbench.sv, 18
arrayReduction, "intB.sum"
```

Method 'sum' is not defined on a sub-array.

Please make sure that the 'sum' operation is defined for the type of the

'with' clause/expr or the array element.

Similarly, "y = intB.xor" will give similar error.

Also, since intB is a multi-dimensional array, the value to be compared with the *with* clause must have correct packed/unpacked dimensions:

```
y = intB.xor(item) with (item > 0); //Compile Error
```

You will get the following compile error from Synopsys – VCS+

Error-[AAC] Illegal arguments in comparison

testbench.sv, 24

arrayReduction, "(\_vcs\_item\_vcs\_2 > 0)"

Illegal aggregate comparison because arrays to be compared must have identical packed/unpacked dimensions.

Error-[ARMORWC] Illegal use of array reduction method

testbench.sv, 24

Array reduction method 'xor' cannot be used if with clause evaluated to a real type data

Comments in the code and simulation log explain working of the example.

# Chapter 4

## Queues



**Introduction** This chapter explores nuances of SystemVerilog “queues,” including queue methods, queue of queues, dynamic array of queues, etc.

SystemVerilog provides several data structures to store a collection of objects. A queue is such a data structure. A queue is a variable-size, ordered collection of homogeneous (same type) elements. A queue supports access to all its elements as well as insertion and removal at the beginning or the end of the queue. The 0th position represents the first and \$ represents the last element of the queue. A queue is actually a one-dimensional unpacked array. Queues are declared using the same syntax as unpacked arrays. The only difference is that a queue can grow and shrink automatically. Thus, like arrays, queues can be manipulated using the indexing, concatenation, slicing operator syntax, and equality operators. The difference between an array and a queue is that an array is a non-variable size collection of the same type of variables, while a queue is a variable size ordered collection of homogeneous objects.

Queues can be used to model last-in, first-out (LIFO) or first-in, first-out (FIFO) buffers. A queue can have variable length, including a length of 0. This makes a queue an ideal candidate as a storage element that can shrink or grow as elements are deleted or added to it without fixing an artificial upper limit on its size as a regular fixed size array.

Queues are declared using the same syntax as unpacked arrays but specifying “\$” as the array size. Here’s the syntax:

```
data_type queue_name [$ : <max_size>];
```

Let us look at a simple example of queues:

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_4](https://doi.org/10.1007/978-3-030-71319-5_4)) contains supplementary material, which is available to authorized users.

```

module dq;
  // A queue of 8-bit bytes - unbounded queue
  bit[7:0] dq1[$];

  // A queue of strings - unbounded queue
  string mname[$] = { "Bob" };

  // An initialized queue - unbounded queue
  bit[15:0] dq2[$] = { 3, 2, 7, 1 };

  // A bounded queue - size = 256. Maximum index @255.
  bit q2[$:255];

  //bit q2[255:$]; // Compile ERROR - invalid syntax

  int dq2_size;

initial
begin
  dq1[0] = 'hff;
  dq1[1] = 'h00;
  dq1[$] = 'h01; //last entry - will override dq1[1]='h00
  $display($stime,,, "dq1=", dq1);

  dq1[1] = 'h02;
  $display($stime,,, "dq1=", dq1);

  mname[1] = "mike"; //push-in - grow the queue
  $display($stime,,, "mname=", mname);

  //displays initialized 4 entries
  $display($stime,,, "dq2=", dq2);
  dq2[4] = {16'h 1111};
  $display($stime,,, "dq2=", dq2);

  q2[0] = 1;
  q2[1] = 0;
  $display($stime,,, "q2=", q2);

  q2[3] = 1; //skipped entry '2' - so no 3rd entry
  $display($stime,,, "q2=", q2);

  dq2_size = dq2.size( );

```

```

$display($stime,,"dq2 size = %0d",dq2_size);

      for (int i = 0; i < dq2_size; i++) //read the
entire queue
$display($stime,,"dq2[%0d] = %0h", i, dq2[i]);

//insert a value at index 256 which is out of bound
//dq2.insert(256,1); //You get a run-time Error
end
endmodule

```

There are two types of queues declared in this example: the bounded queue and the unbounded queue.

Bounded queue is “q2[\$:255];.” Bounded means the maximum index of the queue is bounded. In this example, maximum index is 255, meaning the number of elements that it can store is 256 (0 to 255). If you try to push in element at the 256th index, there will be an overflow and the data will be lost.

The unbounded queues are:

```

bit[7:0] dq1[$]; // A queue of 8-bit bytes - unbounded queue
string mname[$] = { "Bob" }; // A queue of strings -
unbounded queue
bit[15:0] dq2[$] = { 3, 2, 7, 1 }; // An initialized queue -
unbounded queue

```

The size of an unbounded queue is how many elements are pushed in (or how many have been initialized). This size will keep changing as you push and pop entries from the queue, but there is no upper bound.

Now let us go through the example:

bit[7:0] dq1[\$]; declares an unbounded queue of 8-bit bytes. Note that a queue is essentially an unpacked array with a \$ sign.  
 string mname[\$] = { "Bob" }; declares an unbounded queue of string type and has the first element initialized.  
 bit[15:0] dq2[\$] = { 3, 2, 7, 1 }; declares an unbounded queue of 16-bit words with the first four elements initialized.  
 bit q2[\$:255]; declares a bounded queue of bits. Note that q2[255:\$] is invalid syntax and will result in a compile time error. Without such size limitation, the queue can grow or shrink indefinitely (well, a simulator may impose a certain limit in order to not crash simulation).

Rest of the code can be explained using simulation log:

*Simulation log:*

```

0 dq1={'hff, 'h1}
0 dq1={'hff, 'h2}

```

```

0 mname='{"Bob", "mike"}
0 dq2={'h3, 'h2, 'h7, 'h1}
0 dq2={'h3, 'h2, 'h7, 'h1, 'h1111}
0 q2={'h1, 'h0}
0 q2={'h1, 'h0}
0 dq2 size = 5
0 dq2[0] = 3
0 dq2[1] = 2
0 dq2[2] = 7
0 dq2[3] = 1
0 dq2[4] = 1111

```

### V C S S i m u l a t i o n R e p o r t

First we assign values ‘ff’ and ‘00’ to the first two elements of “dq” (dq[0] and dq[1]). Right after that, we assign the last entry (dq1[\$]) (as indicated by \$ sign) ‘h01 to dq1. Now, we had assigned two entries 0 and 1 in previous statements. So, the last entry will be at address dq1[1]. So, when we explicitly assign dq1[\$], it will overwrite the last entry which was the entry dq1[1]. So, when we display dq1, we get the following:

```
0 dq1={'hff, 'h1}
```

Then we rewrite dq1[1] = 'h02. So, now dq1 will have the following values:

```
0 dq1={'hff, 'h2}
```

We now push in a new value “mike” to the queue “mname” and display the array. Since the array already had the initial value “Bob”, the display will show the following:

```
0 mname='{"Bob", "mike"}
```

We first display dq2 which has been initialized in its declaration. So that’s what we see in the simulation log:

```
0 dq2={'h3, 'h2, 'h7, 'h1}
```

Then we push into dq2 the value 16’h1111. Then we display dq2 again to show the newly pushed in value:

```
0 dq2={'h3, 'h2, 'h7, 'h1, 'h1111}
```

Then, we assign  $q2[0] = 1$  and  $q2[1] = 0$ . So, the  $q2$  display shows:

```
0 q2={'h1,'h0}
```

But then, we assign  $q2[3] = 1$ . But note that we have not assigned anything to  $q2[2]$ . So, the queue/simulator will simply ignore this assignment. Nothing will be stored in  $q2[3]$ . To prove that, we again display  $q2$  and see that the newly added value does not exist:

```
0 q2={'h1,'h0}
```

We then read out the entire queue,  $dq2$ . For that, first we get the size of the queue and store it in  $dq2\_size$ . Then we loop through the queue for the entire size and display each element. Here is the simulation log when we read out the entire queue:

```
0 dq2 size = 5
0 dq2[0] = 3
0 dq2[1] = 2
0 dq2[2] = 7
0 dq2[3] = 1
0 dq2[4] = 1111
```

Finally, we insert a value at index 256 (“ $dq2.insert(256);$ ”) (“insert” is explained next). Since “ $dq2$ ” is a bounded array with maximum index at 255, trying to insert a value at index 256 will give us a run-time error as follows (Synopsys – VCS):

Error-[DT-MCWII] Method called with invalid index

testbench.sv, 45

“insert” method called with invalid index (index:256)

Please make sure that the index is positive and less than size.

## 4.1 Queue Methods

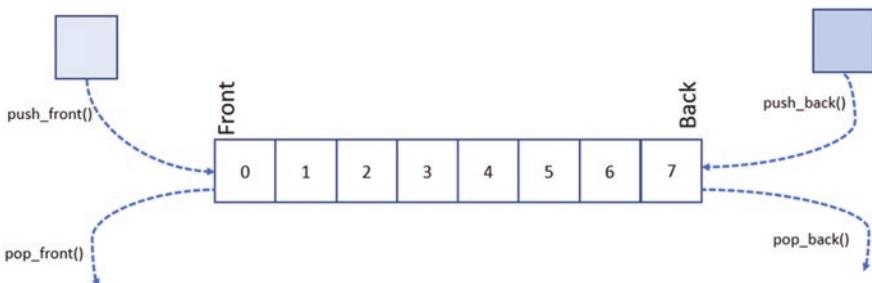
Queues provide many built-in methods. Table 4.1 described these methods.

Here is the graphical representation (Fig. 4.1) of “push” and “pop” methods.

Let us look at an example of how these methods work:

**Table 4.1** Queue methods

Method	Description
<code>size()</code>	<b>function int size();</b> The <code>size()</code> method returns the number of items in the queue. If the queue is empty, it returns to 0.
<code>insert()</code>	<b>function void insert(<input b="" element_t="" index,="" input="" integer="" item);<=""/> The <code>insert()</code> method inserts the given item at the specified index position If the index argument has any bits with unknown (x/z) value, or is negative, or is greater than the current size of the queue, then the method call will have no effect on the queue and may cause a warning to be issued</b>
<code>delete()</code>	<b>function void delete( [input integer index] );</b> If the index is specified, then the <code>delete()</code> method deletes the item at the specified index position If the index is not specified, then the <code>delete()</code> method deletes all the elements in the queue, leaving the queue empty If the index argument has any bits with unknown (x/z) value, or is negative, or is greater than or equal to the current size of the queue, then the method call shall have no effect on the queue and may cause a warning to be issued
<code>pop_front()</code>	<b>function element_t pop_front();</b> The <code>pop_front()</code> method removes and returns the first element of the queue
<code>pop_back()</code>	<b>function element_t pop_back();</b> The <code>pop_back()</code> method removes and returns the last element of the queue
<code>push_front()</code>	<b>function void push_front(<input b="" element_t="" item);<=""/> The <code>push_front()</code> method inserts the given element at the front of the queue</b>
<code>push_back()</code>	<b>function void push_back(<input b="" element_t="" item);<=""/> The <code>push_back()</code> method inserts the given element at the end of the queue</b>

**Fig. 4.1** “push” and “pop” of a queue

```

module dq;
bit[7:0] dq1[$]; // A unbounded queue of unsigned 8-bit
int q3[$:5] = {0,1,2,3,4,5}; //bounded queue
int a;

initial begin
    a = dq1.size( ); //empty queue
    $display ($stime,,, "empty dq1 size = %0d",a);

```

```
dq1[0] = 0; dq1[1] = 1; dq1[2] = 2;
$display ($stime,,, "dq1 SIZE = %0d",dq1.size( ));
$display ($stime,,, "dq1=",dq1);

dq1.insert (3,3); //index, value
$display($stime,,, "After Insert dq1 SIZE = %0d",dq1.
size( ));
$display ($stime,,, "dq1=",dq1);

dq1.delete (3); //index
$display ($stime,,, "After delete dq1 SIZE = %0d",dq1.
size( ));
$display ($stime,,, "dq1=",dq1);

a = dq1.pop_front(); //pop first entry of the queue
$display ($stime,,, "dq1 pop front = %0d ",a);
$display ($stime,,, "dq1=",dq1);

a = dq1.pop_back(); //pop last entry of the queue
$display ($stime,,, "dq1 pop back = %0d ",a);
$display ($stime,,, "dq1=",dq1);

//push the first entry of the queue with '4'
dq1.push_front(4);
$display ($stime,,, "push front dq1=",dq1);

//push the last entry of the queue with '5'
dq1.push_back(5);
$display ($stime,,, "push back dq1=",dq1);

q3_size = q3.size + 5; //size > q3 size
//underflow : pop from index 6,7,8,9,10 - run time Warning
for (int i = 0; i < q3_size; i++)
    $display($stime,,, "q3[%0d] = %0d", i, q3.pop_
front( ) );
end

//Solution for underflow - check for size before pop
while (q3.size( ) > 0)
    $display($stime,,, "q3 = %0d", q3.pop_front ( ));

//overflow : push over the bound limit - run time Warning
for (int i = 0; i < q3_size; i++) begin
    q3.push_front( i );
    $display($stime,,, "q3[%0d] :: q3 = %p", i , q3);
```

```

    end

endmodule

```

*Simulation log:*

- 0 empty dq1 size = 0 //empty queue size
- 0 dq1 SIZE = 3 //size after providing values to first three elements
- 0 dq1='{'h0, 'h1, 'h2} //assigned first three elements
- 0 After Insert dq1 SIZE = 4 //Insert value 3 at index 3.
- 0 dq1='{'h0, 'h1, 'h2, 'h3} //shows inserted value
- 0 After delete dq1 SIZE = 3 //delete value at index 3
- 0 dq1='{'h0, 'h1, 'h2} //shows dq1 after deletion
- 0 dq1 pop front = 0 //pop the front index of the queue
- 0 dq1='{'h1, 'h2} //dq1 after element at index 0 is gone (popped)
- 0 dq1 pop back = 2 //pop the back (last) index of the queue
- 0 dq1='{'h1} //the last index/value is gone
- 0 push front dq1='{'h4, 'h1} //push at the front of the queue (value 4)
- 0 push back dq1='{'h4, 'h1, 'h5} //push at the end of the queue (value 5)

Further simulation log is noted in discussion below.

After every insertion, deletion, pop, push, etc., “dq1” is displayed to show what is going on. Note specifically the pop\_front and pop\_back methods, and see how “dq1” changes. Similarly, observe how “dq1” changes because of push\_front and push\_back.

The following case is interesting. We have declared a bounded queue (“q3”) with upper bound index at 5. That means it has a size of 6 (0 to 5). We initialize these six elements (int q3[\$:5] = {0,1,2,3,4,5};). Then, we pop the array *more* than six times:

```

q3_size = q3.size + 5; //size > q3 size
//underflow : pop from index 6,7,8,9,10 - run time Warning
//Queue 'q3' is empty after index 5.
for (int i = 0; i < q3_size; i++)
    $display($stime,,,"q3[%0d] = %0d", i, q3.pop_
front( ) );
end

```

In other words, we are popping from an empty queue (first six locations (index 0 to 5) will be popped, and then we try to pop from indexes 6,7,8,9,10), and that will cause a run-time warning (can't pop from an empty queue – that is an underflow). Since the queue is empty, the values from these locations will be 0. Here is the simulation log and warning from Aldec – Riviera:

# KERNEL: 0 q3[0] = 0

```

# KERNEL:      0  q3[1] = 1
# KERNEL:      0  q3[2] = 2
# KERNEL:      0  q3[3] = 3
# KERNEL:      0  q3[4] = 4
# KERNEL:      0  q3[5] = 5
# RUNTIME: Warning: RUNTIME_0219 testbench.sv (54): Cannot pop from an
empty queue.
# KERNEL: Time: 0 ns, Iteration: 0, Instance: /dq, Process: @INITIAL#15_0@.
# KERNEL:      0  q3[6] = 0
# RUNTIME: Warning: RUNTIME_0219 testbench.sv (54): Cannot pop from an
empty queue.
# KERNEL: Time: 0 ns, Iteration: 0, Instance: /dq, Process: @
INITIAL#15_0@.
# KERNEL:      0  q3[7] = 0
# RUNTIME: Warning: RUNTIME_0219 testbench.sv (54): Cannot pop from an
empty queue.
# KERNEL: Time: 0 ns, Iteration: 0, Instance: /dq, Process: @
INITIAL#15_0@.
# KERNEL:      0  q3[8] = 0
# RUNTIME: Warning: RUNTIME_0219 testbench.sv (54): Cannot pop from an
empty queue.
# KERNEL: Time: 0 ns, Iteration: 0, Instance: /dq, Process: @
INITIAL#15_0@.
# KERNEL:      0  q3[9] = 0
# RUNTIME: Warning: RUNTIME_0219 testbench.sv (54): Cannot pop from an
empty queue.
# KERNEL: Time: 0 ns, Iteration: 0, Instance: /dq, Process: @
INITIAL#15_0@.
# KERNEL:      0  q3[10] = 0

```

So, what is the solution to avoid such underflow situation? In your testbench you may indeed end up popping entries beyond the upper bound and unless you pay attention to hundreds of Warnings, you may not even know that you had an underflow. So, the solution is to check for “size” before popping, as shown below. Limit the “pop” to the size of the queue:

```

while (q3.size( ) > 0)
    $display($stime,,,"q3 = %0d", q3.pop_front ( ));

```

With this code in place, you will keep popping as long as the size is >0 (no popping if the size is 0). With this solution, you will pop exactly six data from “q3” as shown in the simulation log below. There will not be an underflow:

```

# KERNEL:      0  q3 = 0
# KERNEL:      0  q3 = 1
# KERNEL:      0  q3 = 2

```

```
# KERNEL:      0  q3 = 3
# KERNEL:      0  q3 = 4
# KERNEL:      0  q3 = 5
```

The above case was for underflow. The next case is for overflow:

```
for (int i = 0; i < q3_size; i++) begin
    q3.push_front( i );
    $display($stime,,, "q3[%0d]", i );
end
```

In this case, we are pushing in more data than the upper bound of queue “q3” (which is 5). So, there will be an overflow. And the simulator will give the following log and Warnings (Synopsys – VCS). Once the first six locations (0 to 5) are written to, when you try to write to locations 7 to 10, you will get Warnings, and you notice that *the top-most element gets deleted every time you try to push beyond the upper bound*. That is shown in the log below:

```
0  q3[0] :: q3 = '{0}
0  q3[1] :: q3 = '{1, 0}
0  q3[2] :: q3 = '{2, 1, 0}
0  q3[3] :: q3 = '{3, 2, 1, 0}
0  q3[4] :: q3 = '{4, 3, 2, 1, 0}
0  q3[5] :: q3 = '{5, 4, 3, 2, 1, 0}
```

Warning-[DT-HEBQD] Highest-numbered element of bounded queue deleted

testbench.sv, 62

Addition of new element in bounded queue exceeded the queue's bound.

It would cause deletion of highest-numbered element of queue.

```
0  q3[6] :: q3 = '{6, 5, 4, 3, 2, 1} //top element '0' deleted
```

Warning-[DT-HEBQD] Highest-numbered element of bounded queue deleted

testbench.sv, 62

Addition of new element in bounded queue exceeded the queue's bound.

It would cause deletion of highest-numbered element of queue.

```
0  q3[7] :: q3 = '{7, 6, 5, 4, 3, 2} //top element '1' deleted
```

Warning-[DT-HEBQD] Highest-numbered element of bounded queue deleted

testbench.sv, 62

Addition of new element in bounded queue exceeded the queue's bound.

It would cause deletion of highest-numbered element of queue.

```
0  q3[8] :: q3 = '{8, 7, 6, 5, 4, 3} //top element '2' deleted
```

Warning-[DT-HEBQD] Highest-numbered element of bounded queue deleted

testbench.sv, 62

Addition of new element in bounded queue exceeded the queue's bound.

It would cause deletion of highest-numbered element of queue.

```
0 q3[9] :: q3 = '{9, 8, 7, 6, 5, 4} //top element '3' deleted
```

Warning-[DT-HEBQD] Highest-numbered element of bounded queue deleted  
testbench.sv, 62

Addition of new element in bounded queue exceeded the queue's bound.

It would cause deletion of highest-numbered element of queue.

```
0 q3[10] :: q3 = '{10, 9, 8, 7, 6, 5} //top element '4' deleted.
```

Moving on, there are other ways you can achieve deletion, addition, pushing, and popping operations on a queue simply by using its index in effective manner and shifting the queue. Here is an example:

```
module dq;
    // int queue initialized
    int queue1 [$] = { 100,200,300,400,500 };
    int queue2 [$]; // int queue
    int tmp;

    initial begin
        // Get first item of queue1 (index 0) and store in tmp
        tmp = queue1 [0];
        $display("queue1[0] = ",tmp);

        // Get last item of queue1 (index 4) and store in tmp
        tmp = queue1 [$];
        $display("queue1[$] = ",tmp);

        // Copy all elements in queue1 into queue2
        queue2 = queue1;
        $display("queue2 = ",queue2);

        // Empty the queue1 (delete all items)
        queue1 = { };
        //OR you can also do 'queue1.delete( );'

        $display("queue1 = ",queue1);

        // Replace element at index 2 with 150
        queue2[2] = 150;
        $display("queue2 = ",queue2);

        // Inserts value 250 to index# 2
        queue2.insert (2, 250);
```

```

$display("queue2 = ", queue2);

queue2 = { queue2, 220 }; // Append 220 to queue2
$display("queue2 = ", queue2);

// Put 199 as the first element of queue2
queue2 = { 199, queue2 };
$display("queue2 = ", queue2);

// shift out 0th index (effectively 0th index deleted)
queue2 = queue2 [1:$];
$display("queue2 = ", queue2);

// shift out last index (effectively last index deleted)
queue2 = queue2 [0:$-1];
$display("queue2 = ", queue2);

// shift out first and last item
queue2 = queue2 [1:$-1];
$display("queue2 = ", queue2);
end
endmodule

```

*Simulation log:*

```

queue1[0] =      100 //first item of queue1 (index 0)
queue1[$] =      500 //last item of queue1 (index 4)
queue2 = '{100, 200, 300, 400, 500} //queue2 after queue2=queue1
queue1 = '{}' //queue1 after emptying the queue
queue2 = '{100, 200, 150, 400, 500} //after changing queue2[2]=150
queue2 = '{100, 200, 250, 150, 400, 500} //after inserting '250' at index 2
queue2 = '{100, 200, 250, 150, 400, 500, 220} //appending queue2 with value 220
queue2 = '{199, 100, 200, 250, 150, 400, 500, 220} //putting 199 as the first element
queue2 = '{100, 200, 250, 150, 400, 500, 220} //deleting/shifting first element
(i.e. '199')
queue2 = '{100, 200, 250, 150, 400, 500} //deleting/shifting last element (i.e. '220')
queue2 = '{200, 250, 150, 400} //delete/shift first and last element

```

V C S   S i m u l a t i o n   R e p o r t

The comments in the source code and the simulation log explain what is going on. In this example, we did not use the methods such as “delete,” “pop,” “push,” etc. We simply shifted the elements of the array left or right to achieve the same results.

## 4.2 Queue of SystemVerilog Classes

A queue can be defined of integral-type “class.” The queue can hold class objects. Since a class allows for modular development and can hold properties and methods all encapsulated in a single entity, a queue of classes can be especially useful.

Here is an example:

```

class animals;
    string sname;
    int i1;
    function new (string name="UN");
        sname = name;
        i1++;
    endfunction
endclass

module tb;
    // queue of class type 'animals'
    animals alist [$];
    initial begin
        animals f, f2; //declare two variables of type animals
        // Create a new class object 'f' and push into the queue
        f = new ("lion");
        alist.push_front (f);
        // Create another class object 'f2' and push into the queue
        f2 = new ("tiger");
        alist.push_back (f2);
        // Iterate through queue and access each class object
        foreach (alist[i]) begin
            $display ("alist[%0d] = %s", i, alist[i].sname);
            $display ("alist[%0d] = %p", i, alist[i]);
        end
        // Simply display the whole queue
        $display ("alist = %p", alist);
    end
endmodule

```

*Simulation log:*

# KERNEL: alist[0] = lion

```
# KERNEL: alist[0] = '{sname:"lion", i1:1}
# KERNEL: alist[1] = tiger
# KERNEL: alist[1] = '{sname:"tiger", i1:1}
# KERNEL: alist = '{{sname:"lion", i1:1}, {sname:"tiger", i1:1}}
```

The example declares a class called “animals” which is parameterized with the string “name” and initialized to “UN.” In the testbench, we declare two objects “f” and “f2” of type “animals.” Now, you can create a queue of class objects of type “animals.” You can create a queue of class objects and not the class itself, which is obvious. A new instance/object “f” is created which passes “lion” as the initialization string and pushes it into the front of the queue “alist.” Similarly, a new instance “f2” is created and initialized with string “tiger.” “f2” is pushed at the back of the queue. So, now you have a queue of two class objects, “f” and “f2.” We then simply print out the queue elements.

The above example is derived from Chipverify website (ChipVerify, n.d.). All credits are given to the authors of this example.

### 4.3 Queue of Queues: Dynamic Array of Queues

This section is derived from (testbench, n.d.). You can define queue of queues, dynamic array of queues, associative array of queues, etc. Here is an example:

```
module top;
    typedef int Qint[$];

    // dynamic array of queues
    Qint DynamicQ[ ]; // same as int DynamicQ[ ][\$];

    // queue of queues
    Qint QueueQ[$]; // same as int QueueQ[$][\$];

    // associative array of queues
    Qint AssociativeQ[string]; // same as
                                //int AssociativeQ[string][\$];

    initial begin
        // Dynamic array of 2 queues
        DynamicQ = new[2]; //Create dynamic array of size 2
        (queues)

        // initialize queue 0 with three entries
        DynamicQ[0] = {1,2,3};

        // Push onto queue 1
        DynamicQ[1].push_back(1);
```

```

$display("DynamicQ = %p", DynamicQ);

//push/initialize queue of 3 queues
QueueQ[0].push_front(7);
QueueQ[1].push_back(6);
QueueQ[2].push_back(1);
$display("QueueQ = %p", QueueQ);

// Associative array of queues
AssociativeQ["one"].push_back(5);
AssociativeQ["two"] = {5,6,7,8};
$display("AssociativeQ = %p", AssociativeQ);
end
endmodule : top

```

*Simulation log:*

```

DynamicQ = '{'{1, 2, 3} ,'{1} }
QueueQ = '{'{7} ,'{6} ,'{1} }
AssociativeQ = {"one":'{5} , "two":'{5, 6, 7, 8} }

```

In this example, we define three different types of queues. A dynamic array of queues, a queue of queues, and an associative array of queues:

```

// dynamic array of queues
Qint DynamicQ[ ]; // same as int DynamicQ[ ][\$];

// queue of queues
Qint QueueQ[$]; // same as int QueueQ[$][\$];

// associative array of queues
Qint AssociativeQ[string]; // same as
                           //int AssociativeQ[string][\$];

```

Note how each queue can be defined in two different ways.

Then we perform various operations on these queues. First, we create two dynamic arrays of queues. Then we push/initialize these two arrays (of queues):

```

DynamicQ = new[2]; //Dynamic Array size of 2. Create
two dynamic
                           //array of queues
DynamicQ[0] = {1,2,3}; //initialize queue of first dynamic array
DynamicQ[1].push_back(1); //push_back on queue of second
dynamic array

```

When we display DynamicQ, we get the following simulation log, showing the two dynamic array queues:

```
DynamicQ = '{'{'1, 2, 3} , {'1} }
```

Then we push values on three queues of queues:

```
QueueQ[0].push_front(7);  
QueueQ[1].push_back(6);  
QueueQ[2].push_back(1);
```

And see the three queues with their values, in the simulation log:

```
QueueQ = '{'{'7}, {'6}, {'1}}
```

Finally, we create two associative array queues and initialize them with values:

```
//Queue at associative index/key "one"  
AssociativeQ["one"].push_front(5);  
  
//Queue at associative index/key "two"  
AssociativeQ["two"] = {5, 6, 7, 8};
```

And display the associative array of queues:

```
AssociativeQ = {"one":{'5} , "two":{'5, 6, 7, 8} }
```

# Chapter 5

## Structures



**Introduction** This chapter discusses nuances of SystemVerilog structures, including packed structures, unpacked structures, structure within a structure, structure as module I/O, etc.

SystemVerilog “struct” represents a collection of same or *different* data types that can be referenced as a whole or by the individual data types that make up the structure. This is quite different from arrays where the elements are of the *same* data type. If we need to use a collection of different data types, it is not possible using an array.

Structure is a method of packing data of different types. A structure is a convenient method of handling a group of related data items of different data types. The entire group of data in a structure can be referenced as a whole, or the individual data type can be referenced by name.

If you need to use the same structure in multiple modules, you should put the definition of the structure (mostly defined using `typedef) into a SystemVerilog package and then import it into each RTL module. This way you will only need to define the structure once.

By default, the structures are unpacked. You can indeed have a packed struct by explicitly using the keyword “packed” in its declaration. Here is a simple example showing the difference between an array and a struct:

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_5](https://doi.org/10.1007/978-3-030-71319-5_5)) contains supplementary material, which is available to authorized users.

```

// Normal arrays -> a collection of variables of same
data type
int array [10];           // all elements are of type 'int'
bit [7:0] mem [256]; // all elements are of type 'bit'

// Structures -> a collection of variables of same or different
data types
struct
{
    byte val1;
    int val2;
    string val3; }  DataValue;

```

You can of course declare array of structures or arrays within a structure. For example, here is an array of structures.

```
DataValue v1[20]; //array of structures
```

And here's an array within a structure:

```

struct
{
    byte val1;
    int val2[10]; //array within a structure
    string val3; }  DataValue;

```

## 5.1 Packed Structure

A packed structure consists of bit fields, which are packed together in memory without gaps. *Only packed data types are allowed in a packed structure.*

The structure acts as a vector and thus allows vector-type logic and arithmetic operations (such as add, subtract, etc.) or any other operation that can be performed on a vector. As in a vector, the first member is the most significant, and the subsequent members follow in decreasing significance.

Here is an example of packed structure:

```

struct packed signed {
    byte BE; //2-state
    int addr; //2-state
    int data; //2-state
} pStruct; //signed, 2-state

```

Note that in this example, we are explicitly declaring the packed struct as "signed." By default, they are "unsigned." The "signed" and "unsigned" modifiers affect how the entire structure is perceived when used as a vector in mathematical

or logical operations. 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 the part-select of vectors in Verilog.

Also, note that if all the data types within a packed structure are 2-state (as in the above example), the structure as a whole is treated as a 2-state vector. If any data type within a packed structure is 4-state, the structure as a whole is treated as a 4-state vector. If there are also 2-state members in the 4-state structure, there is an implicit conversion from 4-state to 2-state when reading those members and from 2-state to 4-state when writing them.

An example of unsigned structure:

```
struct packed unsigned {
    integer addr;           //4-state
    logic [31:0] data;     //4-state
    int burst;             //2-state
} upStruct; //unsigned, 4-state
```

In this example, the members are both 4-state and 2-state. Hence the structure will be treated as a 4-state vector.

Here is a working example of a packed structure:

```
module SU;

    struct packed {bit [7:0] intr;  //packed struct
                  logic [23:0] addr;
                } SUR;

    initial begin
        SUR.intr = 'hFF;
        $display($stime,,, "SUR = %h",SUR);
        $display($stime,,, "SUR Intr = %h",SUR.intr);

        //Assign by position
        SUR = '{'h00,'hFF_FF_FF};
        $display($stime,,, "SUR = %h",SUR);

        //Assign by name
        SUR = '{intr:'h01, addr:'hf0f0f0};
        $display($stime,,, "SUR = %h",SUR);

        //Assign default
        SUR = '{default:'h123456};
        $display($stime,,, "SUR = %h",SUR);
```

```

//Assign default
SUR = '{default:'h78};
$display($stime,,, "SUR = %h",SUR);

SUR = 0;
SUR = SUR+'h12; //Arithmetic operation.
// packed structure can be used as a vector
$display($stime,,, "SUR = %h",SUR);
end
endmodule

```

*Simulation log:*

```

0  SUR = ffxxxxxx
0  SUR Intr = ff
0  SUR = 00fffff
0  SUR = 01f0f0f0
0  SUR = 56123456
0  SUR = 78000078
0  SUR = 00000012
V C S  S i m u l a t i o n   R e p o r t

```

In this example, we showcase that individual members can be assigned a value, the vector as a whole can be assigned a value, you can assign values by position or by name, and you can do arithmetic operation on the packed structure as a vector. Note that when we display the structure “SUR,” it is displayed as a vector, not as individual fields. The way to assign values to the entire structure is using ‘{...}’ notation. The values within the ‘{...}’ braces are separate values for each structure member and *not* a concatenation of values.

The first member of the packed 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 (LSB) of the vector and is numbered as bit 0. For example:

```

struct packed {
    logic frame_;
    logic [15:0] address;
    logic [31:0] data;
} control;

```

This packed structure is represented as (Fig. 5.1).



**Fig. 5.1** Packed structure

Note that when you initialize a structure as a whole, the number of values must exactly match the number of members in the structure.

*Packed structures must contain packed variables.* All members of a packed structure must be integral values. An integral value is a value that can be represented as a vector, such as “byte” and “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.

Here is another example of packed structure:

```
module top;

typedef enum
{
    READ,
    WRITE
} PCI_COMMAND;

typedef enum
{
    IRDY,
    TRDY
} PCI_IO;

typedef struct packed
{
    PCI_COMMAND  RW;
    PCI_IO      IR;
    logic [15:0] addr;
    logic [15:0] data;
} PCI_BUS;

PCI_BUS pBUS;

initial begin
    // Assign pBUS fields by name
    pBUS.RW = READ;
    pBUS.IR = IRDY;
    pBUS.addr = 16'ha0a0;
    pBUS.data = 16'hf0f0;

    $display("Fields by name :: pBUS = %p", pBUS);
```

```

// Assign pBUS fields by bit position
pBUS[31:16] = 16'h1010;
pBUS[15: 0] = 16'h2020;

$display("Fields by bit position :: pBUS = %p", pBUS);

// Assign all fields of pBUS
pBUS = '{WRITE, TRDY, 16'h3030, 16'h4040};

$display("All fields at once :: pBUS = %p", pBUS);
end
endmodule

```

*Simulation log:*

```

Fields by name :: pBUS = '{RW:READ, IR:IRDY, addr:'ha0a0, data:'hf0f0}
Fields by bit position :: pBUS = '{RW:READ, IR:IRDY, addr:'h1010, data:'h2020}
All fields at once :: pBUS = '{RW:WRITE, IR:TRDY, addr:'h3030, data:'h4040}

```

V C S   S i m u l a t i o n   R e p o r t

This example shows how we can build a structure from different data types. We define two `typedef PCI\_COMMAND and PCI\_IO and use these typedefs into the packed structure PCI\_BUS. We declare a variable pBUS of type PCI\_BUS.

The example shows how the fields of pBUS can be assigned in different ways. First, we assign the fields by name. Then, we assign these fields by bit position, and then we assign all fields at once. The simulation log shows the value of pBUS after each assignment.

## 5.2 Unpacked Structure

By default, structures are unpacked.

They will not occupy memory in a contiguous manner (the memory allocation will be implementation dependent – type of C compiler).

Some examples:

```

struct {
    bit [15:0] bMode;
    byte enable;
} Cycle;

typedef struct {
    bit [31:0] opcode;
    bit R_W;
}

```

```

logic byteEnb;
integer data;
integer addr;
} read_cycle;

read_cycle rC;

```

Note that signing of an unpacked struct is not allowed. The following is illegal and will generate a compile error:

```

struct signed { //ILLEGAL
bit bE;
bit dE;
}

```

Here is an example:

```

module SU;
    struct {bit [7:0] intr; //unpacked struct
            logic [23:0] addr;
        } SURu;

    initial begin
        SURu.intr = 'hFF; //assign to a single field
        $display($stime,,, "SURu = %p", SURu);
        $display($stime,,, "SURu Intr = %h", SURu.intr);

        SURu = {'h00,'hFF}; //assign to all fields
        $display($stime,,, "SURu = %p", SURu);

        //SURu = SURu + 'h12;
        // ERROR- Unpacked struct can't be used as a vector
    end

    typedef struct {
        int addr = 'hff; //default initial value
        int data;
        byte crc [4] = '{4{1}}; //default initial value
    } bus;

    bus b1;

    initial begin
        $display ("\n");

```

```

$display($stime,,, "b1.addr=%h b1.data=%h b1.crc=%p",
b1.addr,b1.data,b1.crc);

b1 = {'h 1010, 'h a0a0, '{1,2,3,4}}; //overrides
defaults
$display($stime,,, "b1.addr=%h b1.data=%h b1.crc=%p",
b1.addr,b1.data,b1.crc);

end
endmodule

```

*Simulation log:*

```

0 SURu ='{intr:'hff, addr:'xxxxxxxx}
0 SURu Intr = ff
0 SURu ='{intr:'h0, addr:'hff}
0 b1.addr=000000ff b1.data=00000000 b1.crc='{1, 1, 1, 1}
0 b1.addr=00001010 b1.data=0000a0a0 b1.crc='{1, 2, 3, 4}
V C S S i m u l a t i o n R e p o r t

```

Note that we will get a compile time error when we try to do arithmetic operation on this unpacked structure. Hence, the following line will generate a compile time error:

```
//SURu = SURu + 'h12;
// ERROR- Unpacked struct cannot be used as a vector
```

Note also that we can assign default values to the individual elements of a structure. The rest of the code is explained with embedded comments in the code.

### 5.3 Structure as Module I/O

You can use an entire structure as an input or output port of a module. A module port can have a SystemVerilog struct type, which makes it easy to pass the same bundle of signals into and out of multiple modules and keep the same encapsulation throughout a design. For example, a wide command bus between two modules with multiple fields can be grouped into a structure to simplify the RTL code.

Here is an example:

```
typedef struct {
    bit [7:0] intr = 'h AA;
    logic [23:0] addr = 'h FF_FF_FF;
} ext;
```

```

module SU (
    output ext extOut);

    assign extOut = '{intr: 8'hFF, addr:24'haa_aa_aa};

initial begin
    #1; $display($stime,,, "extOut = %p", extOut);
end
endmodule

module top;
    ext extIn;
    //connect extOut of 'SU' with extIn of 'top'
    SU SUInst(.extOut(extIn));

initial begin
    #2; $display($stime,,, "extIn = %p", extIn);
end
endmodule

```

*Simulation log:*

```

1 extOut ='{intr:'hff, addr:'aaaaaaaa}
2 extIn ='{intr:'hff, addr:'aaaaaaaa}
V C S   S i m u l a t i o n   R e p o r t

```

In this example, we first define an unpacked structure called “ext.” We instantiate “ext” as “extout” directly in the output port declaration of module “SU” and assign it a value. We then display out “extOut.” That is shown in the simulation log at time 1.

In the module “top,” we declare “extIn” of type “ext” and connect “extIn” of module “top” with the “extOut” of module “SU.” Then we display “extIn” which is displayed at time 2 in the simulation log.

Both displays (time 1 and time 2) are identical which shows that the value assigned to “extOut” in module “SU” is passed as an input to module “top.”

Here is another example of how you can use a struct as module I/O.

First, here is a module definition without any structure as I/O. All fields are separate. This example shows only three fields, but in reality you can have hundreds of I/O, and it would make sense to group related I/Os into different structures and then pass them as I/O to the module:

```

module PCIe_f (
    output bit [11:0] length,
    output bit [63:0] address,
    output bit [63:0] data_payload
);
endmodule

```

Now, let us bundle “length” and “address” in a struct and use that for the I/O of the module:

```
typedef struct {
    bit [11:0] length;
    bit [63:0] address;
} PCIe_header;

module PCIe_f_with_struct (
    output PCIe_header h,
    output bit [63:0] data_payload
);
endmodule
```

Let us take this one step further and pack all the fields into a struct (struck within a struck) and pass the entire struct as module I/O:

```
typedef struct {
    bit [11:0] length;
    bit [63:0] address;
} PCIe_header;

typedef struct {
    PCIe_header h;
    bit [63:0] data_payload;
} PCIe_packet;

module PCIe_f_with_struct (
    output PCIe_packet p
);
endmodule
```

We hierarchically built a struct, which groups relevant signals into a struct and then pass this struct as module I/O. This is a much more efficient way of modeling module I/O.

## 5.4 Structure as an Argument to Task or Function

Structures can be passed as arguments to a task or a function. To do so, the structure must be declared as a user-defined type (typedef), so that the task or function argument can then be declared as the structure type. Here is an example:

```
typedef struct {
    logic [31:0] addr;
    logic [63:0] data;
    logic [3:0] BEnable;
} control;

function Dbus (input control ctl);
...
endfunction
```

## 5.5 Structure Within a Structure

You can indeed embed a structure within another structure. Here is an example:

```
typedef struct {
    bit [31:0] addr;
    bit [31:0] data; } Bus;
Bus myBus;

typedef struct {
    myBus x;
    myBus y; } multiBus;
```

The struct “Bus” is defined as a typedef. A variable of type “Bus” is declared (“myBus”) and used in struct “multiBus” which declares variable of the type “myBus.”

# Chapter 6

## Union



**Introduction** This chapter discusses packed, unpacked, and tagged unions.

A SystemVerilog union allows a single piece of storage to be represented different ways using different named member types. Because there is only a single storage, only one of the data types can be used at a time. Unions like structures contain members whose individual data types may differ from one another.

However, the *members that compose a union all share the same storage area*. A union allows us to treat the same space in memory as a number of different variables. That is, a union offers a way for a section of memory to be treated as a variable of one type on one occasion and as a different variable of a different type on another occasion. The datatype union is very similar to a structure, but only one of the fields will be valid at a given point in time. Unions allow for the same storage location to represent more than one data type, where it is guaranteed that you will only want to store one of the types of data at any one time.

The main difference between union and struct is that union members overlay the memory of each other so that the size of a union is the one, while struct members are laid out one after another (with optional padding in between). Union should be used when you know that you will access any one member at a time. Also, a union is large enough to contain all its members and has an alignment that fits all its members. Union uses the largest member's memory space.

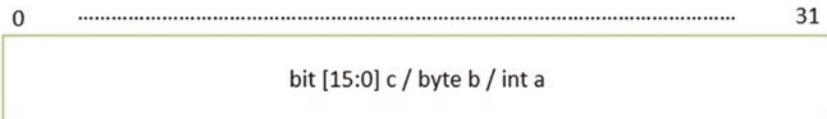
For example:

```
union {
    int a;
    byte b;
    bit [15:0] c;
} data;
```

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_6](https://doi.org/10.1007/978-3-030-71319-5_6)) contains supplementary material, which is available to authorized users.

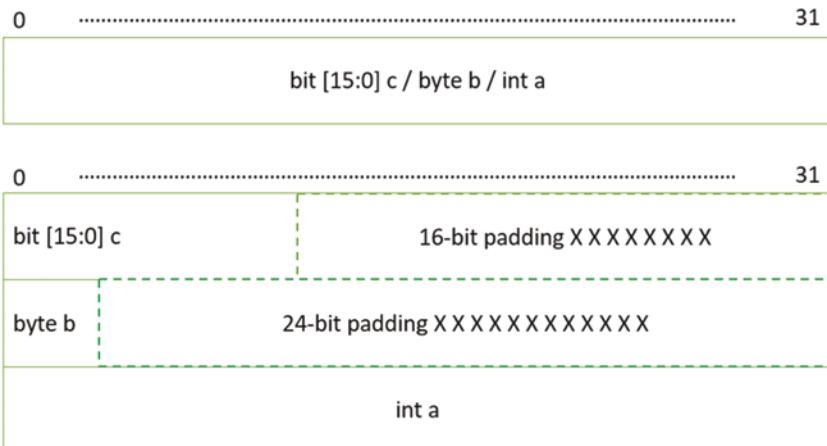
This will be stored as 32-bits in the memory, since “int” is of the largest size. The fields overlay each other. Only one of the fields of this union will be valid at a given point in time. Here is the graphical representation.



If you have a struct with the same fields, as in:

```
struct {
    int a;
    byte b;
    bit [15:0] c;
} data;
```

It will be represented as follows. Struct members are laid out one after another (with optional padding in between). Note that this is an unpacked struct. The following shows alignment at 32-bit boundary. Each field has its own separate memory space.



To reiterate, a structure allocates the total size of all its elements. A union allocates only as much memory as its largest member requires.

Just like with structures, a union can be defined as a type using “typedef.” A union that is defined as a user-defined type is referred to as a *typed union*. For example:

```

typedef union {
    int i;
    bit [15:0] b1;
} data;
data d1, d2;

```

## 6.1 Packed and Unpacked Unions

Unions can also be *packed* and *unpacked* similarly to structures. Only packed data types and integer data types can be used in packed union.

*All members of a packed union must be the same size.*

Like packed structures, packed union can be used as a whole with arithmetic and logical operators, and bit fields can be extracted like from a packed array.

*In an unpacked union, the sizes of data elements can be different. Unions are unpacked by default.*

For example, following is an unpacked union:

```

union { // Unpacked - different sizes OK
    int a;
    byte b;
    bit [15:0] c;
} data;

```

As you notice, the sizes of the data in the union are different; int, byte, and bit[15:0]. This is ok, since this is an unpacked union. But if you took the same union and declared it as “packed,” you will get a compile error:

```

union packed { //NOT OK - packed - sizes must be the same
    int a;
    byte b;
    bit [15:0] c;
} data;

```

You will get the following error (Synopsys – VCS):

Error-[PUMSS] Incorrect packed union members size  
testbench.sv, 7  
"b"

Packed union members must have same size.

Member "b" has different size (8 bits) from next member (16 bits).

Let us look at each union type in detail.

### 6.1.1 Unpacked Unions

We just saw an example of unpacked union above. The sizes of data elements can be different. An unpacked union can contain any variable, including “real,” unpacked structures and unpacked arrays. From RTL point of view, *unpacked unions are not synthesizable*. They are mainly used for high-level/transaction-level modeling in testbenches. Software tools can store values in unpacked unions in an arbitrary manner. There is no requirement that each tool aligns the storage of the different types used within the union in the same way.

Here is a bit more interesting example of an unpacked union as part of a “struct”:

```
struct {
    bit b1;
    union {
        int i;
        real r;
    } uVal;
} sVal;
```

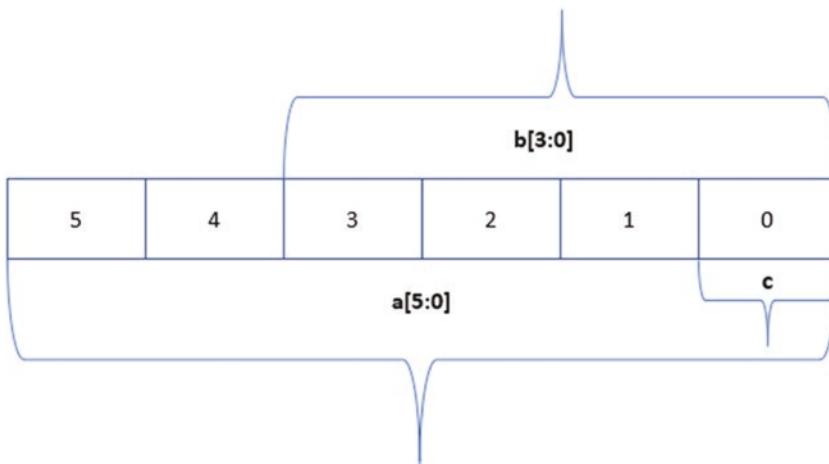
The members of this structure/union can be referenced as:

```
sVal.uVal.i = 5;
sVal.b1 = 1;
```

Following is a simple example (Xilinx) showing how members of an unpacked union are stored in memory:

```
typedef Union {
    logic [5:0] a;
    logic [3:0] b;
    logic c;
} myUnionType
myUnionType myUnion;
```

Since this is an unpacked union, the member sizes can be different. But, the storage will be allocated based on the longest member size, which is “logic [5:0] a.” The memory representation will be as shown below.



Let us look at an example of using a packed “struct” within an unpacked “union” and see how memory is allocated:

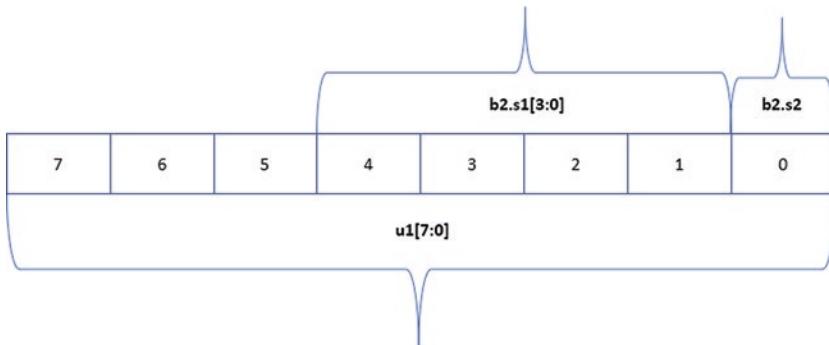
```

typedef struct packed {
    bit [3:0] s1;
    bit s2;
} myStruct;

typedef union {
    logic [7:0] u1;
    myStruct b2;
} mUnionT;
mUnionT Union1;

```

Here is how the union will look in memory. The longest member is “logic [7:0] b1.” So, that is the maximum size of storage for this union.



Now, let us look at an example that shows how union members are written and read and what are the consequences. Reading from an unpacked union member that is different than the last member written may cause indeterminate results (or determinate but the results may look different from what you would expect).

As we saw, only one of the members/fields of a union will be valid at a given point in time. So, if you write to a member (member1) of the union and then read from another member (member2), you will get the value written by member1. This proves that there is only one storage space allocated to all members of a union – all members share the same storage space as we discussed above:

```

module union_example;
    logic [31:0] x;

    typedef union { //OK - Unpacked - different sizes OK
        int a;
        byte b;
        bit [15:0] c;
    } data;

    data d1;

    initial begin
        d1.a = 32'h ffff_ffff; //write to d1.a

        x = d1.b; //read from d1.b
        $display("x = %h",x);

        d1.b = 8'h 01; //write to d1.b

        x = d1.a; //read from d1.a
        $display("x = %h",x);

        d1.c = 16'h 1010; //write to d1.c

        x = d1.a; //read from d1.a
        $display("x = %h",x);

    end
endmodule

```

*Simulation log:*

```
x = ffffffff
x = 00000001
x = 00001010
```

### V C S   S i m u l a t i o n   R e p o r t

This is what is going on in this example. We have declared an unpacked union called “data.” And declared “d1” of type “data.”

In the initial block, we first see how to write to “int a” of the union:

```
d1.a = 32'h ffff_ffff; //write to d1.a
```

Now, note that we have not written anything to “d1.b.” But since there is only one storage for all the members of the union, these members will all have the same value as what is written in “d1.a.” So, when we read from “d1.b” (which hasn’t been written to yet):

```
x = d1.b; //read from d1.b
```

we get the following in simulation log:

```
x = ffffffff
```

This shows that whatever that was written into “d1.a” is reflected on “d1.b.”

Then we write to d1.b:

```
D1.b = 8'h 01;
```

And read from “d1.a.” Now the value written to “d1.b” is reflected on “d1.a” (again, because “d1.a,” “d1.b,” and “d1.c” all share the same memory space). So, in the simulation log, we see that “d1.a” reflects what we wrote into “d1.b”:

```
x = 00000001
```

Lastly, we write to “d1.c”:

```
d1.c = 16'h 1010;
```

and read from “d1.a” which will reflect the value written to “d1.c” as shown in the simulation log:

```
x = 00001010
```

This shows that union members overlay the memory of each other and that the size of a union is the one with longest size. A union only stores a single value, regardless of how many type representations are in the union. To realize the storage of a union in hardware, all members of the union must be stored as the same vector size using the same bit alignment.

So, what if you want to make sure that when you write to a member of the union but then read from another that you get an error? That is possible with tagged unions.

### 6.1.2 Tagged Unions

A union can be declared as tagged, as in:

```
union tagged {
    int a;
    byte b;
    bit [15:0] c;
} data;
```

*tagged* is the keyword. A tagged union contains an implicit member that stores a “tag,” which represents the name of the last union member into which a value was stored. When a value is stored in a tagged union, using a tagged expression, the implicit tag automatically stores information as to which member the value was written.

A tagged union is a type-checked union. That means you can no longer write to the union using one member type and read it back using another. Tagged union enforces type checking by inserting additional bits into the union to store how the union was initially accessed. Due to the added bits, it is unable to freely refer to the same storage using different union members. We will see an example on this: (Sutherland, SystemVerilog for Design)

A value can be written into a tagged union member using a tagged expression. A tagged expression has the keyword *tagged* followed by the member name, followed by a value to be stored. Following the above example:

```
data = tagged a 32'hffff_ffff;
```

Tagged unions require that software tools monitor the usage of the union and generate error message if a value is read from a different union member than the member into which a tagged expression value was last written.

Let us look at a simple example:

```
module tagged_union_example;
logic [31:0] x;

typedef union tagged {
    int a;
    byte b;
    bit [15:0] c;
} data;
```

```

data d1;

initial begin
    d1 = tagged a 32'hffff_ffff; //write to 'a'

    //read from 'b'. Since 'a' was written last, cannot access
    // 'b'. - Error
    x = d1.b;
    $display("x = %h", x);
end
endmodule

```

Simulation of the above code will generate an error as follows:

Error-[TU-INVMEMUSG] Invalid member usage of a tagged union.  
testbench.sv, 15

Member of a tagged union referred is not valid since a different member is in use. The expected tag is 'a', but tag 'b' is used.

Please check which member of the tagged union is in use.

#### V C S   S i m u l a t i o n   R e p o r t

This is what is going on in this example. We create a “tagged” union called “data” and declare “d1” of type “data.”

Then we write a “tagged” expression into the member “int a” of the “d1”:

```
d1 = tagged a 32'hffff_ffff;
```

Then we read from “d1.b.” Two things are happening here. First, we have not written to “d1.b,” but we are reading from it. Second, we wrote to d1.a using a tagged expression, meaning the simulator remembers this last member (d1.a) that was written into. So, when we access “d1.b,” it is an error, since “d1.a” was written to, meaning it is in use. You have to first write to “d1.b” if you want to access it. So, the following code will work:

```

module tagged_union_example;
    logic [31:0] x;

    typedef union tagged {
        int a;
        byte b;
        bit [15:0] c;
    } data;
    data d1;

```

```

initial begin
    d1 = tagged a 32'hffff_ffff; //write to 'a'
    d1 = tagged b 8'h10; //write to 'b'

    //read from 'b' - since 'b' was written last, this is OK
    x = d1.b;
    $display("x = %h", x);
end
endmodule

```

*Simulation log:*

x = 00000010

VCS Simulation Report

### 6.1.3 Packed Union

*All members of a packed union must be the same size.*

Only packed data types and integer data types can be used in packed union. Like packed structures, packed union can be used as a whole with arithmetic and logical operators, and bit fields can be extracted like from a packed array. In a packed union, the number of bits of each union member must be the same. This ensures that a packed union will represent its storage with the same number of bits, regardless of member in which a value is stored. Because of this restriction, packed unions are synthesizable (unpacked unions are not synthesizable). If any member of a packed union is a 4-state type, then the union is 4-state. A packed union cannot contain “real” or “shortreal” variables. It also cannot contain unpacked structures, other unpacked unions, or unpacked arrays.

Following is a packed union:

```

typedef union packed {
    int a;
    bit [31:0] c;
} data;

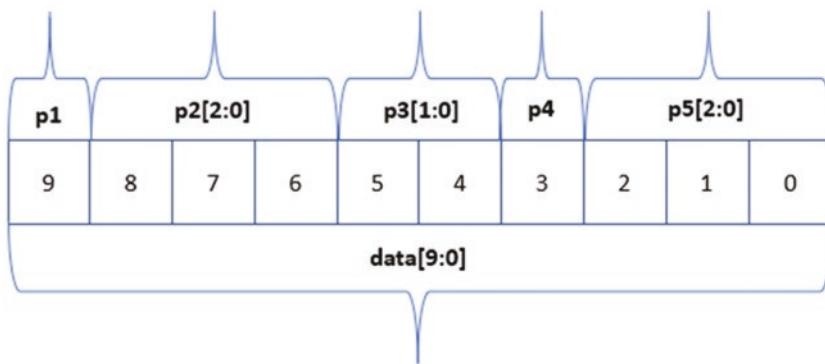
```

*packed* is the keyword. Note that both members are of the same size; “int” which is 32-bits and “bit [31:0] c” which is 32-bits. If they were of different sizes, you will get an error. What we discussed for unpacked unions (i.e., how to write to them, writing to a member and reading from another, etc.) apply here as well, except that all members are of equal size with packed union.

Let us see how memory storage looks like for the following example (Xilinx) of a packed union with a packed “struct” within it:

```
typedef union packed {
    logic [9:0] data;
    struct packed {
        bit p1;
        bit [2:0] p2;
        bit [1:0] p3;
        bit p4;
        bit [2:0] p5;
    } p_modes;
} myUnion;
myUnion Union1;
```

Note that the size of the “struct” is 10-bits wide and so is the size of “data.” In other words, there are two members of this union: the “struct” “p\_modes” and “data,” and the sizes of both members are the same, as required by a packed union (add up the bits of “p\_modes” – it’s 10-bits and so is the size of “data”). Here is how the memory will look like.



# Chapter 7

## Packages



**Introduction** This chapter discusses nuances of packages, how to declare and share them, how to reference data within a package, etc.

SystemVerilog packages provide encapsulation of many different data types, nets, variables, tasks, functions, assertion sequences, properties, and checker declarations. They can be shared among multiple modules, interfaces, programs, and checkers. They have explicitly named scopes (or wildcard scope – accessing all members of a scope) that allow referencing of data in that scope. It provides an uncluttered reusable methodology. Note that items within packages cannot have hierarchical references to identifiers except those created within the package or made visible by the import of another package. Package is a namespace. It also is a top-level object – i.e., it cannot be declared within a module, program, class, or other packages.

Here is an example:

```
package myPack;
    typedef struct {
        int i;
        int j;
    } cStruct;

    function cStruct add (cStruct a , b);
        add.i = a.i + b.i;
        add.j = a.j + b.j;
    endfunction
endpackage
```

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_7](https://doi.org/10.1007/978-3-030-71319-5_7)) contains supplementary material, which is available to authorized users.

```

module top (
    //referencing package item 'cStruct' of myPack
    output myPack::cStruct cout,
    input myPack::cStruct a , b
);

    //referencing 'add' function of myPack
    assign cout = myPack::add(a , b);
endmodule

```

In this example, we declare a package “myPack” which has a struct declaration “cStruct” and a function “add.” They are considered encapsulated in the package “myPack.” In the module “top,” we are using “cStruct” from “myPack” as I/O of the module “top.” Note that we have not explicitly imported the package “myPack.” Instead, we explicitly reference “myPack” with the following syntax:

```
myPack::cStruct
```

This is one way to reference functions, tasks, structs, etc. from a package. Other way is to simply import a package in a module which will then allow access to package data without the need for explicit reference to the package. Here is the above example with “import” of the package:

```

package myPack;
    typedef struct {
        int i;
        int j;
    } cStruct;

    function cStruct add (cStruct a , b);
        add.i = a.i + b.i;
        add.j = a.j + b.j;
    endfunction
endpackage

//importing selective entity from 'myPack'
//import myPack::cStruct;
//import myPack::add;

import myPack::*;

module top (
    output cStruct cout,
    input cStruct a , b
);

```

```
//'add' is visible ... no need for mypack::add(a, b);
assign cout = add(a , b);
endmodule
```

In this example, we can import individual struct and function from “myPack” as shown below:

```
//importing selective entity from 'myPack'
import myPack::cStruct;
import myPack::add;
```

We can import the entire package as well without the need for individual entity import as shown below:

```
import myPack::*; //importing entire myPack
```

Note that once you import a package either way, the struct and the function contained in the package are directly visible in module “top.” Hence, we have “assign cout=add(a,b);” without the need to explicitly call “assign cout=myPack::add(a,b);”.

Here is an example of how referencing variables in imported packages will/will not work (SystemVerilog – LRM):

```
package p;
    typedef enum { FALSE, TRUE } bool;
endpackage

package q;
    typedef enum { ORIGINAL, FALSE } hair;
endpackage

module top1 ;
    import p::*;
    import q::hair;

    hair myhair;

    initial begin
        myhair = q:: FALSE; // OK:
        //myhair = FALSE;
        // ERROR: Direct reference to FALSE refers to the
        // FALSE enumeration literal imported from p
    end
endmodule
```

```

module top2 ;
import p::*;
import q::hair, q::ORIGINAL, q::FALSE;
//import q::*; //will not work.. need above import

hair myhair;

initial begin
    myhair = FALSE;
    // OK: Direct reference to FALSE refers to the
    // FALSE enumeration literal imported from q
end
endmodule

```

In this example, we have declared two packages “p” and “q.” Note that the enum variable FALSE exists in both packages. In module “top1,” we import the entire package p (import p::\*;

\*) and import explicitly the enum type “hair” (import q::hair). And declare “myhair” to be of type “hair.”

Now when we try to assign FALSE (from package “q”) directly to myhair (myhair = FALSE;), it will give an error because the compiler thinks that you are assigning FALSE from package p. You cannot do that since “myhair” is of type “hair” and can (should) only accept FALSE from package q. So, myhair = FALSE will give a compile time error (Synopsys – VCS) that looks like the following:

Error-[ENUMASSIGNTYPE] Different enum types in assignment  
testbench.sv, 20

Different enum types in assignment statement  
Cannot assign a value of type 'p::bool' to a variable of type 'q::hair'  
Source info: myhair = FALSE;

To circumvent this, in module “top2,” we explicitly import the following:

```
import q::hair, q::ORIGINAL, q::FALSE;
```

This makes FALSE from “q” available in scopes where this is imported. So, now when we do myhair = FALSE, it will work.

Note that in module “top2,” if you simply did “import q::\*;,” it will still not work. If you simply did the following in module “top2”:

```

module top2 ;
import p::*;
import q::*;
hair myhair;
initial begin
    myhair = FALSE;

```

```
    end
endmodule
```

You will get the following error (Synopsys – VCS):

Error-[SV-LCM-MWD] Multiple wildcard definition

testbench.sv, 32

top2, "FALSE"

Identifier 'FALSE' getting imported from more than one package  
 p::\* [testbench.sv:25] (Owner Package: p)  
 and q::\* ["testbench.sv", 27] (Owner Package: q)

Make explicit import with the package from where the definition should be picked.

So, this example shows that you need to be careful on how to import packages and reference variables therein.

Let us now look at how importing a package within a package works and the visibility of variables declared in hierarchically imported packages.

Example 1:

```
package P1;
  int a = 1;
endpackage : P1

package P2;
  import P1 :: *;
  int b = 2;
endpackage : P2

module TRY1( );
  import P2 :: *;
  initial begin
    $display("top : P1::a = %0d", a);
    $display("top : P2::b = %0d", b);
  end
endmodule : TRY1
```

*Simulation log (Synopsys – VCS):*

Error-[IND] Identifier not declared

testbench.sv, 14

Identifier 'a' has not been declared yet. If this error is not expected, please check if you have set `default\_nettype to none.

In this example, we declare two packages “P1” and “P2.” “P1” has the variable “a” and “P2” has the variable “b” declared in it. We then import P1::\* into the package “P2”. This means that variable “a” will be visible in package “P2.”

Then in the module “TRY1,” we import P2::\* (but we do not import P1::\*). We try to access both “a” and “b” from the module “TRY1.” Now, you may think that since “P2” imports “P1” that the variables declared in “P1” will be visible in module “TRY1.” But that is not the case. The variable “a” is visible only in package “P2” since that is where import P1::\* took place. In other words, variable “a” is visible only where its package is imported not in another module (or package). You need to understand the implication of importing a package into another package.

Let us see the same example, but here we import both “P1” and “P2” in module TRY1. Let us see if variable “a” is now visible in module TRY1.

Example 2:

```

package P1;
    int a = 1;
endpackage : P1

package P2;
    import P1 :: *;
    int b = 2;
endpackage : P2

module TRY1( );
    import P1 :: *;
    import P2 :: *;
    initial begin
        $display("TRY1 : P1::a = %0d", a);
        $display("TRY1 : P2::b = %0d", b);
    end
endmodule : TRY1

```

The only difference between example 1 and example 2 is that in example 2 we import both P1 and P2. Since, P1 is explicitly imported in module TRY1, the variable “a” (of package P1) is now visible in module TRY1. Here is the simulation log.

*Simulation log:*

```

TRY1 : P1::a = 1
TRY1 : P2::b = 2
V C S   S i m u l a t i o n   R e p o r t

```

Now, you may be wondering why import a package, why cannot I just `include the file in my module. Will not it work the same way? Short answer is no. This has to do with SystemVerilog’s type system. A very good explanation is provided by (Rich, Dave). I will try to summarize it here.

Suppose you have two classes “A” and “B” as defined below:

```

class A;
    int i;
endclass : A

```

```
class B;
int i;
endclass : B
```

SystemVerilog considers these two class definitions unequal types because they have different names (“A” and “B”) even though the content of both the classes is the same. This is because the name of a class includes more than just the simple name “A” and “B.” The names also include the scope where the definition is declared. So, when you declare a class in a package, the package name becomes a prefix to the class name.

Let us now define two packages which contain the definition of class “A”:

```
package P;
    class A;
        int i;
    endclass : A
    A a1;
endpackage : P
```

and another package Q with the same class definition of “A”:

```
package Q;
    class A;
        int i;
    endclass : A
    A a1;
endpackage : Q
```

Note, as before, that there are two definitions of class “A.” With the package prefix added, they are P::A and Q::A and the variables P::a1 and Q::a1 are still type incompatible because they refer to two different class “A”’s.

If you rewrite the above code with `include, nothing really changes because `include is just a shortcut for cut/paste of text in the file where you use `include. So, here are the same packages as above but with `include.

Here is the class “A” definition in file A.sv:

```
class A;
int i;
endclass : A
```

And we `include this file in the two packages:

```
package P;
`include "A.sv"
A a1;
endpackage : P
```

```
package Q;
`include "A.sv"
A a1;
endpackage : Q
```

You will still get type incompatibility between P::a1 and Q::a1.

Now, let us use import instead of `include. First, the package “P” where the class “A” definition is declared:

```
package P;
  class A;
    int i;
  endclass : A
endpackage : P

package R;
import P::A;
A a1;
endpackage : R

package S;
import P::A;
A a1;
endpackage : S
```

The first thing to note here that there is *only one* definition of class “A” (in package “P”). In earlier, examples, we had two definitions of class “A,” one in each package. With this, now, the variables R::a1 and S::a1 are type compatible because they are both of type P::A.

That is the main difference between using `include vs. using “import.” The type compatibility issue is resolved using “import.” When you have type incompatibility, you need to consider the scope where the types are declared as part of the full name.

One final note on importing enumerated types. Note that importing an enumerated type definition does not import its enumerated labels. You have to explicitly import all the labels of the enumerated type (or use a wildcard import). For example, let us say you have the following package definition with an enum type:

```
package enumP;
  typedef enum logic [1:0] {SLOW, FAST, MEDIUM} speed;
endpackage
```

And take the following module:

```
module top;
  import enumP::speed;
  speed s1;
  always @(posedge clk)
  ...
  s1 <= FAST; //ERROR
  ...
endmodule
```

In module “top,” you import the enum “speed” and try to access the enum label “FAST.” But that is not possible. The enum name was imported but the enum labels were not. “FAST” is an enum label in enum “speed” and was not imported.

In order to solve this problem, you have to explicitly import the enum labels. You have to do the following:

```
import enumP::speed;
import enumP::SLOW;
import enumP::FAST;
import enumP::MEDIUM;
```

Another way to solve the problem is to use a wildcard import:

```
import enumP::*;


```

This will import the enum as well as its labels.

# Chapter 8

## Class



**Introduction** This chapter delves into the detail of SystemVerilog *class* which is the bain of object-oriented programming (OOP). It covers all aspects of a *class*, including polymorphism, inheritance, shallow copy/deep copy, parameterized classes, upcasting/downcasting, virtual methods, virtual classes, etc.

### 8.1 Basics

A *class* is the basis for object-oriented programming (OOP). OOP allows you to create verification environments (such as UVM – Universal Verification Methodology) that are at higher levels of abstraction and which are not encumbered with low-signal-level detail. A class facilitates such higher levels of abstractions. With traditional Verilog language, there was no way you can group variables and methods that work on them in a concise format. Only *reg* and *arrays* were available. That all changed with the introduction of classes in SystemVerilog.

A class is a user-defined type. It includes data/variables (known as class properties) and subroutines (known as class methods) that work on the class properties. The subroutines or methods are basically SystemVerilog tasks and functions. The class properties and methods, taken together, define the contents and capabilities of a class. Grouping data (properties) and functions/tasks (methods) in this way helps in creating and maintaining modular testbenches. As mentioned above, a testbench is much easier to code, debug, and simulate at a transaction level.

A class is a namespace and a scope. It contains members and methods, which can be time-consuming tasks or non-time-consuming functions. Class members are

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_8](https://doi.org/10.1007/978-3-030-71319-5_8)) contains supplementary material, which is available to authorized users.

visible to methods defined in the class. Also, a class has a constructor which is a special function that is called when an instance of the class is created. We will discuss all this in upcoming sections.

Classes are always constructed dynamically, which means that the memory for the class is not allocated until procedurally calling a routine that constructs a class object. Class deconstruction/de-allocation is automatic in SystemVerilog after an object has no references. In C++ it is the user's responsibility. SystemVerilog does not have the complex memory allocation and deallocation of C++. Construction of an object is straightforward; and garbage collection, as in Java, is implicit and automatic.

## 8.2 Base Class

Base class is the topmost class in a class hierarchy. You derive extended classes from a base class. We will study extended classes in the next sections. But first let us look at an example of a base class PCI which has command, address, data, CycleType, etc. properties. In addition, there are various things that can be done on these properties using class methods. Each PCI bus is different, but as a class, PCI has certain intrinsic properties that can be encapsulated in a definition. We can then have methods that work on PCI class properties. Note: you may want to refer to Sect. 8.3 on inheritance to see how a base class fits into an extended class and inheritance:

```
module class_TOP( );

class PCI;
    //Class properties
    logic [3:0] command;
    logic [31:0] address;
    logic [31:0] data;
    logic [3:0] CycleType;

    //base class constructor - initialization
    function new( );
        command = 0;
        address = 0;
        CycleType = 4'hf;
        data = 64'bz;
        $display("PCI Init: data=%h command=%b addr=%h
CycleType=%b", data, command, address, CycleType);
    endfunction

```

```

task PCIWriteCycle (clk);
begin
    command = $urandom;
    address = $urandom;
    CycleType = $urandom;
    $display("PCI Write Cycle : clk=%b data=%h command=%b
addr=%h CycleType=%b", clk, data, command, address, CycleType);
end
endtask
endclass : PCI

bit clock;

PCI PCI1; //define variable PCI1 of type PCI

initial begin
    PCI1 = new( ); //instantiate class - allocate memory
                    //PCI1 now holds an object handle.
end

initial begin
    clock = 0;
    forever begin
        #10; clock=!clock;
    end
end

always @(posedge clock) begin
    //access class property using object handle PCI1
    PCI1.data = $urandom;

    //Call Class method PCIWriteCycle
    PCI1.PCIWriteCycle(clock);
end

initial #60 $finish(2);
endmodule

```

Let us dissect this code.

First, we declare a class called PCI. In this class, we declare class properties (data/variables). These properties are now available to class methods. There are no restrictions on the data type of a class property:

```

logic [3:0] command;
logic [31:0] address;
logic [31:0] data, mem;
logic [3:0] CycleType;

```

We then declare what is known as a class constructor.

The new( ) operation is defined as a function without a specified return type (actually, it does have a return type – the class type, it is just implicit). Function new() is called upon object creation (i.e., when the class is instantiated – you have to procedurally call new() explicitly). And when it is called, the functionality in this function will be used to initialize the class properties.

Note that you do not have to have a class constructor. It is not mandatory. If you do not declare one, an implicit new ( ) function will be called automatically, when the class is instantiated. In this case, the variables will be initialized to their default values. Zero for 2-state variables and ‘x’ for 4-state variables.

Here is the base class constructor:

```

function new();
    command = 0;
    address = 0;
    CycleType = 4'hf;
    data = 64'bz;
    $display("PCI Init: data=%h command=%b addr=%h
CycleType=%b", data, command, address, CycleType);
endfunction

```

In this constructor, we assign values to the class properties and display them. As mentioned before, this will be the first function called when this class is instantiated. The constructor is called automatically for you.

Note: SystemVerilog specifies that variables are automatically initialized to some valid value – e.g., “” for strings and 0 for 2-state variables and ‘x’ for 2-state variables. So, you typically don’t have to initialize things. However, it is a good practice to initialize things. It explicitly documents your expectations of the initial value.

We then define a method called PCIWriteCycle. This method will work on the properties of the class. This is a regular SystemVerilog “task.” You can also have functions as class methods (for an understanding of tasks and functions, refer to Chap. 22). In this method, we simply assign random values to class properties to initiate a PCI Write Cycle:

```

task PCIWriteCycle (clk);
begin
    command = $urandom;
    address = $urandom;
    CycleType = $urandom;

```

```

$display("PCI Write Cycle : clk=%b data=%h command=%b
addr=%h CycleType=%b", clk, data, command, address, CycleType);
end
endtask

```

That is it. That is all we have in this class. To recap, we declared class properties, we created a class constructor, and then we created a class method to work on class properties.

Now, after the class declaration, we declare a variable PCI1 of class type PCI. Note that at this time, all we have done is declare a variable and not allocate any memory for the class:

```
PCI PCI1; //define variable PCI1 of type PCI
```

Then, we instantiate the class (i.e., create an object of class PCI) using the new( ) function. This is when memory is allocated for the object PCI. This is also where the class initialization takes place:

```
PCI1 = new( ); //instantiate class - allocate memory
```

*Important Note: The variable PCI1 is said to hold an object handle of an object of class PCI. Do not confuse class variable with object handle (or class handle). You first define a variable of some type of class. Then, when you instantiate the class, the variable now holds the object handle (or stores the object handle).* Note that without instantiating the class, the class variable is said to hold a “null” handle. Note that a class variable may point to (or hold the handle of) different objects during its lifetime. SystemVerilog objects (instances) are referenced using object handles. Class instances, or objects, can be passed around via object handles.

Note that the terms class handle and object handle are interchangeably used in literature. They mean the same thing. I will use the term object handle in the ensuing discussion. Typically, the term “class” refers to the un-instantiated declaration, and the term “object” refers to an allocated instance. So, the term “object handle” is more precise and consistent with the notion that a class is un-instantiated, and an object is an instance.

In addition, you may find in many literatures that often the class variables are themselves referred to as object handles. But I like to make the distinction between a class variable and an object handle as noted above.

When you instantiate the class, it will call the class constructor function new( ) that was declared in class PCI. This function is used to initialize class properties as we have done in class PCI.

Then, we use the object handle PCI1 to invoke the task PCIWriteCycle of class PCI. Note that you can also access class properties using the object handle, as we have done here:

```

    always @(posedge clock) begin
        //access class property using object handle PCI1
        PCI1.data = $urandom;

        //Call Class method PCIWriteCycle
        PCI1.PCIWriteCycle(clock);
    end

```

The simulation log shows that first, when the object PCI was constructed using new( ), the \$display in the function new( ) is displayed. Then, every time we called the method PCIWriteCycle, the randomized data are shown in its display.

*Simulation log:*

```

PCI Init: data=zzzzzzz command=0000 addr=00000000 CycleType=1111
PCI Write Cycle : clk=1 data=12153524 command=0001 addr=8484d609
                  CycleType=0011
PCI Write Cycle : clk=1 data=06b97b0d command=1101 addr=b2c28465
                  CycleType=0010
PCI Write Cycle : clk=1 data=00f3e301 command=1101 addr=3b23f176
                  CycleType=1101
$finish called from file "testbench.sv", line 52.
$finish at simulation time          60
                                V C S   S i m u l a t i o n   R e p o r t

```

One more note on object handles. There are some differences between a C pointer and a SystemVerilog object handle. C pointers are a lot more flexible in what you can do with them, for example, you can increment a C pointer, but you cannot do that with a SystemVerilog object handle.

To recap, here is the terminology used in our discussion above. This will be useful throughout the discussion on classes:

- **Class:** A basic building block containing class properties and methods.

When you declare a class, you are declaring a set of properties and a set of methods that operate on those properties:

```

class MyClass;
    bit [7:0] member1;
    bit member2;
    function void method;
        $display("members are %h %b", member1, member2);
    endfunction
endclass

```

Consider this a user-defined type, like a **typedef**. We are declaring the form and behavior of a type, but nothing is allocated to store the values of this type:

- **Class Object:** An instance of the class. A class must be instantiated before it can be used. Class object is an instance of a class definition that you create when you call its constructor function – new( ). All class objects are created dynamically, on or after time 0.
- A class object is a particular instance of a class type. Each instance is an allocation of the members defined by the class type. The only way to create an object is to call the class constructor using the built-in new( ) method of a class. We classify class objects as dynamic objects because executing a procedural statement is the only way to create them.
- **Class Variables**
- A class variable is where you store the object handle that references a particular class object of a particular class type. Declaring a class variable does not create a class object, only the space to hold the object handles. This contrasts with other data types where the declaration of a variable creates an object of that type and gives you a symbolic name to reference those objects. For example:

```
typedef struct {bit [7:0] member1; bit member2;} MyStruct;
MyStruct StructVar1, StrucVar2
```

This creates and allocates the space for two MyStruct-type objects, and you can use StructVar1.member1 to access one of its members. On the other hand:

```
MyClass ClassVar1, ClassVar2;
```

This creates and allocates the space for two MyClass variables but only allocates the space to hold handles to MyClass objects, not the objects themselves. If you tried to access ClassVar1.member1, you would get a null handle reference error because the initial value of a class variable is the special value null. One of the nice things about handles instead of pointers is that they remove the possibility of corrupt object references that access uninitialized or de-allocated memory.

- **Object Handle:** When you instantiate a class, the class variable is then said to hold the object handle.
- Once you have a class variable, you can call the new( ) method to construct a class object:

```
ClassVar1 = new();
```

This calls the constructor of the MyClass type that returns a handle to a MyClass object and then stores that handle in the MyClass variable, ClassVar1. You can now access ClassVar1.member1 because ClassVar1 references an actual object.

- **Class Property:** Class member (variable) declared in a class. Class members are visible to methods defined in the class.
- **Class Method:** A member function or task (procedural code) that works on class properties.

### 8.3 Extended Class and Inheritance

Inheritance is a backbone of class-based methodology/OOP (object-oriented programming). An extended class is an extension of a base class (or a super class). The extended class *inherits* all the properties and methods of the base class. When you construct an extended class type, you are constructing a single object that has all the properties of the inherited types.

Both the properties and methods of the base class can be overridden (so to say) in an extended (derived) class, or you can add more properties and methods in the extended class. When you override a property or method in an extended class, it hides that property or method of the base class. It does *not* really override it because the original property or method of the base class is still available to other classes extended from the base class. This allows developers to hierarchically build a class-based structure where one needs not repeat what is already been defined in the base class, just improve on it.

For example, a base class called EthernetPacket can be defined. Then extension classes such as EtherErrorPacket, EtherLatencyPacket, etc. can be defined which has the properties of the base class and add to those new properties and methods as required by the extension class. One of the advantages of inheritance is that if you make a change in the base class, it gets propagated to all the extended (derived) classes. Note that you can further extend an extended class.

Here is an example:

```
module class_TOP( );

class base;
    logic [31:0] data1;
    logic [31:0] data2;
    logic [31:0] busx;

    function void bus;
        busx = data1 | data2;
    endfunction
endclass
```

```
function void disp;
    $display("From Base Class :: busx OR = %h", busx);
endfunction
endclass : base

class ext extends base;

logic [31:0] data3; //add a new property

function new ();
    $display("Call base class method from extended class - super.disp");
    super.disp;
endfunction

function void bus; //redefine function 'bus' of class 'base'
    busx = data1 & data2 & data3;
    //data1,data2 inherited from class 'base'
endfunction

function void disp; //redefine function 'disp' of
class 'base'
    $display("From Extended Class :: busx AND = %h",busx);
endfunction
endclass : ext

initial begin
    base b1;
    ext e1;

    b1 = new;
    e1 = new;

    b1.data1 = 32'h ffff_0000;
    b1.data2 = 32'h 0000_ffff;
    b1.bus;
    b1.disp();

    e1.data1 = 32'h 0101_1111;
    e1.data2 = 32'h 1111_ffff;
    e1.data3 = 32'h 1010_1010;
    e1.bus;
    e1.disp();
end
endmodule
```

*Simulation log:*

Call base class method from extended class - super.disp

From Base Class :: busx OR = xxxxxxxx

From Base Class :: busx OR = fffffff

From Extended Class :: busx AND = 00001010

V C S   S i m u l a t i o n   R e p o r t

In the class “base,” we define data1, data2, and busx properties. We also define two functions “bus” and “disp.” In the function “bus” in class ‘base’, we do an OR of data1 and data2 and display the resulting ‘busx’ through function disp. Note that there is no new () function in the base class. So, the variables data1, data2, and busx will be initialized to their default values (which is “x” since all three are of type “logic”).

The class “ext” is the extended version of “base” (“class ext extends base”). Since this is an extended version of “base,” the properties “data1,” “data2,” and “busx” are visible/available to class “ext.” We also declare a new property “data3” in the extended class “ext.” We then override the function “bus” and “disp” in the extended class “ext.” We declare function “bus” in extended class and override the “l” (or) function of “base” class with an “&” function. We also override the function “disp” of class “base” in the extended class “ext.”

Note that we can call a base class method from the extended class using the keyword “super.” In our example, in the extended (ext) class constructor (function new ()), we call “super.disp” to first display the value of “busx” which is “xxxxxxxx” because none of the properties of the base class has been initialized. The new () function of “ext” class is called when it is instantiated (ext e1; e1 = new).

In the initial block, we first change the properties of “b1” instance of “base” class and display “busx.” We see in the simulation log the result of the “l” (or) equation. Then, we change the properties of the extended class (which are properties of class “base” visible to class “ext” and the newly added property “data3”). And use the overridden function “disp” of class “ext” to display the results of “&” operation. Results are evident in the simulation log.

To recap, we saw in this example:

- New classes can be created based on existing class (inheritance).
- The properties of base class are available/inherited in the extended class.
- The inherited class is also called a subclass of the base (a.k.a. parent or super) class.
- The extended class may add new properties and methods or simply modify the inherited methods and/or properties. It is not possible to change the characteristics (e.g., data type) of a base class member, though the derived class can change the value, of course.
- The methods of base class can be overridden in the extended class.
- The properties of base class can be redefined in the extended class (which means that it hides the base class properties).

- Overriding a method of base class in the extended class *hides* the method of the base class. You can still access the base class method using the prefix “super” (e.g., “super.<method>”).
- SystemVerilog allows only single inheritance, meaning a subclass can extend from only one parent class. In general, SystemVerilog does not support multiple inheritance. However, note that SystemVerilog supports a limited form of multiple inheritance using interface classes.

As noted, you can use “super” keyword to access any of the methods in the base class. When you use “super,” you are accessing a member from the perspective of the class you are extended from (from the parent or the base class). You can do super.<method> (i.e., go one level up), but *you cannot do super.super.<method>* (cannot go multi-level up).

As we saw, each class has a constructor called “new( )” (implicit or explicit). Similarly, each extended class also has a constructor new( ). In the new( ) method of the extended class, the first thing (actually, the very first statement) is a call to the base class new( ) method, using the “super” keyword. So, the new( ) constructor of the extended class will first call “super.new( )” to call the constructor of the base class. This call is implicitly added by SystemVerilog or you can define your own. If you do not provide a super.new of your own in the extended class, SystemVerilog will insert one for you. We will discuss “super.new( )” further under the section on chain constructor.

Here is the same example as above, but we are calling the function “disp” from extended class using the *super* keyword from a method of the extended class. In the previous example, we called “super.disp” from the constructor of the extended class. Point being “super.<method>” can be called from outside the constructor as well:

```

class base;
    logic [31:0] data1;
    logic [31:0] data2;
    logic [31:0] busx;

    function void bus;
        busx = data1 | data2;
    endfunction

    function void disp ( );
        $display("From Base class :: busx = %h",busx);
    endfunction
endclass : base

class ext extends base;
    function void bus; //redefine function 'bus' of class 'base'

```

```

busx = data1 & data2; //data1,data2 inherited from
class 'base'
endfunction

function void disp; //redefine function 'disp' of
class 'base'
    super.disp(); //call parent class method
endfunction
endclass : ext

module class_TOP( );
initial begin
    base b1;
    ext e1;

    b1 = new;
    e1 = new;

    b1.data1 = 32'h ffff_0000;
    b1.data2 = 32'h 0000_ffff;
    b1.bus;
    b1.disp();

    e1.data1 = 32'h ffff_0000;
    e1.data2 = 32'h 0000_ffff;
    e1.bus;
    e1.disp();
end
endmodule

```

*Simulation log:*

From Base class :: busx = ffffffff

From Base class :: busx = 00000000

V C S   S i m u l a t i o n   R e p o r t

When you call “e1.disp ()”, you are essentially calling the “disp ()” function of the base class. This is because the “disp” function of class “ext” calls “super.disp ()” which is a call to “disp ()” function of the class “base.”

You can also create a hierarchy of extended classes. We will see such extended hierarchy in future examples. For example:

```

class packet;
logic [31:0] addr;
logic [31:0] data;
function new (...);

```

```

...
endclass

class ethpacket extends packet;
.....
function new(...);
    super.new(...);
    /* Note: Calling super.new() is not necessary for the
default constructor - i.e., no constructor arguments. An explicit
call to super.new() is required for non-default constructors,
i.e., constructors that contain function arguments. Discussed fur-
ther in coming sections */
.....
endfunction

...
endclass

class tokenpacket extends ethpacket;
.....
function new(...);
    super.new(...);
.....
endfunction
.....
endclass

```

In this example, we extend “ethpacket” from the base class “packet” and then extend “tokenpacket” from class “ethpacket.” Each extended class will call the super.new of base class, and it will all cascade up to the base class. But note that you can only cascade up one level. You cannot do super.super.new. Also, note that super. new will be the very first function (very first statement) called from the new () function of an extended class.

### ***8.3.1 Inheritance Memory Allocation***

So, we understand how extended class inherits properties and methods of the base class. Let us now understand how exactly the memory gets allocated for base class and the extended class. This is a simple concept but will be very useful, when we see advanced concepts.

Let us say we have the following code:

```

module class_TOP( );

    class aa;
        int i1;
        function funAA;
        endfunction
    endclass : aa

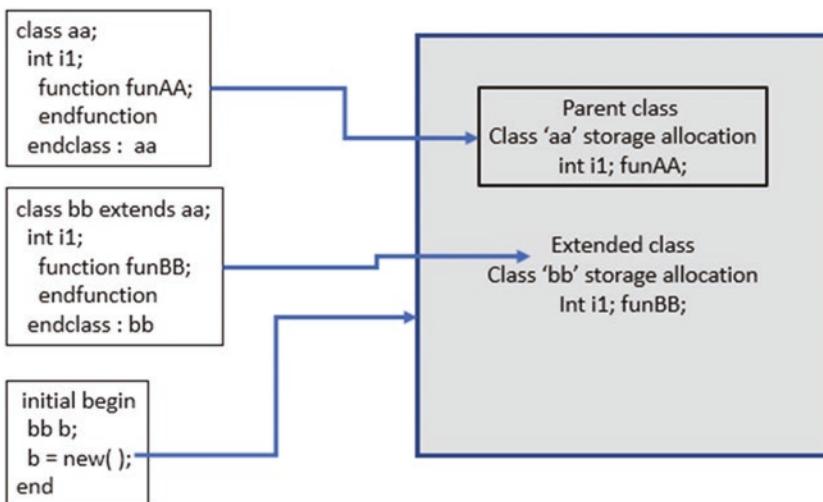
    class bb extends aa;
        int i1;
        function funBB;
        endfunction
    endclass : bb

    initial begin
        bb b;
        b = new( );
    end
endmodule

```

Here is the graphical representation of the code (Fig. 8.1).

Here, when we do “`b = new( )`,” one memory allocation takes place but with two views (because class `bb` includes the code from class `aa`). First, the memory allocation for class “`aa`” where memory for “`int i1`” and function `funAA` gets allocated. After that, the memory for class “`bb`” gets allocated for its properties and methods.



**Fig. 8.1** Inheritance memory allocation

In other words, “`b=new( )`” did not just allocate memory for the class “`bb`” but also for its parent class “`aa`.” As simple as it looks, this is one of the most important concepts to understand: how memory is allocated with the `new( )` function of an extended class. This will be useful when we learn about upcasting and downcasting.

## 8.4 Class Constructor

### 8.4.1 Base Class Constructor

We saw in Sect. 8.2, what a base class constructor does. We saw that if a class does not provide an explicit user-defined constructor (`new( )` method), an implicit `new( )` method is provided automatically.

The base class constructor initializes the class variables to explicit default values, or if no explicit values are provided, the variables will take on the uninitialized value, “0” for 2-state variables and “X” for 4-state variables. The default constructor has no additional effect after its variable (property) initialization. *One note on constructor function new( ) that it cannot be declared either as “virtual” or “static,” but it can indeed be declared as “local” or “protected.”* Also, the constructor function `new( )` has no return type. Even the “void” return type is not allowed. More on all this in sections throughout the chapter.

### 8.4.2 Chain Constructor (`super.new( )`)

*The new( ) method of a derived class (extended class) will first call its base class constructor (`super.new( )`).*

Here is an example of how `super.new( )` works:

```
module class_TOP( );
    class base;
        function new( );
            $display($stime," new( ) from BASE Class");
        endfunction
    endclass : base

    class ext extends base;
        function new;

```

```

super.new( ); //MUST be the first statement

$display($stime,,, "Call super.new from Extended
class");
endfunction
endclass: ext

base base1;
ext ext1;

initial begin;
  ext1 = new( );
end

initial #1 $finish(2);
endmodule

```

*Simulation log:*

0 new( ) from BASE Class  
 0 Call super.new from Extended class

\$finish at simulation time 1  
 V C S   S i m u l a t i o n   R e p o r t

This is a simple example to show the order of execution (of new( )) in base class and extended class. In class “base,” we define its constructor new( ) which has a simple \$display statement. Then we extend the “base” class to “ext” class. In “ext” class constructor new( ), we first explicitly call super.new( ) which will call the new( ) of the “base” class. Then, we instantiate “ext” class with instance name “ext1.” Note that we are only instantiating “ext” and not “base.”

When “ext” is instantiated, its new( ) method will first call super.new( ) which will execute new( ) of the “base” class. *Note that the new( ) function in class “ext” must first call super.new( ) before any other statement. super.new( ) must be the first statement in class new( ) of “ext” class.* In the simulation log, we see that the \$display from “base” class is executed first before the \$display from “ext” class. This is the order of execution; super.new( ) calls new( ) of “base” class and then executes any statements that follow super.new( ).

Note also that if you do not explicitly call super.new from an extended class, it will be implicitly called from the extended class' new( ) function. So, for example, if you remove “super.new” call from the constructor of the class “ext” in the above example, you will still get the same results.

SystemVerilog will do the correct job as long as you do not have arguments to the new( ) method. This is because when the simulator implicitly calls the super.new function, it would not know that the base class new( ) method has arguments. So, it

is highly recommended that you explicitly call super.new to make it less error prone and for readability:

Here is an example.

```
module class_TOP( );

class base;
    logic [31:0] data;

    function new(logic [31:0] dataIn);
        data = dataIn;
        $display($stime,,, "new( ) from BASE Class: data=%h",
                 data);
    endfunction
endclass : base

class ext extends base;

    function new (logic [31:0] dataExt);

        super.new(dataExt); //MUST call super.new
                            //MUST be the first statement
                            //pass dataExt to the base class constructor

        $display($stime,,, "super.new from Extended class");
    endfunction

endclass: ext

base basel;
ext ext1;
logic [31:0] data;

initial begin;
    data = 'hf0f0_f0f0;
    ext1 = new(data);
end

initial #10 $finish(2);
```

```
endmodule
```

*Simulation log:*

```
0 new( ) from BASE Class: data=f0f0f0f0
0 super.new from Extended class
$finish at simulation time          10
V C S   S i m u l a t i o n   R e p o r t
```

This example is identical to the one before it. But in this example, we pass data from the extended class' new( ) function to the base class' new( ) function via the super.new( ) call from the extended class. We pass this data to the extended class during its instantiation ("ext1 = new (data);"). As evident from the simulation log, the *new( ) function of the base class is executed first* and shows that data were passed from extended class to the base class via the super.new( ) function call.

Note that since the "new" function of the base class has an argument, you *must* call super.new( ) from the constructor of the extended class. If you don't, you'll get the following compile error (Synopsys – VCS):

```
Error-[SV-SNCIM] Super new call is missing
testbench.sv, 13
class_TOP
Base class constructor specification is missing in derived class 'ext'.
Please add <base class>>::new arguments to 'extends' specification or add
call to 'super.new' in <derived class>::new.
```

## 8.5 Static Properties

In all the prior examples, we saw that each class instance has its own copy of its variables. These were dynamic properties. But there are times when you need to maintain only one copy of the variable shared by all instances. Such class properties/variables are created using the keyword *static*. Static properties are allocated in memory at elaboration time and shared across all instances of the class. Static variable acts like a global variable in that only one copy exists, no matter how many objects you create. For a static variable, the storage is associated with the class declaration, not the multiple objects that are constructed dynamically.

A good example is when you want to share a FileID among all instances of a class. You want to use the same file for reading from different instances of a class. For example, (SystemVerilog – LRM):

```
class fileIO;
    static integer fileID = $fopen ("data", "r");
```

Since fileID is static, it will be created and initialized once before time 0 and then every instance of class "fileIO" can access the fileID in the usual way, as in the following:

```
fileIO f1;
c = $fgetc (f1.fileID);
```

Note: Functions called to initialize static variables are called at static initialization time, which is before time 0 – i.e., between elaboration and when the simulation starts running, something to be aware of when calling function to initialize static variables.

Now, let us see how this works in practice. Here is an example, where a property is declared static in a class and then accessed from different instances of that class – each one seeing an identical copy of the static property:

```
module class_TOP( );

class base;
    static logic [31:0] data; //static property
endclass : base

base base1, base2;

initial begin;
    base1 = new();
    base2 = new();
    base1.data = 32'h ff00_f0f0;
    $display("base2.data = %h",base2.data);
end

initial #10 $finish(2);
endmodule
```

In this example, in class base, we declare “data” as static property. Then we instantiate “base” twice as “base1” and “base2.” We then set the value of “data” in the “base1” instance and access it from “base2” instance. Here is the simulation log:

*Simulation Log:*

```
base2.data = ff00f0f0
$finish at simulation time          10
V C S   S i m u l a t i o n   R e p o r t
```

As evident from the simulation log, the value set with object handle “base1” is available when accessed by “base2.” This is because “data” is declared static and available to all instances of class “base.” What would happen if you removed the keyword *static* in “data” declaration? You will get “XXXXXXXX” when “data” is accessed from “base2” since non-static “data” is available only to instance “base1.”

Here is another example showing how a static variable increments from one invocation of the new () function to the next:

```

module class_TOP( );

class packet;
    static int packet_cnt;

    function new( );
        packet_cnt = packet_cnt+1;
        $display("From new ( ) :: packet_cnt = %0d",packet_cnt);
    endfunction

endclass

initial begin
    packet p1 = new;
    packet p2 = new;
end
endmodule

```

*Simulation log:*

From new () :: packet\_cnt = 1

From new () :: packet\_cnt = 2

V C S   S i m u l a t i o n   R e p o r t

When we display the values of packet\_cnt from the new () function, you see the incremented valued of packet\_cnt, since each increment builds on the previous value that was statically stored.

Here's another example, showing the difference between a static and a non-static variable and how the static variable increments with each instantiation of a class (since it is shared among all instances), while the non-static variable (which is not shared among all instances) does not increment with each instantiation of the class. “data” is a dynamic property which means each instance of the class has its own separate copy:

```

class PCIe;
    static int addr = 0; //Static property
    int data = 0; //Dynamic property

    function new ();
        addr++;
        data++;
    $display ("addr=%0d data=%0d", addr, data);
    endfunction

```

```
endclass

module tb;
    initial begin
        PCIE p1, p2, p3;
        p1 = new ();
        p2 = new ();
        p3 = new ();
    end
endmodule
```

*Simulation log:*

```
addr=1 data=1
addr=2 data=1
addr=3 data=1
```

V C S   S i m u l a t i o n   R e p o r t

Finally, here is an example showing how static properties and dynamic properties work in a class:

```
module class_TOP( );

class packet;
    static int packet_cnt; //Static property
    int tag; //Dynamic property

    function new( );
        packet_cnt = packet_cnt+1;
        tag = packet_cnt;
    endfunction

    function void disp( );
        $display("packet_cnt = %0d tag = %0d",packet_cnt, tag);
    endfunction

endclass

initial begin
    packet p1 = new;
    packet p2 = new;

    p1.disp;
    p2.disp;
```

```
    end
endmodule
```

*Simulation log:*

```
packet_cnt = 2 tag = 1
packet_cnt = 2 tag = 2
```

The static property “packet\_cnt” incremented each time “packet” was instantiated. So, when you do “p1.disp” or “p2.disp,” you get the *final* value of “packet\_cnt” because it is shared by both instances. Can you figure out why “tag” value shows increments? Keep in mind that “tag” is a dynamic property.

*Exercise:* Display the values of “packet\_cnt” and “tag” from the function new( ) and see the difference in results.

## 8.6 Static Methods

Methods can also be declared “static.” Static method is a method whose scope is static meaning static function remains the same for *all* the objects of a class. In system memory, the method is stored at a single place, and all the objects access that method memory. Static method can be accessed outside a class – even with no instantiation of the class in which the static method is declared. That is because it is a method of class type, not of an instance of a class object.

*The static method can only access static properties.* Access to non-static properties from a static method will result in a compiler error. Also, static methods cannot be virtual. You cannot use the “this” handle (discussed in the next section) with a static method.

Here is an example showing usage of a static method:

```
module class_TOP( );
  class base;
    static logic [31:0] data ; //static property
    logic [31:0] addr; //dynamic property

    static task munge; //Static method
    data = 32'h f0f0_f0f0; //OK to access static variable
    //addr = 32'h ffff_0000; //NOT OK since 'addr' is
not static
    $display("data = %h", data);
  endtask
endmodule
```

```

endclass : base

base base1;

initial begin
    base1.munge;
end

initial #10 $finish(2);
endmodule

```

*Simulation log:*

```

data = f0f0f0f0
$finish at simulation time          10
V C S   S i m u l a t i o n   R e p o r t

```

In this example, we declare a static method called “munge” in class “base.” We also declare a variable “base1” of class type “base.” Note that we are – not – instantiating “base1.” The first thing to note is that we are accessing “munge” using only the class variable “base1” – not – its object handle. This is because “munge” is declared static. Here is the simulation log, which shows the execution of “munge” when accessed using the class variable “base1” (“base1.munge”).

Note also that the only variable available to static task “munge” is the static variable “data.” If you uncomment the following line (i.e., access non-static “addr” from static method munge), you will get a compile error. That is because the “addr” property only exists in individual objects, so task munge could not see it:

```
addr = 32'h ffff_0000; //NOT OK since 'addr' is not static
```

Here is the compiler error from Synopsys – VCS:

```

Error-[SV-AMC] Non-static member access
testbench.sv, 9
class_TOP, "addr"
Illegal access of non-static member 'addr' from static method 'base::munge'.

```

You can also access a static method with the class scope resolution operator. That is because “base1.munge” is accessing the static method from a null object handle (base1). So, in the above example, we could have done the following to access static method “munge.” I prefer this way of calling a static method, since it clearly identifies the class, and we know explicitly that it is a static method. Static fields/methods do not “belong” to objects. For static methods it is highly recommended to use the class scope resolution operator:

```
base::munge;
```

Note that static methods cannot be virtual. So, if you did the following, you would get a compile error:

```
class base;
    virtual static task munge();
endtask
endclass
```

You will get the following error (Synopsys – VCS):

Error-[WUCIQ] Invalid qualifier usage  
testbench.sv, 32

Invalid use of class item qualifiers. Cannot use virtual and static keywords together for method declarations.

Following example is simply to show that you can access, using class resolution operator, static method, and static variables without instantiating the class in which these methods/variables are declared:

```
class setIt;
    static int      k;

    static function set (int p );
        k = p + 100;
    endfunction
endclass

module tbTop;
    initial begin

        setIt::set(10);
        $display("k = %0d",setIt::k);

        setIt::set(20);
        $display("k = %0d",setIt::k);
    end
endmodule
```

*Simulation log:*

```
k = 110
k = 120
```

V C S S i m u l a t i o n R e p o r t

Finally, what we have seen is *static function* (or task) in the above examples. But there is also the concept of *function static*. The difference between “static function” and “function static” is that the former is the static function of class and exists irrespective of the class instances, while the “function static” means variables/parameters of the function are static in nature and not automatic.

For example:

```

class statA;
    task static incr();
        int unsigned stati;
        $display("stati is %d", stati);
        stati++;
    endtask
endclass

module top;
    statA a;
    statA b;

    initial begin
        a = new();
        b = new();

        a.incr();
        b.incr();
        a.incr();
        b.incr();
        a.incr();
    end
endmodule : top

```

#### *Simulation log:*

```

# vsim -voptargs+=acc=npr
# run -all
# stati is      0
# stati is      1
# stati is      2
# stati is      3
# stati is      4
# exit

```

Class “statA” is not static but is declared as “task static” meaning; it is a non-static class with static lifetime of its properties. Since “stati” is now static, it will be shared among all instances of “statA.” Hence, the value increases with each call to “incr.”

## 8.7 “this”

When you have multiple properties or methods in the same class (with different scopes) and with the same names, you need to unambiguously refer to these properties and methods of the current instance. That is what the *this* keyword does. *The “this” keyword denotes a predefined object handle.* The “this” keyword can be used within non-static methods, constraints, and covergroups embedded within the class. The “this” keyword is used to refer to class properties, parameters, and methods of the current instance.

Also, the *this* handle is useful anytime a class instance (an object) needs to refer to itself. We will see an example of this further down.

Adding “this” explicitly is what disambiguates it from the local named variable.

An example of simple disambiguation (but to be honest, *it is a bad programming practice to have variables with the same name hiding the visibility of other variables*). But this example is just to get the point across:

```
class thisClass;
integer tint;
function new (integer tint);
    this.tint = tint; //If you did
'tint=tint' - which
    //'tint' gets assigned?
endfunction
endclass
```

Here, we have two variables with the same name “tint.” One is the variable of the class “thisClass,” while other is the argument of the constructor function `new()`. In order to distinguish between the two, you need to use the keyword *this* (`this.tint`) to refer to the variable “tint” of the class `thisClass`. When you do “`this.tint = tint;`” you are assigning the argument “tint” of the class constructor to the “`thisClass`” variable “tint.” If you simply did “`tint=tint`” in the function `new()`, the compiler would not be able to distinguish between the two “tint”s, and you will get incorrect results (will not get compile/run-time error).

Here is a working example on disambiguation (again, *it is a bad programming practice to have variables with the same name hiding the visibility of other variables*). So, this example is just to show how disambiguation works. Not recommended coding style:

```
class pci;
bit [31:0] abus;
bit [31:0] dbus;

//constructor
function new(bit [31:0] abus, dbus);
```

```

        this.abus = abus + 'hff;
        this.dbus = dbus + 'hf0;
        $display("this.abus = %0h abus = %0h",this.abus,abus);
        $display("this.dbus = %0h dbus = %0h",this.dbus,dbus);
    endfunction
endclass

module PCIMod;
    pci p1;

    initial begin
        p1 = new(32'hF0F0_F0F0,32'hFFFF_0000);
    end
endmodule

```

*Simulation log:*

```

this.abus = f0f0f1ef abus = f0f0f0f0
this.dbus = ffff00f0 dbus = ffff0000

```

### V C S   S i m u l a t i o n   R e p o r t

In this example, we declare a class “pci” with two properties “abus” and “dbus.” The constructor (function new ( )) accepts two arguments, also called “abus” and “dbus.” But to disambiguate between “abus” and “dbus” of the class vs. that of the function new( ), we use the keyword “this” for the “abus” and “dbus” properties of the class. We display both the “this.abus” and “this.dbus” of the class properties and “abus” and “dbus” of the function arguments.

From the module “PCIMod,” we instantiate “pci,” which calls the new( ) constructor of the class “pci.” In the constructor, we modify the arguments passed to function new( ) and assign them to class properties. Since “this” was used with the class properties, you can see from the simulation log that the modified values are reflected on the “this.abus” and “this.dbus,” but the original function arguments passed on “abus” and “dbus” are displayed.

Note: The *this* is an implicit argument to all class methods. It represents a handle to specific object instance the method was called on (Rich, Dave). So, when you call:

```
handle.method(.arg1(expr));
```

The compiler transforms the call to:

```
className::method(.arg1(expr), .this(handle));
```

and the handle gets copied to the *this* argument and is usable inside the method.

Also, as mentioned before, the *this* handle is useful anytime a class instance (an object) needs to refer to itself. Here’s a snippet of code to highlight what this means:

```

class PCIBus;
    PCIRead current;
    ...
    function PCIRead item;
        current = new(this); // 'this' is a 'reference to myself'
        if( !current.randomize() ) ...
        ...
        return current;
    endfunction : item

class PCIRead;
    const PCIBus owner;
    ...
    function new(PCIBus owner = null);
        this.owner = owner;
    ...

```

So, what does `current = new (this)` mean?

“this” is a reference to the object on which the method that contains that line of code is executing. In other words, `current = new (this)` is a line of code in a method (function PCIRead) in the class PCIBus. So, “this” points to an object of class PCIBus. And the input to the constructor (of class PCIRead) is a reference to an object of class PCIBus. We have input a reference to an object of class PCIBus to the constructor (of class PCIRead). So, `current = new(this)` constructs an object of class PCIRead and inputs a reference to an object of class PCIBus (“this”) to it.

And, `this.owner = owner` is to disambiguate “owner” variable of class PCIRead from the function argument “owner.” `this.owner` accesses the property of class PCIRead.

The “this” comes in very handy when you need to link objects together in a data structure, like a tree. *The “this” handle is an implicit argument to all non-static methods including the constructor.* It represents the object that the method is being called on. When you reference a class property name, you are actually referencing an implicit “`this.name`.”

For example, (Rich, Dave):

```

class element;
    element children[$];
    string name;
    function new(string name, element parent);
        this.name = name;
        if (parent != null) parent.children.push_back(this);
    endfunction
    function void printer(int level);
        $display(level,, name);

```

```
foreach(children[c]) children[c].printer(level+1);
endfunction
endclass

class bottom extends element;
    function new(string name, element parent);
        super.new(name, parent);
    endfunction
endclass

class middle extends element;
bottom b1,b2;
    function new(string name, element parent);
        super.new(name, parent);
        b1 = new("b1",this);
        b2 = new("b2",this);
    endfunction
endclass

class top extends element;
middle m1,m2;
    function new;
        super.new("Top", null);
        m1 = new("m1",this);
        m2 = new("m2",this);
    endfunction
endclass

module test;
    top Top;
    initial begin
        Top = new ();
        Top.printer(0);
    end
endmodule
```

*Simulation log:*

```
0 Top
1 m1
2 b1
2 b2
1 m2
2 b1
2 b2
V C S  S i m u l a t i o n  R e p o r t
```

## 8.8 Class Assignment

So far we have seen how objects are created using `new()`. When you simply declare a variable of a class type, memory is not allocated. Then, when you instantiate the class (`new()`), an object handle is created which is stored in the variable that you just created. So, once the class is instantiated, the variable is said to store the object handle, meaning the variable, as object handle, points to where the newly instantiated object is stored in memory. Keeping this in mind, let us see how class assignment works:

```
class PCI;
.....
endclass
PCI p1; //a variable 'p1' of type PCI is created.
         //Memory is not allocated.
p1 = new; //this is where memory is allocated for 'p1'.
```

On that thought, we can do the following:

```
PCI p1;
p1 = new;
PCI p2;
p2 = p1; //class assignment
```

So, `p2` is just a variable of type `PCI`, while `p1` is an object. When you assign `p2 = p1`, there is still just one object (`p1`), and both “`p1`” and “`p2`” variables will now refer to the same object (i.e., the same memory location). Since they are both pointing to the same memory location, a change in “`p1`” will reflect on “`p2`” and vice-versa. Figure 8.2 illustrates the concept. In the figure, we see that “`p1`” and “`p2`” are two variables of type class `PCI`. We then instantiate `p1` (`p1 = new;`). And assign `p2 = p1`. From the figure you can see that both “`p1`” and “`p2`” now reference to the same location in memory. Then when you make a change using the handle “`p1`”, that change is reflected in “`p2`” as well (or vice versa).

Here is an example:

```
module class_TOP( );
.....
class PCITop;
  logic [31:0] addr;
  logic [31:0] data;

  function void disp (string instName);
```

```

    $display("[%s] addr = %h data = %h", instName,
addr, data);
endfunction
endclass : PCITop

PCITop PCI1, PCI2;

initial begin;
PCI1 = new;//create object PCI1

PCI2 = PCI1; //class assignment

PCI1.addr = 'h1234_5678; //using PCI1 handle
PCI1.data = 'hf0f0_f0f0;

PCI1.disp("PCI1");
PCI2.disp("PCI2");

PCI2.addr = 'h8765_4321; //using PCI2 handle
PCI2.data = 'hffff_0101;

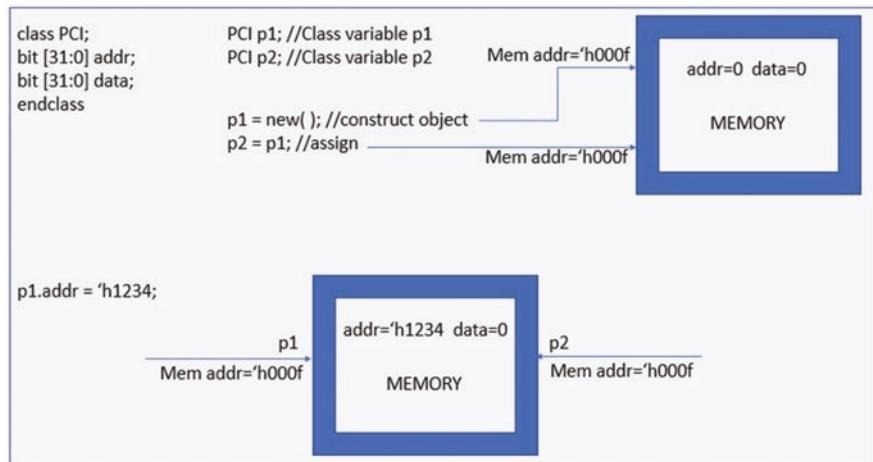
PCI1.disp("PCI1");
PCI2.disp("PCI2");
end
endmodule

```

In this example, we declare a class “PCITop” with addr and data as its properties. We declare two variables of type PCITop (PCI1 and PCI2). Then, we instantiate PCI1 (PCI1 = new). Note that we are only instantiating PCI1 and – not – PCI2. Then we do a class assignment (PCI2 = PCI1). Since only PCI1 was instantiated, both PCI1 and PCI2 will now point to the same object PCI1. In other words, both PCI1 and PCI2 will now point to the same location in memory. Change in PCI1 property values using PCI1 handle (or PCI2 null handle) will reflect on PCI2 (or PCI1). We change PCI1.addr and PCI1.data and see that those changes are reflected on PCI2, as shown in the simulation log. We then change PCI2.addr and PCI2.data and see that those changes are reflected on PCI1.

*Simulation log:*

```
[PCI1] addr = 12345678 data = f0f0f0f0
[PCI2] addr = 12345678 data = f0f0f0f0
[PCI1] addr = 87654321 data = ffff0101
[PCI2] addr = 87654321 data = ffff0101
```



**Fig. 8.2** Class assignment

## 8.9 Shallow Copy

As we saw, class assignment is done as:

```

PCI p1;
p1 = new();
PCI p2;
p2 = p1; //class assignment

```

Both “p1” and “p2” reference the same memory location. But what if you want to make a copy of p1 into p2 – i.e., they do not refer to the same location? For that you do the following:

```

PCI p1;
p1 = new();
PCI p2;
p2 = new p1; //shallow copy

```

“p2” as a new object is created whose class properties are copied from “p1.” This is called shallow copy (we will see what a deep copy means in coming section). Copy allocates memory, and all the variables are copied, such as integers, strings, instance handles, etc., and also randomization and coverage. *But the nested objects themselves are not copied – only the handles and scalar objects and strings.*

Figure 8.3 shows how a shallow copy works.

In Fig. 8.3, we declare a class “PCI” and declare two variables “p1” and “p2” of type PCI. Then we create the object “p1” which allocates memory (assume at

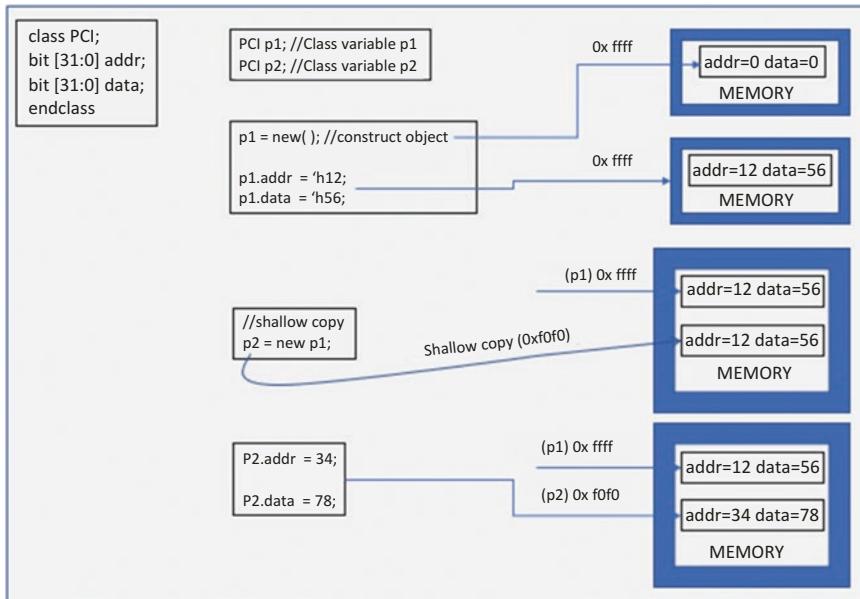


Fig. 8.3 Shallow copy

address `0xffff`) for it and stores the two variables with values (`addr = 'h12` and `data = 'h56`). We then create a shallow copy of `p1` into `p2` (`p2 = new p1;`). Note that since this is a copy (and not assignment), new memory is allocated for the newly created (instantiated) object “`p2`” (assume at address `0xf0f0`). And since this is a copy, the values of “`addr`” and “`data`” are reflected in the shallow copy. Then, we change the properties pointed to by “`p2`” with new values (`addr = 34` and `data = 78`). So, when you look at the final memory, you will see that the values 12 and 56 are stored at the “`p1`” address `0xffff` and the values 34 and 78 are stored at address `0xf0f0`.

Now let us look at a working example. This example will show that the properties of an object are copied, but the nested objects themselves are not copied – only the nested object handles:

```

module class_TOP( );

class PCIClass;
    logic [7:0] burstC;

    function new (logic [7:0] burst);
        burstC = burst;
    endfunction
endclass : PCIClass

```

```

class PCITop;
  logic [31:0] addrTop;
  logic [31:0] dataTop;

  PCIChild PCIC;

  task PCIM(logic [31:0] addr, logic [31:0] data, logic [7:0] burst);
    PCIC = new(burst); //instantiate PCIC
    addrTop = addr;
    dataTop = data;
  endtask

  function void disp (string instName);
    $display("[%s] addr = %h data = %h burst=%h", instName,
addrTop, dataTop, PCIC.burstC);
  endfunction
endclass : PCITop

PCITop PCI1, PCI2;

initial begin;
  PCI1 = new;
  PCI1.PCIM(32'h 0000_00ff, 32'h f0f0_f0f0, 8'h12);
  PCI1.disp("PCI1");

  PCI2 = new PCI1; //Shallow copy of PCI1 into PCI2

  PCI2.disp("PCI2"); //copied content displayed

  PCI2.addrTop = 32'h1234_5678;
  PCI2.dataTop = 32'h5678_abcd;
  PCI2.PCIC.burstC = 8'h 9a;
  PCI2.disp("PCI2");

  PCI1.disp("PCI1");
end
endmodule

```

*Simulation log:*

```
[PCI1] addr = 000000ff data = f0f0f0f0 burst=12
[PCI2] addr = 000000ff data = f0f0f0f0 burst=12
[PCI2] addr = 12345678 data = 5678abcd burst=9a
[PCI1] addr = 000000ff data = f0f0f0f0 burst=9a
```

### V C S   S i m u l a t i o n   R e p o r t

In this example, we declare a class called PCIChild with a property “burstC.” We also declare a class called PCITop with two properties “addrTop” and “dataTop.” We declare a variable PCIc of type PCIChild in the class PCITop and instantiate it in the task “PCIM” (of class PCITop). So, PCIc now holds a handle to the class PCIChild.

We then declare two variables “PCI1” and “PCI2” of type PCITop and instantiate PCI1. We then make a shallow copy of PCI1 into PCI2 (`PCI2 = new PCI1;`). Note that properties are copied. But the nested object PCIc (of class PCITop) is not copied. We then display “addrTop,” “dataTop,” and “burstC” values of PCI1 and PCI2. Since PCI2 has a copy of PCI1 properties, both displays will show the same value. Note that *“burstC” value of PCIChild (handle PCIc) will be the same – because – both PCIc handles (one from PCI1 and one from PCI2) point to the same memory location.* Objects were not copied; only the handles were copied. In other words, the handles (pointers) were copied, but the memory content that they point to were not copied.

We then change the values of properties in PCI2. Then when we display the values of “addrTop,” “dataTop,” and “burstC,” we see that the new values of “addrTop” and “dataTop” are reflected on PCI2, but the values from PCI1 remain as originally assigned. That is because the properties of PCI1 and PCI2 do – not – share the same memory. However, the value of “burstC” for both PCI1 and PCI2 is the same because as mentioned before; only object handles were copied meaning the object handle “PCIc” of both “PCI1” and “PCI2” point to the same memory location. Hence, when we changed the value of “burstC” from PCI2, it got reflected on PCI1 as well. Properties were copied (meaning different memory locations); object handles were copied and not the objects (meaning same memory locations).

Study this carefully since this is one of the important features of classes. Compare this example with the one on deep copy.

## 8.10 Deep Copy

So, since shallow copy does not copy objects, how do we do that? That is where deep copy comes into picture. Deep copy is attained by declaring a custom *copy* function in which we can copy object contents as well. The following example is (almost) identical to the one for shallow copy, but in this example, we define a custom “copy” function to do deep copy:

```

module class_TOP( );

class PCIChild;
    logic [7:0] burstC;

    function new (logic [7:0] burst);
        burstC = burst;
    endfunction
endclass : PCIChild

class PCITop;
    logic [31:0] addrTop;
    logic [31:0] dataTop;

    PCIChild PCIC;

    function new(logic [31:0] addr, logic [31:0] data, logic
[7:0] burst);
        PCIC = new(burst); //instantiate PCIC
        addrTop = addr;
        dataTop = data;
    endfunction

    function void copy (PCITop p);
        addrTop = p.addrTop;
        dataTop = p.dataTop;
        PCIC.burstC = p.PCIC.burstC;
    endfunction

    function void disp (string(instName);
        $display("[%s] addr = %h data = %h burst=%h", inst-
Name, addrTop, dataTop, PCIC.burstC);
    endfunction
endclass : PCITop

PCITop PCI1, PCI2;

initial begin;
    PCI1 = new (32'h 0000_00ff, 32'h f0f0_f0f0, 8'h12);
    PCI1.disp("PCI1");

    PCI2 = new (1,2,3);

```

```

PCI2.copy(PCI1);      //deep copy PCI1 into PCI2

PCI2.disp("PCI2"); //copied content displayed

PCI2.addrTop = 32'h1234_5678;
PCI2.dataTop = 32'h5678_abcd;
PCI2.PCIc.burstC = 8'h 9a;

PCI2.disp("PCI2");
PCI1.disp("PCI1");
end
endmodule

```

*Simulation log:*

[PCI1] addr = 000000ff data = f0f0f0f0 burst=12  
[PCI2] addr = 000000ff data = f0f0f0f0 burst=12  
[PCI2] addr = 12345678 data = 5678abcd **burst=9a**  
[PCI1] addr = 000000ff data = f0f0f0f0 **burst=12**

## V C S   S i m u l a t i o n   R e p o r t

The explanation of this example is identical to that for the shallow copy, so I will not repeat it here. But here is what got added to the class PCITop – a custom “copy” function:

```

function void copy (PCITop p);
    addrTop = p.addrTop;
    dataTop = p.dataTop;
    PCIc.burstC = p.PCIc.burstC;
endfunction

```

Note that in this function we explicitly copy the properties of PCITop and also copy the contents of the nested object PCIc. So, now the properties as well as the object contents are a copy, meaning the properties as well as object contents of PCI1 and PCI2 point to different locations in memory. On calling the copy method, new memory will be allocated to the variables and the objects:

```
PCI2.copy(PCI1);      //deep copy PCI1 into PCI2
```

And we display the contents of PCI2 which will be identical to that of PCI1 because we copied PCI1 into PCI2:

```
[PCI1] addr = 000000ff data = f0f0f0f0 burst=12
[PCI2] addr = 000000ff data = f0f0f0f0 burst=12
```

Then, we change the values of properties and object contents of PCI2 (and leave the PCI1 values to where they were):

```
PCI2.addr = 32'h1234_5678;
PCI2.data = 32'h5678_abcd;
PCI2.PCIc.burstC = 8'h 9a;
```

Then, we again display contents of PCI1 and PCI2:

```
[PCI2] addr = 12345678 data = 5678abcd burst=9a
[PCI1] addr = 000000ff data = f0f0f0f0 burst=12
```

But note that the value of “burst” for PCI1 is different from PCI2. That is because when we changed the value of “burstC” in PCI2 (PCI2.PCIc.burstC = 8'h 9a;), the object content PCI2.PCIc.burstC points to a different location than the object content PCI1.PCIc.burstC. So, the new value got reflected on PCI2.PCIc.burstC, but the value of PCI1.PCIc.burstC remained the same as what was originally assigned to it.

Let us compare the two simulation logs: one for shallow copy and one for deep copy.

*Shallow copy simulation log:*

```
[PCI1] addr = 000000ff data = f0f0f0f0 burst=12
[PCI2] addr = 000000ff data = f0f0f0f0 burst=12
[PCI2] addr = 12345678 data = 5678abcd burst=9a
[PCI1] addr = 000000ff data = f0f0f0f0 burst=9a
```

*Deep copy simulation log:*

```
[PCI1] addr = 000000ff data = f0f0f0f0 burst=12
[PCI2] addr = 000000ff data = f0f0f0f0 burst=12
[PCI2] addr = 12345678 data = 5678abcd burst=9a
[PCI1] addr = 000000ff data = f0f0f0f0 burst=12
```

Compare the two logs and you will understand the difference between shallow copy and deep copy.

Here is another example of deep copy:

```
class bus;
  int addr = 'fff;
  int data = 'haa;
endclass

class icontect;
  int IC_ID_i = 2019;
  bus m_i = new( );

  function void copy(icontect c);
    IC_ID_i = c.IC_ID_i;
    m_i = new c.m_i;
```

```

    endfunction
endclass

module deep_copy();
  iconnect IC1;
  iconnect IC2;

  initial begin
    IC1 = new;
    IC2 = new;
    IC2.copy(IC1);

    IC1.IC_ID_i = 2020;
    IC1.m_i.addr = 'hbb;
    IC1.m_i.data = 'hcc;

    $display("IC1.IC_ID_i = %0d", IC1.IC_ID_i);
    $display("IC1.m_i.addr = %0h", IC1.m_i.addr);
    $display("IC1.m_i.data = %0h", IC1.m_i.data);
    $display("\n");
    $display("IC2.IC_ID_i = %0d", IC2.IC_ID_i);
    $display("IC2.m_i.addr = %0h", IC2.m_i.addr);
    $display("IC2.m_i.data = %0h", IC2.m_i.data);
  end
endmodule

```

*Simulation log:*

```

# run -all
# IC1.IC_ID_i = 2020
# IC1.m_i.addr = bb
# IC1.m_i.data = cc
#
#
# IC2.IC_ID_i = 2019
# IC2.m_i.addr = ff
# IC2.m_i.data = aa
# exit

```

In this example, class “iconnect” instantiates class “bus” (bus m\_i = new(); ) and defines a custom copy function. The argument “c” to the copy function is of type “iconnect.” This function copies the property (IC\_ID\_i = c.IC\_ID\_i;) of class “iconnect” and also copies the object “m\_i” (m\_i = new c.m\_i;). This way when you call this copy function all the object contents of “m\_i” will be copied and not just the object handle. We copied the properties as well as the contents of the object.

We copy IC1 into IC2 (`IC2.copy(IC1)`).

We then change the properties of IC1 (addr, data, and IC\_ID\_i). The simulation log shows that the properties of IC1 and IC2 have different values since both objects (IC1 and IC2) and their embedded object “m\_i” point to different memory locations. They are a deep copy and not a shallow copy.

## 8.11 Upcasting and Downcasting

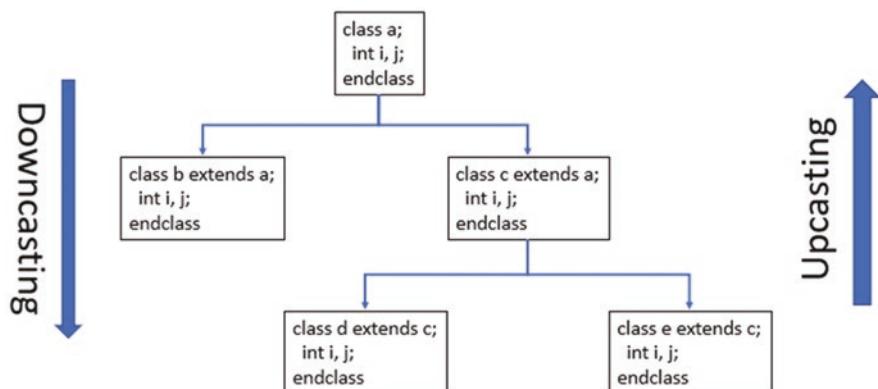
There are many ways to explain upcasting and downcasting. The web is filled with different ways to explain this. I will describe it based on memory allocation model, since that is the best way to understand the concepts and see how they tie into dynamic casting. Let us look at Fig. 8.4.

The explanation is based on what we discussed in inheritance memory allocation (Sect. 8.3.1). As shown in Fig. 8.4, class “e” extends class “c” which in turn extends class “a.” Also, class “b” extends class “a.”

Let us say we do the following:

```
a a1; //base class variable
e e1;
e1 = new;
a1 = e1; //assigning extended object handle 'e1' to
          //base class variable 'a1'
```

When you instantiate '`e1 = new( )`', you are allocating memory for the properties of class e, class c, and class a. Memory is allocated for the entire inheritance tree (upward). So, when we assign an extended object handle “e1” to the base class variable “a1,” we are doing an upcast. What that means is that if you access



**Fig. 8.4** Upcasting and downcasting

“a1.i” it will grab the value of “i” from the memory allocated to class “a.” Similarly, if you access e1.i, it will grab the value of “i” from the memory allocated to class “e.” In other words, even though “e1” is assigned to “a1,” the value of a1.i is different from the value of e1.i. This is called upcasting.

*Upcasting is supported in SystemVerilog.*

*An extended (child) class handle can be assigned to a base (parent) class variable.*

In contrast, let us look at the code below:

```
a a1;
e e1;
a1 = new;
e1 = a1; //ILLEGAL
```

Here, we instantiate “a1 = new.” The memory is allocated for class “a,” but *this memory allocation does not cascade downward to its inheritance tree*. Memory is not allocated for class “e” (or class “c”). So, when we do “e1 = a1,” we do not have storage for “e1” and hence the assignment is not allowed.

*Note that downcasting does not happen automatically via an assignment. But \$cast() does downcasting. This is a dynamic operation, and it returns a bit indicating it succeeded or not.*

*You cannot assign a base (parent) class handle to an extended (child) class variable.*

The problem is that we are making an assumption that the base class instance “a1” is referring to the extended class variable (null handle) “e1” and that they are type compatible.

To check that this assumption is correct, we need to do a dynamic cast. *Dynamic cast is both an assignment and a type check.*

To do “e1 = a1,” we need to first cast “a1” into “e1,” as follows:

```
$cast(e1,a1); //dynamic casting
```

Note: Dynamic casting is further discussed in Sect. 2.14.

Let us first look at assigning child class handle to parent class variable. Here is a working example:

```
class p_class;
    bit [31:0] p_Var;
    function void display();
        $display("p_Var = %0d",p_Var);
    endfunction
endclass

class c_class extends p_class;
    bit [31:0] c_Var;
```

```

function void display( );
    super.display( );
    $display("c_Var = %0d", c_Var);
endfunction
endclass

module top;
initial begin
    p_class p;
    c_class c = new( );
    c.p_Var = 10;
    c.c_Var = 20;
    //assigning child class handle to parent class variable
    p = c;
    c.display( );
end
endmodule

```

In this example, we are declaring a parent class “p\_class” and it’s extended child class “c\_class.” In the module “top,” we declare a variable “p” of class type “p\_class.” We also instantiate class “c\_class” with object handle name “c.” We assign properties of the class “c” some values.

Then, we assign “p = c” – child class handle “c” assigned to parent class variable “p.” So, based on the above discussion of upcasting and memory allocation, when you instantiate “c\_class,” it will allocate memory for “p\_class” as well. That being the case, “c” can now indeed be assigned to “p” because memory for “p” was allocated when “c” was instantiated. Memory for both “c” and “p” are allocated. Here is the simulation log as expected:

```

p_Var = 10
c_Var = 20
VCS Simulation Report

```

Now, let us reverse the roles and assign a parent class variable to a child class handle. Simply replace “p = c” statement with “c = p” in the code. Now, you will get a compile time error (Synopsys – VCS):

```

Error-[SV-ICA] Illegal class assignment
testbench.sv, 32
"c = p;"
Expression 'p' on rhs is not a class or a compatible class and hence cannot
be assigned to a class handle on lhs.
Please make sure that the lhs and rhs expressions are compatible.

```

That is where you need to do a dynamic cast. It makes sure that when the parent class variable “p” is assigned to child class handle “c,” they are type compatible. Check to see that the parent class variable is pointing to a type compatible child

class handle. Here is the change in the initial block. Rest of the code remains the same:

```
p_class p;
c_class c = new( );
c.p_Var = 10;
c.c_Var = 20;
c.display( );
$cast(c, p); //dynamic cast
```

This will pass run-time check.

Let us look at another example:

```
class animals;
    string color = "white";
    function void disp;
        $display("color = %s", color);
    endfunction
endclass

class buffalo extends animals;
    string color = "black";
    function void disp;
        $display("color = %s", color);
    endfunction
endclass

program tb;
    initial begin
        animals p;
        buffalo c;

        c = new( ); //allocate memory for c
                    //this will allocate memory for both 'c' and 'p'

        p = c; //upcasting
        p.disp;
        c.disp;
    end
endprogram
```

*Simulation log:*

color = white

color = black

```
$finish at simulation time          0
V C S   S i m u l a t i o n   R e p o r t
```

In this example, we declare two classes “animals” and “buffalo,” where “buffalo” is extended from “animals.” The string parameter of “buffalo” overrides the string parameter of “animals.”

We declare two variables, one of type “animals” (“animals p”) and another of type “buffalo” (“buffalo c”). We then instantiate “c.” So, as per upcasting rules, this will not only allocate memory for “c” but also for “p” (memory is allocated to the entire upward class chain). Then we assign “p = c.” What this means is that when we call the function “disp” from “c,” it will access it from class “buffalo,” and when we call function “disp” from “p” (parent class), it will access it from class “animals” (even though we did not instantiate class “animals”). Two principles come into picture here. One is upcasting and another is assignment of child class handle to parent class variable. Simulation log shows the desired results.

## 8.12 Polymorphism

With knowledge of class assignment, inheritance, and upcasting/downcasting, we will now be able to discuss polymorphism. Polymorphism is derived from Greek word, where “poly” means many and “morph” means forms, i.e., it is an ability to appear in many forms. The OOP (object-oriented programming) term for multiple routines sharing a common name is “polymorphism.”

In SystemVerilog, it means that we can use a superclass variable to hold subclass objects and to reference the methods of those subclasses directly from the superclass variable. Appropriate subclass methods can be accessed from the superclass variable, even though the compiler did not know – at compile time – what was going to be loaded into it. So, polymorphism is the ability to have the same code act differently based on the type of the object that the superclass variable is holding.

To achieve this, functions and/or tasks in a base class are declared virtual so that the extended class can override them (actually, that is not entirely true. You can override a function even if it is not declared as virtual (as we have seen before). The difference is whether you can access the overridden implementation from a base class handle or a derived class handle.

So, we can say that polymorphism = inheritance + virtual methods + upcasting.  
Let us look at an example:

```
class vehicle; // Parent class
    virtual function vehicle_type( ); // Virtual function
        $display("vehicle");
    endfunction

    virtual task color( ); // Virtual task
```

```
        $display("It has color");
    endtask
endclass

class four_wheeler extends vehicle; //child class
function vehicle_type( ); //override parent class virtual
                        //function
    $display("It is a four wheeler");
endfunction

task color( ); //override parent class virtual task
    $display("It has different colors");
endtask
endclass

class BENZ extends four_wheeler; // "grand" child class
function vehicle_type();
    $display("It is a Benz");
endfunction

task color();
    $display("It is Black");
endtask
endclass

module polymorphism;
initial begin
    vehicle vehcl;
    four_wheeler four_whlr;
    BENZ benz;

    four_whlr = new ( );
    benz = new ( );

    vehcl=four_whlr;//parent class variable holding child
                    //class handle. No need to create an
                    //object (of vehcl) by calling a new method

    vehcl.vehicle_type( );
    // accessing child (four_whlr) class method from
    // parent class (vehcl) variable

    vehcl=benz;
    vehcl.vehicle_type( );
    //accessing "grand" child method from parent class
```

```

//variable (vehcl)

four_whlr=benz;
four_whlr.color( );
// accessing child class method from parent class
// "four_whlr"
end
endmodule

```

*Simulation log:*

It is a four wheeler  
 It is a Benz  
 It is Black  
 \$finish at simulation time 0

V C S   S i m u l a t i o n   R e p o r t

In this code, we first declare a base (parent) class called “vehicle.”

*Important point:* In this base class, we declare two virtual methods “vehicle\_type” and “color.” We will discuss virtual methods in Sect. 8.13.1. For now, understand that, when we assign a child class object handle to the parent class variable, that in order to access the methods of the child class object using the parent class variable, we need to declare the methods of the parent class as virtual. If we do not declare parent class methods as virtual, polymorphism will not work. If you do not get this, do not worry, we will discuss virtual methods in Sect. 8.13.1.

Then we create a class called “four\_wheeler” which extends parent class “vehicle” and overrides the virtual methods of the parent class. We further extend the class “four\_wheeler” to create class “BENZ.” So, in essence, we have created a hierarchy of classes. “four\_wheeler” extends “vehicle” and “BENZ” extends “four\_wheeler.”

In the program “polymorphism,” we declare variables of each type but instantiate only the two child classes, namely, “four\_wheeler” (named “four\_whlr”) and “BENZ” (named “benz”). Note that we did not instantiate parent class “vehicle,” simply declared a variable of its type called “vehcl.” Note the effect of upcasting into play here. When you instantiate child class “four\_wheeler,” you allocate memory for both, class “four\_wheeler” and class “vehicle.”

We then assign “vehcl = four\_whlr.” Assign child class instance handle to the parent class variable. With this, we can now access the methods declared in “four\_wheeler” directly using the parent class variable “vehcl.” This is polymorphism. We do not have to use the child\_class handle (“four\_whlr”) to access its method. We were able to do so, simply using the parent class variable (“vehcl”). We then call the method “vehicle\_type” using the parent class variable “vehcl” (“vehcl.vehicle\_type”). This will execute the function “vehicle\_type” method of the child class “four\_wheeler.”

We then assign “vehcl” to “benz” (in other words, assigned the grand child to parent class variable). With this we are now able to access the methods of “benz”

directly using the parent class variable “vehcl.” We access the method “vehicle\_type” of “benz.” The result is shown in simulation log.

So, we are able to access the methods of “four\_whlr” and “benz” directly using the parent class variable “vehcl.” In other words, even though all the methods are being called using the same parent class variable, different methods are getting called. That is polymorphism.

Similarly, we extend “four\_wheeler” to create class “BENZ” (so, now “four\_wheeler” is the parent class). Then we assign “four\_whlr = benz.” This will now allow us to access, “BENZ” class methods directly by using parent class handle “four\_whlr.” Using the same arguments as above, we then execute “color” method of class “BENZ” using the parent class handle “four\_whlr” (“four\_whlr.color”). The result is shown in simulation log.

Let us look at an example that further exemplifies polymorphism. This is more likely the scenario where you will use polymorphism. In this example, we take an array of the parent class and assign it object handles of different subclasses. This way, we can access methods/properties of different subclasses directly looping through the parent array class variable:

```
// base class
class animals;
    virtual function void display( );
        $display("Inside base class animals");
    endfunction
endclass

// extended class 1
class parrot extends animals;
    function void display( );
        $display("Inside extended class parrot");
    endfunction
endclass

// extended class 2
class tiger extends animals;
    function void display( );
        $display("Inside extended class tiger");
    endfunction
endclass

// extended class 3
class lion extends animals;
    function void display( );
        $display("Inside extended class lion");
    endfunction
endclass
```

```

// module
module polymorphism;

initial begin
  //instantiate subclasses
  parrot p1 = new( );
  lion A1 = new( );
  tiger t1 = new( );

  //base class array variable
  animals a1[3];

  //assigning extended class object handles to
  //base class array variable
  a1[0] = p1;
  a1[1] = A1;
  a1[2] = t1;

  //accessing extended class methods using base class
variable
  a1[0].display( );
  a1[1].display( );
  a1[2].display( );
end
endmodule

```

*Simulation log:*

Inside extended class parrot  
 Inside extended class lion  
 Inside extended class tiger

V C S   S i m u l a t i o n   R e p o r t

In this example, we create a base class “animals” and create three extended classes called “parrot,” “tiger,” and “lion.” The virtual function “display” of base class is overridden in all three extended classes. Then we instantiate all three extended classes. We declare an array variable of base class type (“animals a1[3]”). Now for each element of the array “a1,” we assign an extended class object handle. With this, we can now access the child class method “display” simply using an array element of the base class array variable. In other words, even though all the methods are called using the array elements of the same base class variable, different methods get called.

## 8.13 Virtual Methods, Pure Virtual Methods, and Virtual Classes

### 8.13.1 Virtual Methods

This topic ties directly into polymorphism discussion. Polymorphism worked because the base class methods were declared virtual. To recap, when we assign a child class object (instance) handle to the parent class variable, that in order to access the methods of the child class object using the parent class variable, we need to declare the methods of the parent class as “virtual.”

Virtual methods include the “virtual” keyword before the function or task keyword. Classes with non-virtual methods fix the method code to the class object when constructed. Virtual method functionality is set at run time. Note that once a method is declared “virtual,” it is always “virtual.” You cannot take a “virtual” method of the base class and convert it into a non-virtual method in an extended class.

Let us take an example.

First example. We *do not* declare the parent class methods as virtual. Let us see what happens:

```
class packet;
    function void disp;
        $display("From packet");
    endfunction
endclass

class eth_packet extends packet;
    function void disp;
        $display("From eth_packet");
    endfunction
endclass

module class_TOP( );
    initial begin
        packet p1;
        eth_packet e1 = new;

        //assign child class handle to parent class variable.
        p1 = e1;
        p1.disp;

    end
endmodule
```

In this example, we declare two classes. “eth\_packet” is extended from its parent class “packet.”

Then, in the initial block, we declare the following:

```
packet p1;
eth_packet e1 = new;
```

“p1” is a variable of parent-type “packet” and e1 is an instance of “eth\_packet.”

Now, we assign “e1” object handle to “p1” variable (“p1 = e1”), child class handle assigned to parent class variable. Then, we access “p1.disp.” We did exactly the same in polymorphism examples. But in this example, note that the “disp” function in “packet” (parent class) is *not* declared virtual. “p1.disp” will not look into the instance “e1” to execute its “disp” function. Instead, “p1.disp” will execute the “disp” function declared in class “packet.” In other words, since the method is not declared virtual, it blocks access to subclass members for subclass instance in a parent class variable.

To reiterate, in this case, the class members/methods are resolved by searching from the *class handle type*. Even though the parent class variable points to a child instance, we do not look into the child instance for the function “disp.” Hence, we do not execute the function “disp” from the class “eth\_packet”: instead we execute it from the class “packet.” Without the “disp” being virtual, the simulator looks for the first non-virtual function, which is “disp” in class packet and executes it.

#### *Simulation log:*

From packet

#### VCS Simulation Report

So, we did everything as we did in polymorphism examples, but got different results. That is because the function “disp” in the parent class “packet” was not declared “virtual.”

Let us take the same example but declare function “disp” in parent class “packet” as virtual. Identical example as above, except for one line change:

```
class packet;
//parent class method declared 'virtual'
virtual function void disp;
    $display("From packet");
endfunction
endclass

class eth_packet extends packet;
function void disp;
    $display("From eth_packet");
endfunction
endclass

module class_TOP( );
```

```

initial begin
    packet p1;
    eth_packet e1 = new;

    //assign child class handle to parent class variable.
    p1 = e1;
    p1.disp;

end
endmodule

```

In this example, when we do the “`p1 = e1`” and then “`p1.disp`,” the simulator will first look into the class “`packet`” for the method/function “`disp`” according to its handle type. But when it reaches the function “`disp`,” it sees that it is declared *virtual*. In SystemVerilog, when a method in the parent class is declared virtual, the same method in child classes also gets declared virtual automatically. The simulator looks for the last virtual function in the chain of extended classes. So, “`p1.disp`” will now look into the class instance “`e1`” for the same method. We will now execute the function “`disp`” from the child instance “`e1`” – exactly as we saw in the examples on polymorphism. Here is the simulation log:

*Simulation log:*

From `eth_packet`

V C S   S i m u l a t i o n   R e p o r t

So, *in order for polymorphism to work, we must declare the methods in the parent class to be virtual*. Note that *when we declare the parent class method as virtual, the same method in subclass is automatically declared virtual*.

If all this sounds, confusing, well to some extent, it is. Revisit the description above and experiment with the examples I have provided, and you will understand what is going on.

High-level points to note:

- Virtual method functionality is set at run time.
- Extending a virtual method requires strict method argument compatibility.
- A base class can have a non-virtual method, and an extended class can override that method with a virtual method.
- Once a method is declared to be virtual, all extended class overrides of that method will also be virtual, with or without the `virtual` keyword.

### 8.13.2 Virtual (Abstract) Class and Pure Virtual Method

In many projects, you want to declare a prototype class with methods that can be overridden by sub-projects. You want to declare a class that sets out as a prototype for subclasses. You want to declare only a method prototype (a.k.a. *pure virtual method*), which must be customized and defined by each sub-project (each

subclass). That is what a virtual (a.k.a. an abstract) class allows you to do. You can generalize the virtual class and provide method prototypes that will be redefined by its subclasses. This allows different subclasses to share a common method prototype. A set of classes may be created that can be viewed as all begin derived from a base class.

For example, a common base class of type packet that sets out the structure of packets, but is incomplete,

would never be constructed. This is characterized as a virtual or abstract class. From this abstract base class, however, a number of useful subclasses may be derived, such as Ethernet packets, GPS packets, etc.

Each of these packets might look very similar, all needing the same set of methods, but they could vary,

significantly in terms of their internal details.

Abstract classes form the basis of many class libraries by implementing core pieces of functionality like configuration, reporting, and interprocess communication. Abstract classes also provide an API that makes it easier to integrate class based models from many different independent sources. This is why we see many local and protected members inside an abstract class restricting us to the published API. Such methodology is implemented in Universal Verification Methodology (UVM).

A virtual class cannot be instantiated. It can only be extended by a non-virtual subclass. You can define a variable of virtual class type.

Here is the syntax:

```
virtual class Packet;  
endclass
```

A virtual method in a virtual class can be declared as a prototype without providing an implementation. This is called a “*pure virtual*” method. “pure” is the keyword. Here is the syntax. In this example, “pure virtual packetDisp” can only be a prototype. Cannot have implementation in it:

```
virtual class Packet;  
  pure virtual packetDisp ( );  
endclass
```

Main points:

- An abstract (virtual) class cannot be instantiated. It can only be derived.
- You can declare a variable of class type “virtual.”
- A virtual class needs to be extended (derived subclass) to provide implementation detail.
- “pure virtual” methods can only be declared in a “virtual class.”
- “pure virtual” method can only be a prototype. Cannot have implementation.

- The methods in a “virtual” class are not “virtual” by default. They must be explicitly declared “virtual.”

Since you cannot instantiate a virtual class, if you did the following, you would get a compile *error*:

```

virtual class bus;
    bit [31:0] addr;
endclass

module top;
    initial begin
        bus b1; //Declare a variable of type virtual
class - OK.
        b1 = new( );
        //Compile ERROR - cannot instantiate a virtual class
    end
endmodule

```

Here is the compile *error* you will get (Synopsys – VCS):

Error-[SV-ACCNBI] An abstract class cannot be instantiated  
testbench.sv, 48

top, "b1 = new();"

Instantiation of the object 'b1' cannot be done because its type 'bus' is  
an abstract base class.

Perhaps there is a derived class that should be used.

Here is a complete example. The example showcases a “virtual class” and a “pure virtual” method contained in it. It also shows that you can have “virtual” methods as well as regular methods in a “virtual class”:

```

virtual class BaseClass;

    //pure virtual method can be declared only in a vir-
    tual class.
    //pure virtual method can only be a prototype.
    //Cannot have implementation.

    pure virtual function void disp( );

    //You can indeed define a virtual function/task as well
    virtual function void displ( );
    $display("Virtual Function 'displ' from class BaseClass");

```

```

    endfunction

    //You can also define a regular function/task
    function void disp2( );
        $display("Function 'disp2' from BaseClass");
    endfunction
endclass

class ChildClass extends BaseClass;

    //MUST define implementation of 'disp' since it is
declared
    //''pure virtual' in 'virtual class' BaseClass

    function void disp( );
        $display("pure virtual function 'disp' of baseClass
implemented in class ChildClass");
    endfunction

    function void disp1( );
        $display("virtual function 'disp1' of baseClass over-
ridden in class ChildClass");
    endfunction

endclass

module tb;
    BaseClass base;
    ChildClass child;

    initial begin
//base = new; //cannot instantiate virtual class - Compile ERROR
        child = new;
        base = child; //upcasting

        base.disp;
        base.disp1;
        base.disp2;
    end
endmodule

```

*Simulation log:*

pure virtual function 'disp' of baseClass implemented in class ChildClass  
virtual function 'disp1' of baseClass overridden in class ChildClass  
Function 'disp2' from BaseClass

### V C S   S i m u l a t i o n   R e p o r t

In this example, we declare a “virtual class” BaseClass (also known as an abstract class). In this class, we declare a “pure” virtual method called “disp.” Since this is a “pure virtual” method, it can only be a prototype. It cannot have implementation in it. This “pure virtual” method can only be declared in a “virtual” class.

If you change the “virtual class BaseClass” to “class BaseClass,” you will get the following compile error (Synopsys – VCS):

```
Error-[PVMNA] Pure virtual method not allowed
testbench.sv, 7
"BaseClass::disp"
A pure virtual method cannot be defined in a non-abstract class 'BaseClass'.
```

For the sake of completeness, I have also shown that you can indeed declare a “virtual” method (i.e., non-pure) and also a regular method. In other words, a “virtual” class is not restricted to “pure virtual” methods only.

But it makes most sense to declare the methods of a “virtual” class as “pure virtual.” This way the class remains abstract, and various projects can then implement the required functionality of the “pure virtual” methods. By declaring a method as pure virtual, you are (1) disallowing the base class to be instantiated by itself – it can only be instantiated via a derived class and (2) requiring that derived classes provide an implementation.

So, in our example, we have defined three method types: “pure virtual,” “virtual,” and regular method. The class “ChildClass” extends the “virtual” class BaseClass. In the ChildClass, we provide implementation of the “pure virtual” method “disp.” If you do not provide an implementation of the “pure virtual” method, you will get a compile error as shown below (Synopsys – VCS):

```
Error-[SV-VMNI] Virtual method not implemented
testbench.sv, 7
Virtual method 'disp' not implemented in class 'ChildClass' (declared in
testbench.sv, at line 20).
```

We then declare a variable for the “virtual” class BaseClass (BaseClass base;) and also a variable for ChildClass (ChildClass child;). We then instantiate the ChildClass (child = new;). Then we assign “child” class instance to base class handle “base” (base = child;). This is upcasting as we have seen in previous sections. This allows the base class variable to access child class methods.

Following three statements are then executed:

```
base.disp;
```

This will display the \$display from the extended class where implementation for “disp” is provided. In the simulation log, you will see:

```
pure virtual function 'disp' of baseClass implemented in class ChildClass
```

Then we do:

```
base.disp1;
```

This will execute the \$display from the extended class “ChildClass” based on the principles of polymorphism (refer to the previous section to see how polymorphism works). You will see the following in the simulation log:

```
virtual function 'disp1' of baseClass overridden in class ChildClass
```

Then we do:

```
base.disp2;
```

This will simply \$display from the method “disp2” of base class. So, you will see the following in the simulation log:

Function 'disp2' from BaseClass

Following are further examples of ways in which virtual class/method works.

Example 1:

```
virtual class bus;
  bit [7:0] addr;

  pure virtual function void IC;

endclass

virtual class iConnect extends bus;
  //Not necessary to provide body for function IC
  //You may, but not necessary
endclass

class iConnect1 extends bus;

  //MUST provide body for function IC
  function void IC;
  endfunction
endclass
```

In this model, the “virtual class bus” has a “pure virtual” method called IC. Since this is a “pure virtual” method, it must be defined in an extended class of “bus.” In our case, it is *not* necessary to provide the definition of “pure virtual” method IC in extended “virtual class iConnect.” You *may* provide the definition in the extended class, but it is not necessary, if the extended class is also “virtual.” And, and as we know, if the extended class is not “virtual,” then you must specify the definition of the “pure virtual” function IC in the extended class.

Here’s a variation of the above example, showing further caveat of the rules.

Example 2:

```
virtual class bus;
    bit [7:0] addr;
    pure virtual function void IC;
endclass

virtual class iConnect extends bus;
    //body for function IC is provided
    virtual function void IC;
    endfunction
endclass

class iConnect1 extends iConnect;
    //Body for function IC optional
endclass
```

Similar to the above example, a “pure virtual” method IC is declared in class “bus.” Then we declare another “virtual class” iConnect which is extended from class “bus.” As noted above, since iConnect is a virtual class, you do not necessarily have to provide the body for function “IC.” But in this example, we do provide the body for function “IC” and declare that also as virtual (you do not necessarily have to). Now, we declare another class iConnect1 which is extended from class “iConnect” (note that iConnect1 is extended from “iConnect” and not from class “bus”). Since we have provided the body of function “IC” in the virtual class “iConnect,” we do not necessarily have to provide the body for function “IC” in the extended class “iConnect1.”

I suggest reading the paper titled “SystemVerilog Virtual Classes, Methods, Interfaces and their use in verification and UVM” by Sunburst Design (Cummings, Sunburst Design, n.d.)

## 8.14 Data Hiding (“local,” “protected,” “const”) and Encapsulation

By default, the members and methods of a class are accessible from outside the class using the class’ object handle, i.e., they are public. What if we do not want some members and some methods to be accessible from outside the class? We want to keep the members of a class local so that no external class (even the extended class) can access it. This is to prevent accidental modification of the class members/methods.

In large projects, we might use externally provided “base” class libraries (as in UVM). Such third-party base class libraries need to make sure that their customers cannot accidentally change/override members and methods of the class which can have a serious ripple effect on the rest of the project. This also prevents other programmers from relying on a specific implementation. Data hiding hides implementation details to reduce complexity and raise the level of abstraction. Note that non-local methods that access local class properties or methods can be inherited and work properly as methods of the subclass.

The technique of hiding data within the class and making it available only through methods is called *encapsulation*. In other words, encapsulation is creating containers of data along with their associated behaviors, i.e., the code that operates on that data.

There are two ways to hide data in a class. They are achieved by prefixing the class members with the following keywords:

```
local
protected
```

### **8.14.1 Local Members**

Local members are not visible outside the class and are not visible to extended/derived classes either. Methods local to the class can access these local members. Both, properties and methods, of a class can be “local.” The keyword to make a class member local is *local*. Here is a simple example:

```
class packet;
    local int addr; //local property

    local function void disp(input int data); //local method
        $display("data = %h", data);
        $display("addr = %h", addr); //access local property
    endfunction
endclass

module class_TOP( );
    initial begin
        packet p1;
        p1 = new( );

        // p1.addr = 'hff;
        //COMPILE ERROR - can't access 'local' property
        // p1.disp(20);
        //COMPILE ERROR - can't access 'local' method
```

```

    end
endmodule

```

In class “packet,” we define a “local” property called “addr.” Since this is local, it is visible only to its methods within the class. We also declare a function “disp” as a local method. It can access the local property “addr.” Note that this function (“disp”) does –*not* – have to be local to access the local property. It can be a regular function as well.

In the module “class\_top,” when we try to access the local property “addr” of the class “packet” (“p1.addr = ‘hff;”), we will get the following error (Synopsys – VCS):

Error-[SV-ICVA] Illegal class variable access

testbench.sv, 25

Local member 'addr' of class 'packet' is not visible to scope 'class\_TOP'.

Please make sure that the above member is accessed only from its own class properties as it is declared as local.

Similarly, if we try to access the method “disp” of class “packet” (“p1.disp;”), we get a compile error as follows (Synopsys – VCS):

Error-[SV-ICMA] Illegal class method access

testbench.sv, 26

Local method 'disp' of class 'packet' is not visible to scope 'class\_TOP'.

Please make sure that the above method is called only from its own class properties as it is declared as local.

Now, let us take the same example and see if “local” properties/methods can be accessed from an extended class:

```

class packet;
    local int addr;

    local function void disp(input int data);
        $display("data = %h", data);
        $display("addr = %h", addr); //access local property
    endfunction

endclass

class eth_packet extends packet;
    function set_addr;
        //addr = 'hff; //COMPILE ERROR -
        //can't access 'local' property from extended class
    endfunction

```

```

function void eth_disp;
    //super.disp(50); //COMPILE ERROR -
    //can't access 'local' method from extended class
endfunction

function void disp(input int data);
    //OK to override 'local' method in extended class
    $display("From eth_packet data=%d", data);
endfunction
endclass

module class_TOP();
    initial begin

        eth_packet e1;
        e1 = new();
        e1.disp(50);

    end
endmodule

```

This example is similar to the one above it. But in this example, we extend the base class “packet” to the class “eth\_packet.” In “eth\_packet” we try to access the local property “addr” (of class “packet”). This will give us a compile error as follows:

Error-[SV-ICVA] Illegal class variable access  
testbench.sv, 14

Local member 'addr' of class 'packet' is not visible to scope 'eth\_packet'.  
Please make sure that the above member is accessed only from its own class properties as it is declared as local.

Similarly, when we try to access the local method “disp” of class “packet” (from “eth\_packet”), we get the following compile error:

Error-[SV-ICMA] Illegal class method access  
testbench.sv, 19

Local method 'disp' of class 'packet' is not visible to scope 'eth\_packet'.  
Please make sure that the above method is called only from its own class properties as it is declared as local.

We then override the local method “disp” (of class “packet”) in class “eth\_packet.” This is ok. Overriding a local method of a parent class in a child class is ok.

We then access the method “disp” of “eth\_packet” from module “class\_TOP”:

```
eth_packet e1;
e1 = new( );
e1.disp( );
```

And as expected, we get the \$display from the “disp” function of the class “eth\_packet.”

*Simulation log:*

```
From eth_packet data=      50
VCS Simulation Report
```

### 8.14.2 Protected Members

A protected class property or method has all of the characteristics of a local member, except that it can

be inherited, it is visible, to extended/derived classes. They can be accessed by the extended subclass but not from outside the class. The keyword to make a class member protected is *protected*. Note that, obviously, it is an error to define members to be both local and protected or to duplicate any of the other modifiers.

Let us look at the same example as above but declare properties as “protected” properties. We will see that the protected properties and methods are visible to the extended class – but – not visible outside of the class. Also, note that protected properties within the class can be accessed by normal methods (i.e., the methods do *not* have to be “protected”):

```
class packet;
  protected int addr;

  protected function void disp(input int data);
    $display("From packet");
    $display("\t data = %h", data);
    $display("\t addr = %h", addr);
  endfunction

endclass

class eth_packet extends packet;
  function set_addr;
    addr = 'hff;
    //protected property 'addr' visible to extended class
  endfunction
```

```

function void eth_disp;
    super.disp('hff);
        //protected method 'disp' visible to extended class
endfunction

function void disp(input int data);
    //OK to override 'protected' method in extended class
    $display("From eth_packet");
    $display("\t data = %h", data);
endfunction
endclass

module class_TOP();
    initial begin

        packet p1;
        eth_packet e1;

        e1 = new();
        p1 = new();

        //p1.disp(20); //COMPILE ERROR
        //protected methods 'disp' not visible outside the class

        e1.eth_disp;
        e1.disp('h ffff);

    end
endmodule

```

In this example, we declare a protected property “addr” and a protected method “disp” in class “packet.” In the extended class “eth\_packet,” we access the property “addr.” This is ok, since protected properties of a base class are accessible/visible to its extended class. Similarly, we access the method “disp” (of class “packet”) from the extended class “eth\_packet.” This is ok as well, since protected methods of a base class are accessible/visible to its extended class.

In the module “class\_TOP,” we instantiate both “packet” and “eth\_packet.” But when we try to access “disp” function of class “packet” from module “class\_TOP,” we get a compile error. This is because a protected method (or property) is not visible from outside the class. Here is the error from Synopsys VCS:

Error-[SV-ICMA] Illegal class method access  
testbench.sv, 39

Protected method 'disp' of class 'packet' is not visible to scope 'class\_TOP'.

Please make sure that the above method is called only from its own class properties as it is declared as protected.

We then access the “disp” and “eth\_disp” methods of class “eth\_packet” from module “class\_TOP.” Note that “eth\_disp” accesses the protected method “disp” of class “packet.” We get the following simulation log.

*Simulation log:*

From packet

```
data = 000000ff  
addr = 00000000
```

From eth\_packet

```
data = 0000ffff
```

```
V C S S i m u l a t i o n R e p o r t
```

Finally, it is considered a good practice to make members local or protected. This facilitates data hiding that we discussed earlier.

### **8.14.3 “*const*” Class Properties**

A class property can be defined as “read-only” by declaring it as a constant. Sometimes in our base classes, we want some of our class properties to be read-only so that others cannot change them. This specific behavior can be achieved using the *const* qualifier. It means that if the class property is declared with *const* keyword, it cannot be modified or updated during simulation run time.

Constant declaration can be of two types, i.e., global constant and instance constant. The keyword to declare a property constant is *const*.

#### **Global Constant**

For global constant properties, the constant value (initial value) is assigned at the time of declaration with :*const*” qualifier. Thereafter the same value is kept by that property. No update to that property value is allowed.

#### **Instance Constant**

For instance constant properties, it’s a two-step process. First, the property is declared inside the class with “*const*” qualifier. Second, the value of that property is assigned inside the constructor of that class, i.e., new( ) method of the class. Here after, this initialized value is not allowed to be updated.

Here is an example showing correct and incorrect use of global and instance constants:

```
class packet;
  const int serialNum = 'h1234; //Global const
  const int packetID;           //Instance const

  //Global constant is read-only
  //serialNum = 'hff; //COMPILE ERROR

  //Instance const can only be assigned in the
constructor.
  //packetID = 'h ffff; //COMPILE ERROR

  //constructor
  function new;
    packetID = 'h4567;
    //instance const assignment in the constructor
  endfunction

  function void disp(input int packetID);
    $display("From packet");
    $display("\t serialNum = %h",this.serialNum);
    $display("\t packetID = %h",this.packetID);
  endfunction

endclass

class eth_packet extends packet;

  function new;
    super.new;
    //Instance constant cannot be re-initialized.
    //packetID = 'h ffff; //COMPILE ERROR
  endfunction

  function void eth_disp;
    super.disp(packetID);
  endfunction

endclass

module class_TOP( );
  initial begin
```

```

    eth_packet e1;
    e1 = new( );
    e1.eth_disp;

end
endmodule

```

*Simulation log:*

From packet

```

serialNum = 00001234
packetID = 00004567
V C S   S i m u l a t i o n   R e p o r t

```

In this example, we declare two constants. The global constant “serialNum” (“`const int serialNum = 'h1234;`”). The reason this is considered global is because it is initialized directly during its declaration. Once initialized, it cannot be overwritten. In other words, it is a read-only variable. We also declare an instance constant “packetID.” The reason this is considered an instance constant is because it is not initialized during its declaration. Its initial value can only be assigned in the class constructor (`new()` function).

Following code that tries to reassign a value to the global constant “serialNum” will give a compile error:

```

//Global constant is read-only
serialNum = 'hff; //COMPILE ERROR

```

The compile error will be as follows (Synopsys – VCS):

Error-[SE] Syntax error

Following verilog source has syntax error :

```

"testbench.sv", 7: token is '='
    serialNum = 'hff; //COMPILE ERROR

```

Similarly, if we try to assign a value to the instance constant outside of the constructor, we will get a compile error:

```

//Instance const can only be assigned in the
constructor.
packetID = 'h ffff; //COMPILE ERROR

```

Here is the compile error for above code from Synopsys – VCS:

Error-[SE] Syntax error

Following verilog source has syntax error :

```

"testbench.sv", 7: token is '='
    serialNum = 'hff; //COMPILE ERROR

```

We assign a value to the instance constant “packetID” in the constructor of the class packet.

In the class “eth\_packet,” we try to assign a value to the instance constant “packetID” from the class’ constructor (new()). This is not allowed either, since the value of “packetID” was assigned in the constructor of the class “packet.” So, the following code will give a compile error:

```
function new;
    super.new;
    //Instance constant cannot be re-initialized.
    packetID = 'h ffff; //COMPILE ERROR
endfunction
```

Compile error from Synopsys – VCS:

Error-[IIIC] Invalid Initialization  
testbench.sv, 30

Invalid initialization of instance constant: 'packetID' cannot be initialized more than once in constructor. There is a potential re-initialization at

statement : this.packetID = 'h0000ffff;

Previous at: testbench.sv,14

Source info: this.packetID = 'h00004567;

Note that “packetID” can be assigned either in the constructor of class “packet” or in the constructor of class “eth\_packet” but not both.

We then instantiate (in module class\_TOP) “eth\_packet” and invoke its display function to get the simulation log.

Note also that in the above example, we declared the global constant outside of the class constructor. Only the instance constant was initialized in the class constructor. However, you can declare and initialize the global constant in the class constructor as well (run-time global constant), as shown in the example below:

```
class packet;
    const int packetID; //Instance const

    //constructor
    function new (num);
        //global constant in the constructor
        const int serialNum = num;

        //instance const in the constructor
        packetID = 'h4567;
    endfunction
endclass
```

```

class eth_packet extends packet;
    int num;

    function new ();
        super.new ('h 1234);
    endfunction
endclass

module class_TOP( );
    initial begin
        eth_packet e1;
        e1 = new();
    end
endmodule

```

## 8.15 Class Scope Resolution Operator ( :: ) and “extern”

We saw in earlier chapter on “package” how a scope resolution operator is used to access members of a package. This section describes the scope resolution operator ( :: ) as applied to a class.

The class scope resolution operator ( :: ) is used to specify an identifier defined within the scope of a class. It uniquely identifies a member of a particular class. For example, if `packet` is a class that has a *static* public property (or method) named `my_packet`, then you can access `my_packet` from outside the class as `Packet::my_packet`. Classes and other scopes can have the same identifier which may create a namespace collision if referred to without a scope resolution operator. You can access class methods (that are declared “extern”) from outside the class as well.

It is worth noting the difference between the class scope resolution operator ( :: ) and object member reference operator ( . ). For instance, if `mc` is an object instance of `My_class`, `mc.my_x` refers to the member `my_x` of the instance “`mc`,” whereas `My_class::my_x` refers to the member of the class `My_class`.

Here is the syntax:

```
class_type :: identifier
```

The left operand of the operator ( :: ) is a class-type name (or a package name, as we saw in the previous chapter – and for the sake of completeness – it can also be a covergroup type, coverpoint type, “cross” name, typedef name). The right-hand side of the operator should be an identifier like a variable or method name.

The :: operator allows access to *static* members (class properties and methods) from outside the class, as well as access to public or protected elements of a super-class from within the derived classes. Note that *you cannot access non-static*

*members from outside the class using the class scope resolution operator*, but you can indeed access non-static members in a derived class using the class scope resolution operator.

Here is an example to illustrate this point:

```
class packet;
    bit [31:0] addr; //non-static var
    static bit [31:0] id; //static var
    typedef enum {RED, GREEN, BLUE} RGB;
        //typedef enum is static by default

    extern static function void display (int a, b);
endclass

static function void packet::display (int a, b);
    $display("packet values a=%0d b=%0d",a,b);
endfunction

class eth_packet extends packet;

    function new;
        packet::addr = 'hff;
            // Can access non-static members in derived class
        $display("packet addr = %h", packet::addr);
    endfunction

endclass

module sro_class;

    packet::RGB r1; //scope resolution operator for typedef
    int id=10;

    initial begin
        packet p;
        eth_packet ep;

        p = new( );
        ep = new ( );

        $cast(r1,1);
        $display("%s",r1);
```

```

packet::id = 20;
packet::display(packet::id, id);

//packet::addr = 'hff; //COMPILE ERROR
//Can't access non-static members outside the class
end
endmodule

```

*Simulation log:*

packet addr = 000000ff

GREEN

packet values a=20 b=10

### V C S   S i m u l a t i o n   R e p o r t

In this example, in class “packet,” we declare two properties, one static (“id”) and another non-static (“addr”). We also declare a typedef enum which is a static element by default. We also declare an *extern* function “display.” The “extern” function means only the prototype header is declared in the class and its implementation is defined outside the class. Such “extern” method can be accessed using the class resolution operator as we see in the example above.

In the derived class, “eth\_packet,” we use the class resolution operator :: to access “addr” of class “packet.” This is allowed even though “addr” is non-static. This is because we are accessing “addr” from a derived class (“eth\_packet”).

In the module sro\_class, we declare “int id = 10.” This “id” is different from the one declared in class “packet.” We then access the “id” from class packet using the resolution operator :: (packet::id = 20;). This way the “id” from module “sro\_class” does not conflict with the “id” from class packet. We also access the “extern” method “display” of the class “packet” from the module. This method must also be declared static in the class “packet” – else you will get a compile error.

We also declare “r1” of type RGB using the scope resolution operator (packet::RGB r1;). Note that a typedef is a static element by default.

When we try to access “addr” from module sro\_class, we get a compile error. This is because “addr” is non-static and you cannot use class resolution operator on non-static members outside the class. So, if you did the following, in module sro\_class:

```
packet::addr = 'hff;
```

you will get the following compile error (Synopsys – VCS):

Error-[SV-IRTAV] Illegal reference to automatic variable

testbench.sv, 34

"\$unit::packet::addr"

Hierarchical reference to automatic variable 'addr' is not legal.

Same goes for the method “display.” If it was not declared static and you try to access it from outside the class, you will get a compile error (Synopsys – VCS):

```
Error-[ISRF] Illegal scoped reference found
testbench.sv, 36
"packet::display"
Scoped reference to the non-static class task/function 'packet::display' is
not allowed.
```

The simulation log shows the results of three different display statements.

To recap, the scope resolution operator enables the following:

- Access to static public members (methods and class properties) from outside the class.
- Access to public or protected or non-static members of a superclass from the derived class.
- Access to all static elements of a class:
  - Static properties
  - Static methods
  - Typedefs
  - Enumeration
  - Structures
  - Unions
- Scope resolution operator also allows you to reference items in a package (Chap. 7).

A note on “extern”:

As we saw in the previous example, we were able to move a method definition out of the body of class declaration. We declare the method prototype (task or a function) and the full argument specification with the keyword *extern* in the prototype declaration. The *extern* qualifier indicates that the body of the method (i.e., its implementation) is to be found outside the class declaration. Outside the class declaration, we define the full method and tie the method back to its class using the class resolution operator (::). For example:

```
class packet;
  //function prototype
  extern static function void display (int a, b);
endclass

//function body outside the class
static function void packet :: display (int a, b);
  $display("packet values a=%0d b=%0d",a, b);
endfunction
```

An out-of-block declaration must be declared in the same scope as the class declaration. Also, you cannot provide more than one out-of-block declaration for a given “extern” method. An out-of-block method declaration will be able to access all declarations of the class in which the prototype is declared.

## 8.16 Parameterized Class

As we have seen under the chapter on “module” (Chap. 9), you can parameterize a SystemVerilog module. Similarly, you can parameterize a class. If you are going to use a generic base class with many different values/types, you should declare a class with parameters. It would be much easier to write a generic class which can be instantiated in multiple ways to achieve, for example, different array sizes or different data types. This avoids the need to rewrite code for specific features like size or type and instead allow a single specification to be used for different objects.

Parameters are like constants that are local to the specified class. You can have default value for each parameter (highly recommended) which can be overridden during class instantiation. Parameters represent a value or a data type. The compiler evaluates parameters as part of its elaboration and code generation phase before the simulation starts. In other words, we cannot change a parameter once the simulation starts.

A combination of a generic class and actual parameter values is called a *specialization*.

Here is the syntax:

```
class <class_name> #(<parameters>);
```

For example:

```
class mClass #(int DATA = 32);
```

Then you can override these parameters during instantiation of the class:

```
<class_name> #(<parameters>) <class_instance_name>;
```

For example:

```
mClass #(64) mClassInst;  
mClass #(.DATA(64)) mClassInst;
```

There are two types of parameters that can be declared: value parameter to pass a value to the class and type parameter to pass a data type to the class.

One quick note from a methodology point of view. Make the parameter name upper case to show it is not a variable.

### 8.16.1 Value Parameters

Here is an example of how parameters are used to pass values to the class:

```

module top;

class packet #(parameter int A_WIDTH = 32, D_WIDTH = 32);
    //parameter keyword is optional
    bit [A_WIDTH-1:0] address;
    bit [D_WIDTH-1:0] data      ;

    function new();
        address = 'hff;
        data     = 'h11;
    endfunction
endclass

initial begin
    packet #(.A_WIDTH(64), .D_WIDTH(16)) p1;//override
param values
    packet #(16,8) p2; //Override param values
    packet p3; //default values of parameters

    $display ("p1.address = %0d bits", $bits(p1.address));
    $display ("p1.data = %0d bits", $bits(p1.data));

    $display ("p2.address = %0d bits", $bits(p2.address));
    $display ("p2.data = %0d bits", $bits(p2.data));

    $display ("p3.address = %0d bits", $bits(p3.address));
    $display ("p3.data = %0d bits", $bits(p3.data));
end
endmodule

```

*Simulation log:*

p1.address = 64 bits  
 p1.data = 16 bits  
 p2.address = 16 bits  
 p2.data = 8 bits  
 p3.address = 32 bits  
 p3.data = 32 bits

V C S   S i m u l a t i o n   R e p o r t

We define class “packet” with parameters A\_WIDTH and D\_WIDTH each with a default value of 32. We use these parameters to size the variables “address” and “data.”

In the initial block, we instantiate the class “packet” twice with different parameter values and once keeping the default parameter values as is. The \$display statements show how the widths of “address” and “data” change with the parameter override with each instance of the class.

### ***8.16.2 Type Parameters***

Let us look at a simple example showing how to parameterize a class with “data type”:

```
module top;

    //parameterize data type
    class packet #(parameter type I = int);
        I data; //data of type I (default 'int')
    endclass

    initial begin
        //Instantiate 'packet' with different data type overrides.
        packet #(bit[3:0]) p1; //override with data type
        'bit[3:0]
        packet p2; //default data type 'int'
        packet #(real) p3; //override with data type 'real'

        $display ("p1.data Type = ", $typename(p1.data));
        $display ("p2.data Type = ", $typename(p2.data));
        $display ("p3.data Type = ", $typename(p3.data));
    end
endmodule
```

*Simulation log:*

p1.data Type = bit[3:0]

p2.data Type = int

p3.data Type = real

VCS Simulation Report

In class “packet,” we declare a parameter “type I = int.” This is a data-type parameter. The type name is “I” and its default value is “int.” This type can be overridden during instantiation of the class.

We declare “data” of type ‘I’ (which means ‘int data’ – since “int” is the default value of “I”).

In the initial block, we instantiate “packet” with different “type” overrides. These overridden types will determine the “type” of “data” in class “packet.” The simulation log shows the overridden “type” with each instantiation.

Let us look at the same example but add a method (function) to the model and see how different data-type overrides affect the function result:

```

module top;

//class packet #(parameter type I = int);
//keyword 'parameter' is optional
class packet #(type I = int);

    I data; //data of type I (default 'int')

    //returns of type 'I'; with function input of type 'I'
    function I mult (I a);
        return data * a;
    endfunction

endclass


initial begin
    packet #(bit[3:0]) p1 = new ( );
    packet p2 = new ( );
    packet #(real) p3 = new ( );

    $display ("p1.data Type = ", $typename(p1.data));
    p1.data = 4'b1000;
    $display ("p1.mult = %b\n", p1.mult(4'b1000));

    $display ("p2.data Type = ", $typename(p2.data));
    p2.data = -12;
    $display ("p2.mult = %0d\n", p2.mult(2));

    $display ("p3.data Type = ", $typename(p3.data));
    p3.data = 2.34;
    $display ("p3.mult = %0.3f", p3.mult(0.1));

end

```

```

    endmodule

Simulation log:
p1.data Type = bit[3:0]
p1.mult = 0000

p2.data Type = int
p2.mult = -24

p3.data Type = real
p3.mult = 0.234

```

### V C S   S i m u l a t i o n   R e p o r t

This is an identical example as the one before, except that we have added a “function” called “mult” to the model. The function returns of type “I” (“I” is the type parameter in class “packet” with default value “int”) and also takes an input of type “I.”

In the “initial” block, we instantiate class “packet” with different “type” overrides and display the overridden “type” (using \$typename). We assign, for each instance, data value that is compatible with the “type” of that instance and invoke the function “mult” to do multiplication with the provided data value.

Note that the first instance “p1” has the data value “bit[3:0]” meaning it is a 4-bit data type. Hence, when we assign “p1.data = 4'b1000” and multiply it with “4'b1000,” the result is 4'b0000, since there is no overflow bit in “bit[3:0].” This proves that the default data type “int” of class packet was overridden with type “bit[3:0]” of instance p1.

Other multiplication results show the function (“mult”) return values with the correct data types.

*Note that any “type” can be supplied as a parameter, including user-defined types such as “class” or “struct.”*

### 8.16.3 Parameterized Class with Static Properties

This section is derived from (Verification, n.d.). As we have discussed in the prior sections, static properties are allocated in memory at elaboration time and shared across all instances of the class. They are initialized before time 0. But it is slightly different when it comes to using them in a generic class with parameters. The static class properties do not get allocated unless their class is specialized. Let us see this via the following example:

```

// Class with parameters
class with_param #(type T = int);
    static T static_with_p;
endclass

```

```

// Class without Parameters
class without_param;
    static int static_wo_p;
endclass

module top;

initial begin

    $display("static_wo_p = %0d", without_param :: static_wo_p);
    $display("static_with_p = %0d", with_param :: static_with_p);

end
endmodule: top

```

In this example, we declare two classes. One with parameters (`class with_param`) and one without (`class without_param`). In each class we declare a static property. Now, we try to access these static properties (via `$display`), and we get different results. When we access property “`static_wo_p`” using the class resolution operator – without instantiating the class – the compiler is happy. In other words, we can see that the static property “`static_wo_p`” is accessed and initialized using the class definition – without any object construction. But trying to access “`static_with_p`,” the same way will give a warning from the simulator (some simulators may give an error). This warning is from Synopsys – VCS:

Warning-[PCSRMIO] Class scope used outside of class

testbench.sv, 59

"with\_param::static\_with\_p"

An unspecialized class scope '::' reference was seen. This refers to the generic class, and may only be used inside the class 'with\_param'. To access a static member of the default specialization outside the class 'with\_param', use 'with\_param#( )::<staticMember>' instead. This will be an error in a future release.

However, if we run the same code with the following modification:

```

$display("static_with_p = %0d", with_param #( ) :: static_with_p);

```

The error goes away. So, here the specialization is done using the default parameter value. *This shows that the static properties do not get allocated until the generic parameterized class is specialized.*

Let us now analyze what happens in terms of static properties when the generic parameterized class is specialized in different ways and with multiple instances. Note that every specialization has a unique set of static properties. Let us see the following example:

```

module param_static;

class with_param #(type T = int);

    static T counter = 2;

    function new;
        counter++;
    endfunction: new
endclass: with_param

class with_param_extend extends with_param #(real);
endclass: with_param_extend

typedef with_param #(byte) s_byte;

s_byte S1 = new();
s_byte S2 = new();
with_param S3 = new();
with_param #(bit[2:0]) S4 = new();
with_param_extend S5 = new();

initial begin
    $display ("Counter value of S1 instance = %0d", with_param
#(byte)::counter);
    $display ("Counter value of S2 instance = %0d", s_byte::
counter);
    $display ("Counter value of S3 instance = %0d", with_param
#()::counter);
    $display ("Counter value of S4 instance = %0d", with_param
#(bit[2:0]):counter);
    $display ("Counter value of S5 instance = %0f", with_param_extend::counter);
end

```

```
endmodule: param_static
```

*Simulation log:*

```
Counter value of S1 instance = 4
Counter value of S2 instance = 4
Counter value of S3 instance = 3
Counter value of S4 instance = 3
Counter value of S5 instance = 3.000000
```

### V C S   S i m u l a t i o n   R e p o r t

In this example, we declare two classes: “with\_param” which is parameterized and has a “static” property (counter) and a second class “with\_parameter\_extend” which extends the class “with\_param.” We have added the static property, i.e., the “counter” as part of the generic class “with\_param.” But as we saw in the previous example, there is no allocation for “counter” until there is a reference that specializes the class “with\_param.”

We then typedef “with\_param” class to s\_byte (`typedef with_param #(byte) s_byte;`). S1 and S2 create two unique specializations of the class “with\_param” and thus creates two instances of the static property “counter.” We allocated the first “counter” upon specializing “with\_param” with a byte (instances S1 and S2) and allocated the second “counter” upon specializing “with\_param” with its default type “int” (instance S3).

S1 and S2 are two instances of specialized “s\_byte” type and constructs two objects that share the same static property “counter.” So, each initialization increments the “counter” by 1 with the resulting value 4 – as shown in the simulation log.

Instance S3 creates an object of class “with\_param” with the default parameter value “int.” It uses the existing specialization and the “counter” associated with it is left at value 3 – as shown in the simulation log.

We then declare another variable S4 and create another unique specialization of generic class “with\_param” of type “bit[2:0].” Again, the “counter” value after this initialization will be 3.

Finally, we instantiate “S5” for the extended class “with\_param\_extend” which extends with\_param with type = real. Its “counter” value will also be incremented by 1 resulting in 3.000000 as the final value, as shown in the simulation log.

So, the point, as stated earlier, is that every time a *unique* specialization is created for a parameterized class, a new instance of the static property is initialized.

#### 8.16.4 Extending Parameterized Class

You can indeed extend a parameterized class into another parameterized class. Here are some examples:

```

class class1 #(type T = int);
...
endclass

class class2 #(type P = real) extends class1;
// 'class2' has parameter 'P = real' while 'class1' is
left with
// default parameter 'T = int'

class class3 #(type P = real) extends class1 #(integer);
// 'class3' has parameter 'P = real' while class 'class1'
// overrides with parameter 'integer'

class class4 #(type P = real) extends class1 #(P);
// parameter 'T' of class1 is now 'P'. Class4 extends
the base
// class 'class1' using the parameterized type 'P =
real' which
// is the parameter of the class 'class4'.

```

## 8.17 Difference Between Class and Struct

So, we have now seen both a struct and a class. A “struct” represents a collection of same or *different* data types that can be referenced as a whole or by the individual data types that make up the structure. A class is a user-defined type. It includes data/variables (known as class properties) and subroutines (known as class methods) that work on the class properties.

They seem quite similar. But there are differences that you need to be aware of. Here they are at a high level:

**Re-usability:** As classes form the basic framework, they can be re-used; structs, however, are individual elements with specific properties, so they cannot be re-used. You can derive new classes from other classes, but you cannot derive new structs from other structs.

**Inheritance:** Classes can be further inherited to form subclasses, but structs cannot utilize inheritance. Class can create a subclass that will inherit parent class’ properties and methods, whereas structure does not support inheritance.

**Memory Allocation:** A struct is like a variable in that it always consumes memory. A class does not consume any memory until you allocate the memory with a call to `new()`.

**Member Variable Initialization:** You can initialize structs the same way you initialize variables. But, there is no constructor and therefore no automatic initialization. Classes can be initialized in the constructor which is called automatically when `new()` is called.

# Chapter 9

## SystemVerilog “module”



**Introduction** This chapter explores the nuances of SystemVerilog “module,” a fundamental building block of SystemVerilog. It includes module headers (ANSI or non-ANSI), module parameters, localparams, nested modules, etc.

SystemVerilog module (hereafter referred to simply as module) is a fundamental building block (along with a program, a checker, a class, a package, and an interface) of the language. Everything starts, hierarchically, from a module. It encapsulates data, functionality, timing, and design hierarchy. A module can have high-level procedural blocks (tasks, functions, always block, initial blocks, etc.) or gate-level primitives (and, or, xor, tri, trireg, etc.) or both. It can also have timing blocks (specify blocks). It can be used for design or for high-level verification blocks or both under the same module. It can contain SystemVerilog Assertions and Functional Coverage constructs as well. And as aforementioned, it can instantiate other modules to create a module hierarchy.

The module definition is enclosed between the keywords *module* and *endmodule*. The identifier that follows the keyword “module” is the name of the module.

In general, some of the constructs that a module can contain include the following:

- Ports, with port declaration.
- Procedural blocks.
- Specify blocks.
- Generate blocks.
- Class definitions and instantiations of class objects.
- Instantiations of other modules, programs, checkers, and primitives.
- Task and function definitions.
- Continuous assignments.

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_9](https://doi.org/10.1007/978-3-030-71319-5_9)) contains supplementary material, which is available to authorized users.

- Data declarations, such as nets, variables, structures, and unions.
- Constant declarations.

Here is an example:

```
module mymod (ina, inb, outa, outb);
    output outa, outb;
    input ina, inb;

    and a1 (outa, ina, inb);
    assign outb = ina && inb;

    lowermodule lm1 (...);

initial begin
    .....
end

always begin
    .....
end

function myfunc;
    .....
endfunction

task mytask;
    .....
endtask

specify
    ...
endspecify

endmodule
```

## 9.1 Module Header Definition

The module header contains the following:

- The name of the module
- The port list of the module
- The direction and size of each port (ANSI or non-ANSI style as we will soon see)

- The parameter constants of the module
- The type of data passed with the ports
- The default lifetime (static or automatic) of subroutines defined within the module

First, let us look at the styles of module headers. There are two styles in which module header ports can be defined: the ANSI style and the non-ANSI style.

Let us look at each one.

### **9.1.1 ANSI-Style Module Header**

In this style, you declare the I/O direction of the ports directly in the module header itself as shown below. Each port declaration provides the complete information about the port. The port's direction, width, net or variable, and signedness are completely described. The port identifier cannot be redeclared, in part or in full, inside the module body:

```
module test0 (input a, output logic b, wire c);
    logic b; //ERROR
    logic a; //ERROR
endmodule
```

As you notice, we have declared the I/O direction (along with type) in the module header itself. Note that once the module I/O is declared in the header, they cannot be redeclared after the module header. The error you will get will look like the following (Cadence-Incisive):

```
irun: 15.20-s038: (c) Copyright 1995-2017 Cadence Design Systems, Inc.
logic b; //ERROR
|
ncvlog: *E,DUPIDN (testbench.sv,5|8): identifier 'b' previously declared
[12.5(IEEE)].
logic a; //ERROR
|
ncvlog: *W,ILLPDX (testbench.sv,6|8): Multiple declarations for a port not allowed
in module with ANSI list of port declarations (port 'a') [12.3.4(IEEE-2001)].
```

We will discuss non-ANSI-style module header in the next section, but here's a high-level definition to enable the discussion in this section.

In the non-ANSI-style module header, separate declarations are used in the `module_port_list`. The direction of the ports is defined in the module declaration list not in the module header itself (i.e., the module I/O direction is declared outside of the module header).

Keeping that in mind, note that you can mix ANSI- and non-ANSI-style ports in a module header. Here is an example:

```
module test0 (input a, output logic b, c, d); //OK
endmodule
```

In this case, the direction and type of ports “c” and “d” are not specified. So, they look like non-ANSI-style port declarations. But since the port direction/type is declared for the previous port “b,” the ports “c” and “d” take on the same direction and type. Here is an example:

```
module test0 (input a, output logic b, c, d); //OK
initial begin
    b = 0;
    d = 0;
    $display("b=%0d c=%0d d=%0d",b,c,d);
end
endmodule
```

Here “c” and “d” are now of direction “output” and type “logic.” The model drives “b” and “d” since they are outputs but does not drive “c” to show that it will take on the default value x (unknown) since “c” is of type logic.

*Simulation log:*

b=0 c=x d=0

V C S   S i m u l a t i o n   R e p o r t

BUT the following will give an error because “c” and “d” have already taken on the “output” direction and “logic” type from port “b.” So, redeclaring ports “c” and “d” will give errors (even though they look like non-ANSI-style port declarations in the header). First the code and then the errors:

```
module test0 (input a, output logic b, c, d); //OK - line 1
    inout c; //ERROR - line 2
    output d; //ERROR - line 3
initial begin
    b = 0;
    d = 0;
    $display("b=%0d c=%0d d=%0d",b,c,d);
end
endmodule
```

Errors are as follows (Synopsys – VCS):

Error-[IDPD] IO declaration previously declared

The variable 'c' is already defined as a port.

"testbench.sv", 2

Source info: inout c; //ERROR

Error-[V2KAOP] Mixed parameter declaration style

"testbench.sv", 3

Source info: output d; //ERROR

It is illegal to have both ANSI and old style parameter declarations in one module.

Error-[IDPD] IO declaration previously declared

The variable 'd' is already defined as a port.

"testbench.sv", 3

Source info: output d; //ERROR

### 9.1.2 ANSI-Style first Port Rules

If the *first* port is non-ANSI style, then the rest must be non-ANSI style. You cannot have ANSI-style ports following the first port as non-ANSI style. If the direction, port kind, and data type are all omitted for the first port in the port list, then all ports will be assumed to be non-ANSI style, and port direction and optional-type declarations will be declared after the port list.

For example:

```
module test0(a, output logic b, c, d); //ERROR
endmodule
```

You will get following error (Synopsys – VCS):

Error-[SE] Syntax error

Following verilog source has syntax error :

```
Keyword 'output' cannot be used as port connection name of module
'test0'
"testbench.sv", 1: token is 'output'
module test0(a, output logic b, c, d);
```

Let us further explore how the *first* port in ANSI-style declaration works. Here are the rules:

- If the direction is omitted, the port will refer to “inout.”
- If the data type is omitted, the port type will default to “logic.”

With these rules in mind, here are examples of how *first* port will be interpreted (courtesy (SystemVerilog – LRM)):

```
module mh0 (wire x); // inout wire logic x
module mh1 (integer x); // inout wire integer x
module mh2 (inout integer x); // inout wire integer x
module mh3 ([5:0] x); // inout wire logic [5:0] x
module mh4 (var x); // ERROR: direction defaults to inout,
                    // which cannot be var
module mh5 (input x); // input wire logic x
```

```

module mh6 (input var x); // input var logic x
module mh7 (input var integer x); // input var integer x
module mh8 (output x); // output wire logic x
module mh9 (output var x); // output var logic x
    module mh10(output signed [5:0] x); // output wire logic
signed [5:0] x
        module mh12(ref [5:0] x); // ref var logic [5:0] x
        module mh13(ref x [5:0]); // ref var logic x [5:0]

```

### 9.1.3 ANSI-Style subsequent Port Rules

So, what about the subsequent ports (i.e., ports after the first port)? Here are the rules:

- If the direction, port kind, and data type are all omitted, then they will be inherited from the previous port.
- If the direction is omitted, it will be inherited from the previous port.
- If the data type is omitted, it will default to logic.

Based on these rules, here are examples of *subsequent* ports (courtesy (SystemVerilog – LRM)):

```

module mh14(wire x, y[7:0]);
// inout wire logic x
// inout wire logic y[7:0]

module mh15(integer x, signed [5:0] y);
// inout wire integer x
// inout wire logic signed [5:0] y

module mh16([5:0] x, wire y);
// inout wire logic [5:0] x
// inout wire logic y

module mh17(input var integer x, wire y);
// input var integer x
// input wire logic y

module mh18(output var x, input y);
// output var logic x
// input wire logic y

module mh19(output signed [5:0] x, integer y);
// output wire logic signed [5:0] x
// output var integer y

```

```

module mh20(ref [5:0] x, y);
// ref var logic [5:0] x
// ref var logic [5:0] y

module mh21(ref x [5:0], y);
// ref var logic x [5:0]
// ref var logic y

```

### **9.1.4 Non-ANSI-Style Module Header**

In the non-ANSI-style module header, separate declarations are used in the module\_port\_list. The direction of the ports is defined in the module declaration list not in the module header itself (i.e., the module I/O direction is declared outside of the module header). It looks like the following:

```

module_name (port_list);
port direction and size declarations;
port type declarations;
parameter declarations;

```

Here is an example:

```

module test(a, b, c, d, e, f, g, h);

input [7:0] a; //default unsigned
input [7:0] b;
input signed [7:0] c; //signed
input [7:0] d;

output [7:0] e; //default unsigned
output [7:0] f;
output signed [7:0] g;
output [7:0] h;

endmodule

```

In the module header, we declare eight ports, but their direction or type is not specified in the header itself. The direction and type are specified after the header. Note that by default all ports are unsigned except when explicitly defined as signed.

Here is a bit more interesting example:

```

module complex_ports ( {c, d}, .e(f), i[31:0]);

    output logic c, d;
    output var [31:0] i;
    //input e;
    //ERROR : 'e' is not visible in module 'complex_ports'
    input f; //Correct

    initial begin
        i = 0; c = 1; d= 1;
    end

    endmodule

module top;

    //wire g; //WARNING
    //1-bit expression is connected to 2-bit port
    wire [1:0] g; //Correct
    wire k, h;
    wire e;
    var [31:0] i;

    complex_ports cpl ({k, h} , e); //WARNING
        //fewer port connections than the module definition"
    complex_ports cp (g , e, i); //Correct

    endmodule

```

There are a few things going on in this example.

First, the module header declaration has a port that is concatenation of two variables ( $\{c, d\}$ ). This is perfectly legal. This helps you connect a larger width (bus) vector when the module is instantiated.

We also declare the *type* of the I/O ports directly with the direction of the ports. Here we declare output “c, d” of type “logic” and output “i” of type “var.” This way you do not have to redeclare the output or input to give them a *type*. For example:

```
output logic c, d;
```

can also be declared as:

```
output c, d;
logic c, d;
```

Next, we connect the I/O “.e (f)” directly in the port list. Yes, you can connect the port of the module “complex\_ports” directly (name-based connection) to the wire “e” of the module “top.” But note that the way name-based connections work is that, in .e(f), the .e is the name of the port/wire in the *instantiating* module (module “top” in our case) and “f” is the name of the port/wire in the *instantiated* module (module “complex\_ports” in our case). Hence, if you try to declare “e” as an input or output in module “complex\_ports,” you will get an error because “e” is visible only in module “top.”

In module “top,” we connect “wire [1:0] g” to output {c,d} of module complex\_ports. This way a 2-bit wire is directly connected to the concatenated port {c, d} of module complex\_ports.

Note that if you connect a smaller number of ports or incorrect width of ports, you will get a compiler warning, as noted in the source code.

Here is another example showing how part-selects vectors can be declared in a module header and connected in the instantiating module.

```
module test(a, b[7:0], b[15:8],c, d);
//two ports with same name but with different part selects

    input a;
    input [7:0] c, d;

    //single vector declaration for 2 part-select ports
    output [15:0] b;
    logic [15:0] b;

    initial begin
        b[7:0] = 'hff;
        b[15:8] = 'h01;
    end

endmodule


module top;
    wire [15:0] bTop;
    logic a;
    logic [7:0] c, d;

    test test1(a, bTop[15:8], bTop[7:0], c, d); //OK
    //test test1(.a(a), .b[7:0](bTop[15:8]), .b[15:8]
(bTop[7:0]), .c(c), .d(d)); //ERROR
```

```
//test test1(.a(a), .b(bTop[15:8]), .b(bTop[7:0]), .c(c),
.d(d)); //Warning

endmodule
```

In this example, we take the bus/vector “output [15:0] b;” and split it into two ports in the module header declaration, each is a part-select of the bus “b.” Yes, you can have two ports with the same name but different part-selects. When we instantiate “test” in module “top,” we can again split an internal bus of module “top” and connect it to the part-selects of the module “test.” But note that the following instantiation will give an error:

```
//test test1(.a(a), .b[7:0](bTop[15:8]), .b[15:8](bTop[7:0]),
.c(c), .d(d)); //ERROR
```

This is a syntax error. You cannot have a name-based connection with the part-selects of module “test” as done here. Here is the error you will get (Synopsys – VCS):

#### Error-[SE] Syntax error

Following verilog source has syntax error :

"testbench.sv", 27: token is '['

```
test test1(.a(a), .b[7:0](bTop[15:8]), .b[15:8](bTop[7:0]), .c(c), .d(d));
//ERROR ^
```

So, you try to do the following. We take the entire bus “b” of module “top” and connect it twice to the part-selects of module “test.” That is a no-no. It is syntactically correct, but there is a bus width mismatch, and you will get a warning. The compiler also thinks that you have connected the port “b” twice and you will get a warning for that as well:

```
//test test1(.a(a), .b(bTop[15:8]), .b(bTop[7:0]), .c(c),
.d(d)); //Warning
```

Here is the warning you will get (Synopsys – VCS):

#### Warning-[DPIMI] Duplicate port in module instantiation

testbench.sv, 28

"test test1( .a (a), .b (bTop[15:8]), .b (bTop[7:0]), .c (c), .d (d));"

Port 'b' is connected more than once for instance 'test1' of 'test'.

Extra connection will be ignored.

#### Warning-[TFIPC] Too few instance port connections

testbench.sv, 28

top, "test test1( .a (a), .b (bTop[7:0]), .c (c), .d (d));"

The above instance has fewer port connections than the module definition.

## 9.2 Default Port Values

You can indeed have default values assigned to the inputs (not the outputs) of a module. These default values are constant expressions and are evaluated in the scope of the module where they are defined. Note that *defaults can only be specified for ANSI-style declaration.*

When the module is instantiated, input ports with default values can be omitted from the instantiation, and

the compiler will insert the corresponding default values.

Here is an example of how default input values get used and how they are overwritten:

```

module lowMod (
    output logic [7:0] dataout,
    input A1 = 1'b0,
    input [7:0] datain = 'h aa);

    assign dataout = datain;

    always @ (A1 or datain)
        $display ("A1 = %0d datain = %0h", A1, datain);

endmodule


module topMod (
    output logic [31:0] dataout,
    input [7:0] datain);

    parameter logic [7:0] tModData = 8'hbb;

    lowMod lmod0 (dataout[31:24], 1, 8'hFF);
    // Constant literal overrides defaults in lowMod definition
    // In lowMod: A1 = 1; datain = 8'hFF

    lowMod lmod1 (dataout[23:16]);
    //Omitted port for A1 and datain, default parameter
values of 0
    //and 8'aa are used in lowMod
    //In lowMod: A1 = 0; datain = 8'haa

    lowMod lmod2 (dataout[15:8], 1'bz, tModData);

```

```

// The parameter value 8'hbb and 1'bz from the
// instantiating scope is used
// In lowMod: A1 = 1'bz; datain = 8'hbb

endmodule

```

*Simulation log:*

A1 = z datain = bb  
A1 = 0 datain = aa  
A1 = 1 datain = ff

### V C S   S i m u l a t i o n   R e p o r t

In this example, in the module “lowMod,” we declare two inputs “A1” and “datain,” each of which is assigned a default value. In the “topMod” we instantiate “lowMod” three times and each time we either omit connection to these two inputs or assign a value to these inputs.

So, the following code:

```
lowMod lmod0 (dataout[31:24], 1, 8'hFF);
```

will result in “A1” being overwritten by 1 and “datain” being overwritten by 8’h ff. When we display “A1” and “datain” of lowMod, we get the following:

A1 = 1 datain = ff

Similarly, the following code simply does not connect to “A1” or “datain”:

```
lowMod lmod1 (dataout[23:16]);
```

And the two inputs take on their default values, as shown in simulation log:

A1 = 0 datain = aa

Lastly, we assign 1’bz to “A1” and “tModData” (which a constant = ‘h bb) to “datain”:

```
lowMod lmod2 (dataout[15:8], 1'bz, tModData);
```

to get the following in simulation log:

A1 = z datain = bb

## 9.3 Module Instantiation

In this section we will discuss hierarchical instantiation of modules. Note that the top-level modules are not instantiated. Top-level modules are modules that are included in the SystemVerilog source text, but do not appear in any module instantiation statement. There is always at least one top-level module in design hierarchy. A top-level module is implicitly instantiated once, and its instance name is the same as the module name.

Note that parentheses are required for all module instantiations, even when the instantiated module does not have ports.

The following example shows how modules are instantiated and how their ports are connected in the instantiating module:

```

module test0 (input a, output logic b, wire c, wire d);
endmodule

module TOP;
    wire a, b, c, h, i, j;

    test0 test0Inst1 (h, i, j, a);      //position based
    test0 test1Inst2 (.a(h), .b(i), .c(j), .d(a)); //name based
    test0 test0Inst4 (.a, .b, .c);    //implicit name based

    //test0 test0Inst3 (*.);
    //wildcard - ERROR - 'd' port does not exist in TOP

    //wildcard and explicit name based
    test0 test0Inst5 (*.*, .d(h));
endmodule

```

Module test0 is instantiated four times in the module TOP. The first instance connects ports based on the position of each port:

```
test0 test0Inst1 (h, i, j, a);      //position based
```

Here “h” net of TOP is connected to “input a” of test0. Similarly, “i” is connected to “output logic b,” and “j” is connected to “wire c” and “a” is connected to “wire d” in test0.

Next we see name-based connections.

*Name-based connections are highly recommended.* Explicit correspondence of names makes this connection type much less error prone.

*Position-based connections are error prone and not recommended.* Incorrect port order can be time-consuming to detect and debug:

```
test0 test1Inst2 (.a(h), .b(i), .c(j), .d(a)); //name based
```

Here we explicitly connect the nets a, b, c, d of module test0 with the nets h, i, j, a of module TOP. Note that the syntax is that the name specified with “.” is the name in the *instantiated* module, while the name in parenthesis () is the name in the *instantiating* module. Here is the syntax:

```
.name_in_instantiated_module (name_in_instantiating_module);
```

Next, we use implicit name-based instantiation. Again, this is possible because there are nets in module TOP which has the same names as those in the module test0. However, there is no net/variable named “d” in TOP module. So, we can do the following but “wire d” of module test0 will remain unconnected:

```
test0 test0Inst4 (.a, .b, .c); //implicit name based
```

is the same as:

```
test0 test1Inst2 (.a(a), .b(b), .c(c));
```

Next, we use a wildcard to connect the nets. This requires that the names and types of nets in module TOP be the same as the names and types of nets in module test0. We have declared “wire a, b, c” in module TOP which directly corresponds to the nets a, b, c in module test0. BUT, we do not have any net/variable named “d” in module TOP. So, the following line will give an ERROR:

```
test0 test0Inst3 (*.); //wildcard - ERROR
```

To circumvent that, we need to use wildcard along with explicit name-based connection as follows:

```
test0 test0Inst5 (*.*, .d(h)); //wildcard and explicit  
name based
```

is equivalent to:

```
test0 test1Inst2 (.a(a), .b(b), .c(c), .d(h));
```

You can also instantiate a single module in an instantiation statement of multiple modules. For example, two instances of the module “Dflop” can be instantiated as follows:

```
Dflop      df1 (q, d, clk),
           df2 (q1, d1, clk);
```

### 9.3.1 \$root

Also, sometimes there are instance paths that need to be explicitly specified as originating from the top-level module. \$root unambiguously refers to a path emanating from the top-level module. \$root is the root of the instantiation tree. \$root is useful to disambiguate a local path (which takes precedence) from a rooted path. For example, A.B.C can mean the local A.B.C or a top level A.B.C (assuming there is an instance A that contains an instance B at both the top level and in the current module):

```
$root.A.B.C //item C within instance B within instance A
```

The instance name \$root refers to the top of the instantiated design and is used to unambiguously gain access to the top of the design.

## 9.4 Nested Modules

Yes, you can nest a module within a module. This was simply not possible in Verilog. And even now, some simulators do not support it. I personally have not found much use of it. But here it is for the sake of completeness.

When you nest a module within another module, the outer name space is visible to the inner module so

that any name declared there can be used. In the following example, module test0 is defined externally and also embedded within module TOP. Module TOP will inherit the definition of module test0 embedded within it and not the one declared outside of it. This way you can have the same module with different definitions of it in different modules. I strongly suggest you use caution using nested modules.

Here is an example:

```
module test0 (input a, output logic b, wire c, wire d);
    initial begin
        b = 1'b1;
        $display("external test0 b = ",b);
    end
endmodule
```

```

module TOP;
    wire a, b, c;
    logic h, i, j;

    module test0 (input a, output logic b, wire c, wire d);
        initial begin
            h = 1'b0;
            $display("nested test0 h = ",h);
        end
    endmodule

    test0 test1Inst2 (.a(h), .b(i), .c(j), .d(a)); //Uses
nested module test0
endmodule

```

*Simulation log:*

```

run -all;
# KERNEL: nested test0 h = 0
# KERNEL: external test0 b = 1
# KERNEL: Simulation has finished.

```

We have declared module test0 twice, one outside of module TOP and one inside of it (nested). Both have their own initial block. As you can see, both co-exist just fine and each one has its own functionality. The nested module test0 will inherit the nets/variables declared in module TOP as shown in the simulation log.

## 9.5 Module Parameters

Parameters are elaboration time constants. They are useful in modular code development. Code can be written using these parameters which can then be overridden to customize the module for different functionalities. Port declarations can be based on parameter declaration, and the ports can then be resized by overriding these port parameters. Here is an example using non-ANSI-style module header:

```

module BUS (clk, rw, addr, data, enb); //Non-ANSI style header
parameter msb=31, lsb=0;
input [(msb-16):lsb] addr;
inout [msb : lsb] data;
input clk, rw, enb;
endmodule

```

Or you can do the same with ANSI-style module header as shown below:

```

module BUS #(parameter msb = 31, lsb = 0)
    (input clk, rw, enb,
     input [ (msb-16):lsb] addr,
     inout [msb : lsb] data);
endmodule

```

In these two examples, we define two parameters, namely, “msb” and “lsb” to size the addr and data buses. This allows us to then override these parameters when we instantiate “BUS.” That allows us to use the module “BUS” with different configurations.

### 9.5.1 Overriding Module Parameters: defparam

In this section we will see how module parameters can be overridden. There are two ways to override a module parameter. One is to use the *defparam* statement, and other is to use *the module instance parameter value assignment*. Let us first look at parameter override using *defparam*.

Here is a simple example:

```

module tMOD ();
    int i1,i2;
    parameter [2:0] A = 3'h2;
    parameter [2:0] B = 3'h2;

    initial begin
        i1 = A;
        i2 = B;
        $display("i1 = %0h i2 = %0h",i1,i2);
    end
endmodule

module m2;

tMOD tm1 ();

defparam tm1.A = 7;
defparam tm1.B = 3;
endmodule

```

*Simulation log:*

```
i1 = 7 i2 = 3
```

### VCS Simulation Report

In this example, we define two parameters A and B in the module tMOD and assign them constant values. In the module m2, we then override these parameters using defparam statements. The values assigned to A and B in the module tMOD were 3'h2 and 3'h2, respectively. But in module “m2,” we override these parameters with tm1.A = 7 and tm1.B = 3. These new values will now take effect as shown in the simulation log.

#### **9.5.2 Overriding Module Parameters: Module Instance Parameter Value Assignment**

In this method, we use one of the two forms of module instance parameter value assignments: the assignments by ordered list and assignment by name.

We have seen how delay values to gate instances are provided. The idea of overriding by ordered list is the same. Assignment by name is the same as when we use name-based connection in module instance.

Here is an example of position-based value assignment by ordered list, same as above but without the use of defparam. The comment explains how this works:

```
module tMOD ( );
    int i1,i2;
    parameter [2:0] A = 3'h2;
    parameter [2:0] B = 3'h2;

    initial begin
        i1 = A;
        i2 = B;
        $display("i1 = %0h i2 = %0h",i1,i2);
    end
endmodule

module m2;
    //override parameters by position based ordered list
    tMOD #(7,3) tm1 ( );
endmodule
```

*Simulation log:*

```
i1 = 7 i2 = 3
```

### VCS Simulation Report

Similarly, the following is an example of how to override using name based ordered list:

```

module tMOD ( );
    int i1,i2;
    parameter [2:0] A = 3'h2;
    parameter [2:0] B = 3'h2;

    initial begin
        i1 = A;
        i2 = B;
        $display("i1 = %0h i2 = %0h",i1,i2);
    end
endmodule

module m2;
    tMOD #( .A(7), .B(3) ) tm1 ( ); //name-based parameter
override
endmodule

```

*Simulation log:*

i1 = 7 i2 = 3

VCS Simulation Report

### 9.5.3 Localparam

There is also the concept of *localparam* which as the name suggests is local to a module. It cannot be overridden using *defparam* or the ordered list methods as we saw with parameters. Here is an example:

```

module test0 (input a, output b, wire c);
    parameter max=10, min = 5; //defparam will overwrite this

    localparam msb = max*min; //cannot be overridden by
defparam
    // so, make them dependent on 'parameters' which can be
    // overridden from outside the module

    initial begin
        #1; $display(max, min, msb);
    end
endmodule

```

```

module top;
    wire a, b, c;

    test0 test0Inst1(.*);
    defparam test0Inst1.max = 20, test0Inst1.min = 2;
    //defparam test0Inst1.msb = 50; //ERROR - msb is localparam
endmodule

```

*Simulation log:*

```

20      2      40
V C S   S i m u l a t i o n   R e p o r t

```

As we see in the model, we cannot override a localparam from outside the module. But there is a workaround. Make the localparam dependent on parameters. That way you can change the values of parameters from outside the module which in turn will change the value of the localparam. As you see in the simulation log, we change the value of localparam “msb” by changing the values of the parameters “max” and “min.” The comments in the code explain the functionality.

#### 9.5.4 Parameter Dependence

Some more points on parameter dependence. As we saw, we made the localparam dependent on non-local parameters. Let us now look at an example where you have the following dependency:

```
parameter burst_size = 8, line_size = burst_size * 4;
```

It is obvious that if you change the value of “burst\_size” (either by “defparam” or in the instantiation statement for the module), the value of line\_size will change. But note that if you directly change the value of line\_size, it will take on this new value, regardless of the value of burst\_size.

Parameters can also have type dependency on other parameters, including type parameters. For example:

```

parameter index = 1;
parameter [index:0] value = 4;
parameter type T = int;
parameter T p3 = 'hff; //type dependency

```

If parameter “index” changes, the size of parameter “value” changes. If the type parameter T changes, the value of p3 is recomputed. But, for example, if “T” in this example was overridden with a “class” type, the evaluation of p3 would be illegal and would result in an elaboration time error. Here is an example:

```
module tMOD ( );
    parameter index = 1;
    parameter [index:0] value = 4;
    parameter type T = int;
    parameter T p3 = 'hff; //type dependency
endmodule

module m2;

    class mCLASS;
    endclass

    tMOD tm1 ( );
    tMOD #(2,10,real,0) tm2( ); //OK
    tMOD #(2,10,mCLASS,0) tm3( ); //Cannot override with
a 'class'
                                            //type
endmodule
```

In this example, the instance “tm1” of tMOD leaves the parameters to their default values (does not override them). The instance “tm2” overrides the parameters including the type parameter (type T = int) with the “real” type. That is OK. But when instance “tm3” tries to override the type parameter “T” with a “class” type, you will get the following elaboration time error (Mentor – Questa):

```
# ** Error: testbench.sv(6): (vopt-184) Internal Error - Illegal assignment to class..
```

# Chapter 10

## SystemVerilog “program”



**Introduction** This chapter describes a SystemVerilog *program* and its differentiation with SystemVerilog *module*. Among other aspects, it describes how race can be avoided between a testbench and a DUT.

As we saw previously, a module provides a hierarchical scope for design elements. It encapsulates hierarchy of other modules and is well suited for hardware description of a design. In contrast, the *program* was introduced to encapsulate testbench constructs. Emphasis is not on the hardware-detail such as RTL, wires, etc. rather modeling an environment that allows for verification of a design. The *program* construct allows for a clear separation between design module and verification program.

One of the main reasons for using a *program* for a testbench is to avoid race condition between design and verification environments. If you use a *module* for both design and verification, there is an inherent race condition between the two modules (this is not entirely true; you can design your design and testbench such that such race conditions do not occur. Designers use correct rules for module-to-module communication to avoid module to module race).

Such race condition is eliminated when you use a *module* for design and a *program* for testbench. We will see examples of that shortly.

Here are some key points that define a program:

- Provides a race-free interaction between a “program” (testbench) and a “module” (DUT).
- A “program” can contain one or more “initial” and/or “final” blocks. But it cannot contain an “always” block.
- It cannot contain primitives, UDPs, or instances of modules, interfaces, or other programs.

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_10](https://doi.org/10.1007/978-3-030-71319-5_10)) contains supplementary material, which is available to authorized users.

- The program “finishes” (i.e., implicit call to \$finish) as soon as all the threads and all their descendant threads have ended in the program.
- Type and data declarations within a program are local to the program and have static lifetime.
- A program has special execution semantics that avoids race condition in interacting with a “module.” This is a very important feature of a “program.”
- Program blocks can be nested within modules or interfaces.
- Concurrent assertions are allowed in “program” blocks.
- Calling “program” subroutines (functions and tasks) or “program” variables from within design modules are illegal and will result in an error. However, programs are allowed to call subroutines in other programs or in design modules.
- A “program” block can be nested within a module. This way multiple “programs” within the same module can share variables local to that scope. So, following is allowed:

```
module tb;
    logic [7:0] addr;
        program p1;
            ...
        endprogram
        program p2;
            ...
        endprogram
    endmodule
```

The connection between design “module” and testbench “program” uses the same interconnect mechanism used to specify port connections, including interfaces:

- Program port declaration syntax and semantics are the same as those of modules.
- All elements declared within the “program” block will get executed in the reactive region:
  - Statements within a “program” (scheduled in reactive region) that are sensitive to changes in signals in a design “module” (which are scheduled in active region) will avoid race condition because active region is scheduled before the reactive region.
  - Reactive region is one of the last few phases before simulation time advances, and by then all design element statements would have been executed and the testbench (“program”) will see the updated values. Thus, avoiding race condition.

Here is a simple example:

```
program testP (input a, input b, input clk);

initial begin //OK
end
```

```
//always @(a) //ILLEGAL - 'always' block not allowed.
// begin
// end

task t1; //OK
endtask

function f1; //OK
endfunction

endprogram
```

As you see in the example, an “always” block is not allowed in a program. You will get a compile time error. “task” and “function” are ok and so is the initial block. Note that there are no events taking place in the program. Hold on to that thought.

Now, let us take the above example one step further. I am adding a “module” test0 to the design. So, the “module” (described next) and the “program” testP will simulate together. Here is module test0:

```
module test0 (input a, output logic b, output logic clk);

initial
begin
    b = 1; clk= 0;
    forever begin #10; clk = !clk; end
end

initial $monitor($stime,,,"`t monitor from test0 clk=%0d
b=%0d",clk,b);

initial #200 $finish(2); //BUT simulation terminates
at time 0
//because 'program' does not
have any
//activity beyond time 0.

always @(posedge clk)
begin
    b = !b;
    $display ($stime,,,"`t from test0 clk=%0d b=%0d",clk, b);
end

testP testPi (a, b, clk);

endmodule
```

‘module’ test0 instantiates “program” testP. Note that the module test0 has a free running clock and an ‘always’ block that runs on that clock. There is also a \$finish at time 200. So, you would think that the simulation will run until time 200 and then stop. NO. That is not what is going to happen because program testP does not have any activity beyond time 0. And as stated above, the program will terminate simulation as soon as all its threads are completed. Since there is no activity, the simulation threads complete at time 0. And so, will entire simulation. You will not get any displays because of the \$display in the “always” block. Here is the simulation log that shows only the \$monitor display at time 0:

```
0      monitor from test0 clk=0 b=1
$finish at simulation time      0
V C S  S i m u l a t i o n   R e p o r t
```

As noted above, you cannot access variables of a “program” from a “module.” For example:

```
module test0 (output logic c);

    testP testPi( );

    assign testPi.c = 1'b1; //ERROR
endmodule

program testP();
    wire c;
endprogram
```

In this example, we are trying to access “wire c” of program “testP” from module “test0.” This is not allowed and will result in following compile time error (Synopsys – VCS):

Error-[SV-IRTPE] Illegal reference to program element  
 testbench.sv, 8  
 "test0.testPi.c"

Illegal reference to program variable 'c' from outside program block  
 Program signals can only be accessed from within program scope.  
 Please make sure that the scope is same.

## 10.1 Eliminating Testbench Races

One of the main advantages of having testbench code in a program vs. a module is that the interaction between a design module and a testbench program will avoid race condition.

For example, let us say you are updating a variable, in a design module, as (@ posedge clk; a = 0;):

```
module m1( );
  @ (posedge clk) a=0; ...
endmodule
```

Now if you are looking at this variable “a” from *another module*:

```
module m2( );
  @ (posedge clk) if (a == 0) ...
endmodule
```

you will not know, if the second module will indeed see the value “0” that the first module output. So, there is a race condition between two modules. Different simulators may exhibit different behaviors. No error or warning will be issued.

Let us look at the following example. First, I show two modules interacting with each other. The simulation log of this will show a race condition. Then I will show a module interacting with a program, the simulation log of which will show that the race condition is gone.

Example: Module interacting with module. First, module test0:

```
module test0 (output logic b, output logic clk);
  initial
    begin
      b = 1; clk= 0;
      forever begin #10; clk = !clk; end
    end

  always @ (posedge clk)
    begin
      b = !b;
      $display ($stime,," from test0 clk=%0d b=%0d",clk,b);
    end

  testP testPi (b, clk);
endmodule
```

Next, module testP:

```

module testP (input b, input clk);

    initial begin
        repeat (3) begin
            @(posedge clk);
            $display ($stime,,, " from testP clk=%0d b=%0d",clk, b);
        end
    end

    initial #60 $finish;
endmodule

```

*Simulation log:*

```

10  from testP clk=1 b=1 //race condition at time 10. Value of 'b' differs
10  from test0 clk=1 b=0
30  from testP clk=1 b=0 //race condition at time 30. Value of 'b' differs
30  from test0 clk=1 b=1
50  from testP clk=1 b=1 //race condition at time 50. Value of 'b' differs
50  from test0 clk=1 b=0
$finish called from file "design.sv", line 11.
$finish at simulation time          60

```

V C S   S i m u l a t i o n   R e p o r t

In the module test0, we have a simple “always” loop that updates “b” at every posedge clk. We then display the clk and “b” values. “b” is the output of module test0.

In the module testP, we simply display the values of clk and “b.” “b” is the input to module testP.

If there was no race condition the value of “b” that the module test0 outputs will be the same as the value “b” that the module testP inputs. BUT it’s not, as shown in the simulation log.

Now, we will change the definition of “module testP” to “program testP” as shown below. Everything in this “program” is identical to what we saw in module testP – except that we have replaced “module” keyword with “program.” Module test0 remains the same:

```

program testP (input b, input clk);

    initial begin
        repeat (3) begin
            @(posedge clk);
            $display ($stime,,, " from testP clk=%0d b=%0d",clk,b);
        end
    end

```

```

    end

    initial #60 $finish;

endprogram

```

*Simulation log:*

```

10  from test0 clk=1 b=0  //No race condition at time 10
10  from testP clk=1 b=0
30  from test0 clk=1 b=1  //No race condition at time 30
30  from testP clk=1 b=1
50  from test0 clk=1 b=0  //No race condition at time 50
50  from testP clk=1 b=0

$finish called from file "design.sv", line 11.
$finish at simulation time          60

```

V C S   S i m u l a t i o n   R e p o r t

Everything being completely equal, we now see that the race condition that existed, when “b” is updated in module test0 and when it’s looked upon by program testP, is gone. This proves that the update on variables and their evaluation between modules will create a race condition, while the same between a module and a program will avoid this race condition.

The program construct addresses the race issue by scheduling its execution in the reactive region, after all design events have been processed, including clocks driven by non-blocking assignments. Reactive region is one of the last few phases before simulation time advances, and by then all design element statements would have been executed and the “program” will see the updated values. It is important to have this demarcation between the execution of design and testbench statements because it will give a more deterministic output between the two.

Here is another example, showing that a non-blocking assignment in “module DUT” does not get updated correct in the testbench when the testbench is a “module.” But that it does get updated correct when the testbench is a “program.”

First, both the DUT and testbench are declared in modules:

```

//-----
//DUT
//-----
module DUT(output int addr, output int data);
    initial begin
        addr <= 10;
        data <= 20;
    end
endmodule

```

```

//-----
//testbench - a 'module'
//-----
module testbench(input int addr, input int data);
    initial begin
        $display("\t Addr = %0d Data = %0d",addr,data);
    end
endmodule

//-----
//testbench top
//-----
module tbench_top;
    wire [31:0] addr;
    wire [31:0] data;

    //design instance
    DUT dut(addr, data);

    //testbench instance
    testbench test(addr, data);
endmodule

```

*Simulation log:*

```
# run -all
#     Addr = 0 Data = 0
```

In this example, we assign “addr” and “data” using non-blocking assignments in the DUT module, and we display these values in the testbench (which is also a module). As you notice in the simulation log, the assigned values of “addr” and “data” do not get reflected in the testbench “module.” Both the DUT and the testbench modules simulate in the active region of the simulation time tick, and there is no guarantee which values will be seen by the testbench. The assigned values or the ones before the assignment. There is a race condition.

Now let us look at the same example, but the testbench is now a “program.” Everything is identical except that the testbench is now declared as a “program”:

```

//-----
//DUT
//-----
module DUT(output int addr, output int data);
    initial begin
        addr <= 10;
        data <= 20;
    end

```

```

    endmodule

//-----
//testbench - a 'program'
//-----
program testbench(input int addr, input int data);
    initial begin
        $display("\t Addr = %0d Data = %0d",addr,data);
        $exit;
    end
endprogram

//-----
//testbench top
//-----
module tbench_top;
    wire [31:0] addr;
    wire [31:0] data;

    //design instance
    DUT dut(addr,data);

    //testbench instance
    testbench test(addr,data);
endmodule

```

*Simulation log:*

```
# run -all
#     Addr = 10 Data = 20
```

As you notice from simulation log, that when the testbench is declared as a “program,” the “addr” and “data” value assigned in the DUT correctly get reflected in the testbench. That is because the DUT simulated in the active region, while the testbench simulated in the reactive region which gets executed after the activity in the active region has been completed.

Finally, note that in addition to the normal simulation finish tasks such as \$stop and \$finish, a program can use the \$exit control task as well. The “program” block terminates the threads of all its initial procedures as well as all of their descendant threads explicitly by calling the \$exit system task.

# Chapter 11

## SystemVerilog “interface”



**Introduction** This chapter discusses nuances of SystemVerilog “interface,” including modports (import/export), tasks/functions in an interface, parameterized interfaces, etc.

### 11.1 Interfaces

Normally, we connect two module instances directly via either position based or name-based connection, each net connected to another in the instances. This maybe well and good for a few connections, but in latest designs, we are talking about hundreds of nets going in between module instances. Worse when you add a new signal between two blocks. You not only have to edit both the connecting blocks to add the new port but also the higher-level modules that connect up the modules.

The interface construct is specifically created to encapsulate communication between blocks in a modular fashion. Interface allows for connection between module instances to be less error prone and reduces typing hundreds of nets for each instance of the module. By encapsulating the communication between blocks, the interface construct also facilitates design reuse. Interface is an important capability of SystemVerilog.

Essentially, the interface is a named bundle of nets or variables. An interface allows a number of signals to be grouped together and represented as a single port. The declarations of the signals that make up the interface are contained in a single location/declaration. Each module that uses these signals then has a single port of the interface type, instead of many ports with discrete signals. The ability to replace

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_11](https://doi.org/10.1007/978-3-030-71319-5_11)) contains supplementary material, which is available to authorized users.

a group of names by a single name can significantly reduce the size of a description and improve its maintainability.

The interface is instantiated in a design and can be accessed through a port or as a single item. A significant proportion of a design often consists of port lists and port connection lists, which are just repetitions of names. The ability to replace a group of names by a single name can significantly reduce the size of a description, improve its maintainability, and make the process less error prone.

Additional power of the interface comes from its ability to encapsulate functionality as well as connectivity, making an interface, at its highest level, more like a class template. Modules that are connected via an interface can simply call the subroutine members of that interface to drive the communication. With the functionality thus encapsulated in the interface and isolated from the module, the abstraction level and/or granularity of the communication protocol can be easily changed by replacing the interface with a different interface containing the same members but implemented at a different level of abstraction. The modules connected via the interface do not need to change at all.

In addition to subroutine methods, an interface can also contain processes (i.e., initial or always procedures) and continuous assignments, which are useful for system-level modeling and testbench applications. The interface can include its own protocol checker that automatically verifies that all modules connected via the interface conform to the specified protocol. Other applications, such as functional coverage recording and reporting, protocol checking, and assertions, can also be built into the interface.

Interfaces can be declared and instantiated in modules, but modules can neither be declared nor instantiated in interfaces.

The interface block is defined by the keyword pair, *interface* and *endinterface*. This keyword pair is used to define a separate structural block, similar to a SystemVerilog “module.”

A simple interface declaration is as follows:

```
interface identifier;
...
interface_items
...
endinterface [: identifier]
```

Some high-level points of an interface:

- Interface encapsulates connectivity. It encapsulates the communication between blocks.
- In its simplest form, it is a named bundle of nets or variables.
- An interface can be passed as a single item through a port, thus replacing a group of names by a single name.
- It encapsulates parameters, constants, variables, functions and tasks (a.k.a. *interface methods*), “initial” and “always” blocks, and continuous assignments, useful for encapsulating all interface related functionality in the interface itself. Thus, the functionality can be isolated from the modules that are connected via the interface.

- Interface can also include functional coverage recording and reporting, protocol checking, and SV assertions.
- An interface may be parameterized in the same way as a module. This helps for modular and reusable environment.
- Interfaces can also include instances of other interfaces, allowing more complex, hierarchical interfaces to be modeled. In other words, a port of an interface can also be defined as an interface. This capability allows one interface to be connected to another interface.

Let us look at an example – without – the use of an interface:

```
module PCIM( input logic req,
              input logic clk,
              input logic [1:0] CBE,
              input logic [7:0] addr,
              inout wire [7:0] data,
              output bit irdy );
  //...
endmodule

module PCIS (
  input logic clk,
  output logic gnt,
  input bit irdy,
  inout wire [7:0] data,
  output logic req,
  output logic start,
  input logic [7:0] addr );
  //...
endmodule

module top;
  wire req, gnt, start, irdy;
  logic clk = 0;
  wire [1:0] mode;
  wire [7:0] addr;
  wire [7:0] data;
  wire [1:0] CBE;
  PCIM mem(req, clk, CBE, addr, data, irdy);
  PCIS cpu(clk, gnt, irdy, data, req, start, addr);
endmodule
```

As you can see, the PCIM module and PCIS module have many signals that need to be connected in the module “top.” With position or name-based connections, the chances of making errors are very high when you connect the PCIM and PCIS

modules in the module “top.” Also, all the signals are repeated, once with the module declaration and then when these modules are instantiated, and they all need to be redeclared in the module “top.” Very tedious, repetitive, and error prone.

Let us look at one more example, also – without – an interface. Then we will see the same example – with – an interface and see how easier, modular, and less error prone the code becomes:

```
module cache (input logic [7:0] Addr, input logic [7:0] data, input
RWN, input Enb);
    //...
endmodule

module CPU (output logic [7:0] data, output logic [7:0] Addr,
output Enb, output RWN);
    //...
endmodule

// Traditional way of connecting
module TOP;
    wire [7:0] Addr;
    wire [7:0] data;
    wire RWN;
    wire Enb;

    cache cache1 (Addr, data, RWN, Enb);
    CPU cpu1 (data, Addr, Enb, RWN);
endmodule
```

This example shows the traditional way of connecting two modules (module cache and the module CPU). We had to declare the I/O ports twice: once for module cache and then again for module CPU. Then when we connect the two module instances in module TOP, we again must use the port names for position-based connection between instance cache1 and the instance cpu1. We also need to declare the ports as wires in the module TOP. A lot of repetition with no reusability and many chances of making errors.

Let us look at the same example with the use of an interface:

```
interface busIntf; //Interface declaration
    logic [7:0] Addr;
    logic [7:0] data;
    logic RWN;
    logic Enb;
endinterface

module cache (busIntf cacheIntf); //Interface as I/O
```

```

//....
endmodule

module CPU (busIntf CPUIntf); //Interface as I/O
//....
endmodule

// Using the interface
module TOP;

busIntf TOPIntf (); //Instantiate interface

cache cache1 (.cacheIntf(TOPIntf));
CPU cpu1 (.CPUIntf(TOPIntf));

endmodule

```

We first define an interface named “busIntf” which encapsulates all the interface signals, I/O of the underlying modules. In module “cache” we simply use the interface as I/O (i.e., instantiate in the module I/O port list). We do the same for the module “CPU.” Then in the module TOP, we instantiate the “busIntf” and call it TOPIntf. We then simply use this instance of the interface to connect the module instance cache1 with the module instance cpu1.

As you see, the I/O signals of all the modules are declared only once in the interface. It is then used in the underlying modules and also used to connect modules. Very less error prone and saves a lot of time in typing and retyping. Very modular and reusable.

Let us now see how you can access the ports declared in an interface. We need to be able to access interface signals in the modules that use the interface. Here is an example:

```

interface Bus;
  logic [7:0] Addr, Data;
  logic RWn;
  logic Enb;
endinterface

module cache (Bus MemBus);
  logic [7:0] mem[0:255];

  always @ (MemBus.RWn) //Use MemBus interface to access
its ports
begin
  #1;

```

```

mem[MemBus.Addr] = TestCache.mem[MemBus.Addr];
$display ("MemBus Addr = ",MemBus.Addr);
$display ("\t MemBus Data = ",mem[MemBus.Addr]);
end
endmodule

module TestCache;
logic[7:0] mem[0:3];

Bus RAMBus(); // Instance the interface
cache cachel (.MemBus(RAMBus)); // Connect it

initial
begin
RAMBus.RWn = 0; //Use RAMBus interface to access its ports
RAMBus.Addr = 0;
RAMBus.Enb = 1;
for (int I=0; I<2; I++) begin
RAMBus.RWn = ~RAMBus.RWn;
RAMBus.Addr = RAMBus.Addr + 1;
mem[RAMBus.Addr] = $urandom; //Write Data to Mem
$display ("RAMBus Addr = ",RAMBus.Addr);
$display ("\t RAMBus Data = ",mem[RAMBus.Addr]);
#10;
end
end
endmodule

```

*Simulation log:*

```

RAMBus Addr = 1
    RAMBus Data = 54
MemBus Addr = 1
    MemBus Data = 54
RAMBus Addr = 2
    RAMBus Data = 60
MemBus Addr = 2
    MemBus Data = 60

```

In this example, we define an interface “Bus” and instantiate it in the module “cache” as its I/O, as “Bus MemBus.” To access the signals of the interface “Bus,” you simply use the instance name “MemBus” of the interface as shown in the code. Similarly, we use the interface instance name “RAMBus” in the module TestCache to access signals of the interface “Bus.”

So far, we have seen interfaces with all the required signals encapsulated within it. But what if you want an external signal to be connected to the interface. In other

words, interface itself needs an I/O. This is useful to customize interface connections with explicit connection to interface I/O. Here is an example:

```

interface iFace(input logic clk); //external input 'clk'
    logic [7:0] data;
    logic [7:0] addr;
endinterface

module PCIM(iFace iFacePCIM);
//...
endmodule

module PCIS (iFace iFacePCIS);
//...
endmodule

module top;
    logic topClk;

    //External topClk connected to clk
    iFace iFaceInst(.clk(topClk)); //Instantiate interface

    //wire [7:0] topdata;
    //iFace iFaceInst(.data(topdata));
        //ERROR - port 'data' not defined

    //Connect to PCIM - name based
    PCIM mem(.iFacePCIM(iFaceInst));

    //Connect to PCIS - position based
    PCIS cpu(iFaceInst);
endmodule

```

In this example, we define an interface “iFace” with an external input “clk.” This allows us to use the interface and connect different clocks to the interface. In the module “top,” we declare “topClk” and connect it to the interface clock “clk.” This is done when we instantiate the interface as shown in the code. This is a very important feature and allows for a lot of flexibility in customizing an interface.

But note that you cannot access signals internal to the interface and connect those to some external signals. So, the following will result an error:

```

//wire [7:0] topdata;
//iFace iFaceInst(.data(topdata));
    //ERROR - port 'data' not defined

```

Following is the error you will get (Synopsys – VCS):

Error-[UPIMI-E] Undefined port in module instantiation  
testbench.sv, 24

Port "data" is not defined in interface 'iFace' defined in "testbench.sv", 4  
Interface instance: iFace iFaceInst(.data (topdata));

## 11.2 Modports

Interfaces provide a practical and straightforward way to simplify connections between modules. However, each module connected to an interface may need to see a slightly different view of the connections within the interface. For example, to a slave on a bus, a “request” signal might be an output from the slave, whereas to a processor on the same bus, “request” would be an input.

Modport (abbreviation for “module port”) allows you to specify the direction (input, output, etc.) of the signals declared within an interface. Also, in a complex interface between several different modules, it may be that not every module needs to see the same set of signals within the interface. Modports make it possible to create a customized view of the interface for each module connected.

Note that if a module is connected to the interface without specifying a modport, the module will have access to all signals defined in the interface.

*The directions of signals are from inside the module point of view.* You can declare as many modports as you like. Here is an example:

```
interface mIntf;
    wire a, b, c, d, e, f;
    modport master (input a, b, output c, d);
    modport slave (output a, b, input c, d);
    modport bus (output e, f);
endinterface
```

Interface “mIntf” defines two modports. Modport “master” has two inputs and two outputs. This means that a and b are inputs to module that uses modport “master” and c and d are outputs from the module that uses modport “master.” Similar analogy applies to modport “slave” and “bus.”

Now, let us see how modules connect to these modports. There are three ways. Continuing with above example, here is the first one:

```
interface mIntf;
    wire a, b, c, d;
    modport master (input a, b, output c, d);
    modport slave (output a, b, input c, d);
endinterface
```

```

//modport and its instantiation in module header
module m1 (mIntf.master iM);
//...
endmodule

//modport and its instantiation in module header
module s1 (mIntf.slave iS);
//...
endmodule

module top;

mIntf mI(); //Instantiate interface

m1 u1(.iM(mI)); //Connect modport
s1 u2(.iS(mI));
endmodule

```

In this example, the modport list name (“master” or “slave”) is specified in the module header.

The *interface\_name.modport\_name* selects the interface, and the modport name selects the appropriate directional information for the interface signals accessed in the module header.

Module “m1” declares in its header, modport “mIntf.master,” and gives it an instance name, “iM.” This way you give directionality to the module “m1.” In other words, module “m1”’s port directionality is that of the modport “master.” Modport “master”’s I/O definition now applies to the module “m1”’s port direction. Similar definition is applied to module “s1.”

Next, in module “top,” we instantiate modules “m1” and “s1” and connect the “master” modport interface of “mIntf” with the “slave” modport interface of “mIntf.”

Here is the second way to do the same:

```

interface mIntf;
    wire a, b, c, d;
    modport master (input a, b, output c, d);
    modport slave (output a, b, input c, d);
endinterface

//Use mIntf as I/O (not the 'master' modport)
module m1 (mIntf iM);
//...
endmodule

```

```
//Use mIntf as I/O (not the 'slave' modport)
module s1 (mIntf iS);
//...
endmodule

module top;

mIntf mI(); //Instantiate interface

//Instantiate module and connect 'master' modport
m1 u1(.iM(mI.master));

//Instantiate module and connect 'slave' modport
s1 u2(.iS(mI.slave));
endmodule
```

In this example, we connect “master” and “slave” modports slightly different. In the module header of “m1” and “s1,” we use the entire interface (mIntf) as the module I/O. But when we instantiate “m1” and “s1” in module top, we connect the interface “mIntf” to the modports “master” and “slave.” This is another way to connect modules with modport directionality.

Here is the third way where we connect the port bundle though a *generic interface*:

```
interface mIntf;
    wire a, b, c, d;
    modport master (input a, b, output c, d);
    modport slave (output a, b, input c, d);
endinterface

module m1 (interface iM); //Use generic 'interface'
//...
endmodule

module s1 (interface iS); //Use generic 'interface'
//...
endmodule

module top;

mIntf mI(); //Instantiate interface
```

```
//Instantiate module and connect master modport  
m1 u1(.iM(mI.master));  
  
//Instantiate module and connect slave modport  
s1 u2(.iS(mI.slave));  
endmodule
```

In this example, we use the keyword *interface* in the module header. The actual interface is instantiated in module “top” and connected as shown (same as in example two). This way you do not need to know the name of the interface when declaring module headers.

Here is the same example, but it shows how to access signals internal to an interface/modport from an external module:

```
interface mIntf;  
    logic a, b, c, d;  
    modport master (input a, b, output c, d);  
    modport slave (output a, b, input c, d);  
endinterface  
  
module m1 (interface iM); //Use generic interface  
//...  
endmodule  
  
module s1 (interface iS); //Use generic interface  
  
//access modport signals internal to 'mIntf' interface  
always @(iS.c)  
    iS.b = iS.c && iS.d;  
endmodule  
  
module top;  
  
    mIntf mI(); //Instantiate interface  
  
    //Instantiate module and connect master modport  
    m1 u1(.iM(mI.master));  
  
    //Instantiate module and connect slave modport  
    s1 u2(.iS(mI.slave));  
endmodule
```

Module “s1” has an always block that uses modport signals internal to the interface “mIntf.”

*Note:* Even when an interface is defined with modports, modules can still be connected to the complete interface, without specifying a specific modport. However, the port directions of signals within an interface are only defined as part of a modport view. When no modport is specified as part of the connection to the interface, all nets in the interface are assumed to have a bidirectional “inout” direction, and all variables in the interface are assumed to be of type “ref.”

### 11.3 Tasks and Functions in an Interface

Tasks and functions (a.k.a. *interface methods*) can be defined in an interface. The code for communication between modules is only written once, as interface methods, and shared by each module connected using the interface.

Interface methods can operate on any of the signals within the interface. Values can be passed into the interface methods from outside the interface as an input argument. Values can be written out from the interface methods as output arguments or function returns.

Interface methods offer several advantages for modeling large designs. The code for communicating between modules does not need to be distributed across multiple modules. Instead, the communication code is encapsulated in the interface and shared by each module connected to the interface. The modules connected to the interface simply call the interface methods, instead of having to implement the communication protocol functionality within the modules. For example, describing a bus protocol within each module that uses the bus leads to duplicate code. If any change needs to be made to the bus protocol, the code for the protocol must be changes in each module that shares the bus.

You can define these tasks/functions directly in an interface or within a modport in an interface. If the interface is connected via a modport, the method must be specified using the ‘*import*’ keyword. We will see that a bit later.

Here is an example of using tasks in an interface and referring to them from external modules that use this interface:

```
interface mIntf (input logic clk);
    logic req, gnt, start, rdy;
    logic [7:0] addr, data;

    task READ(input logic [15:0] ReadAddr);
        //...
    endtask

    task WRITE;
        //...
    endtask

```

```
    endtask

endinterface

module Memory (interface memIntf);
    logic [15:0] raddr;

    always @ (posedge memIntf.clk)
        memIntf.gnt <= memIntf.req && memIntf.rdy;

    always @ (memIntf.rdy)
        memIntf.READ(raddr); //Call READ task of 'mIntf'

endmodule

module CPU (interface CPUIntf);

    always @ (CPUIntf.clk)
        if (CPUIntf.gnt == 1'b1)
            CPUIntf.WRITE; //Call WRITE task of interface 'mIntf'

endmodule

module TOP;
    logic clk;

    initial begin
        clk = 0;
        forever #5; clk = !clk;
    end

    initial #100 $finish;
    mIntf mI(clk); //instantiate interface

    Memory mem(mI);
    CPU cpu(mI);
endmodule
```

In this example, we define two tasks, namely, READ and WRITE, in the interface “mIntf.” These tasks are now available to wherever this interface is used. Idea behind embedding tasks in an interface is to have tasks that work on the interface

signals be defined where these signals are defined, i.e., in the interface. This allows for modular code development. We call the READ task from module memory and WRITE task from module CPU.

### 11.3.1 “import” Tasks in a Modport

When tasks are defined in a modport, they are declared as *import* tasks. Note that in the above example, we declared tasks directly in the interface – not – in the modports. Following example shows how to declare tasks in modports:

```

interface mIntf (input logic clk);
    logic req, gnt, start, rdy;
    logic [7:0] addr, data;

    modport PCIS (input req, start, rdy, addr,
                  output data,
                  import READ);
    //import into the module that uses modports

    modport PCIM (input req, start, rdy, data,
                  output addr,
                  import WRITE);
    //import into the module that uses modports

    task READ(input logic [7:0] ReadAddr);
        begin
            @(posedge clk) if (req == 1) addr = ReadAddr;
        end
    endtask

    task WRITE;
        begin
            @(negedge clk) if (gnt == 1) data = 'h ff;
        end
    endtask

endinterface

module Memory (interface memIntf);
    logic [15:0] raddr;

    always @(posedge memIntf.clk)
        memIntf.gnt <= memIntf.req && memIntf.rdy;

```

```
    always @ (memIntf.rdy)
        memIntf.READ(raddr); //task imported from modport PCIS

    endmodule

    module CPU (interface CPUIntf);

        always @ (CPUIntf.clk)
            if (CPUIntf.gnt == 1'b1)
                CPUIntf.WRITE; //task imported from modport PCIM

    endmodule

    module TOP;
        logic clk;

        initial begin
            clk = 0;
            forever #5; clk = !clk;
        end

        initial #100 $finish;

        mIntf mI(clk); //instantiate interface

        //has access to modport PCIS tasks
        Memory mem(.memIntf(mI.PCIS));

        //has access to modport PCIM tasks
        CPU cpu(.CPUIntf(mI.PCIM));

    endmodule
```

In this example, we declare, as import tasks, task READ in modport PCIS and task WRITE in modport PCIM. These tasks are now imported into any module that uses this interface/modport. Module “memory” is connected (in module TOP) to the modport PCIS. So, the tasks imported from modport PCIS are available to module “memory.” Similarly, tasks imported from modport PCIM are available in module “CPU.”

### ***11.3.2 “export” Tasks in a Modport***

“export” defines tasks in one module and calls them in another, using modports to control task access. This is a mechanism whereby you define a task or function in one module and then “export” it through an interface to other modules. An “export” declaration in an interface modport does not require a full prototype of the task or function arguments. Only the task or function name needs to be listed in the modport declaration. The “export” declaration allows a module to “export” a task or function to an interface through a specific modport of the interface.

Here is an example:

```

interface mIntf (input logic clk);
    logic req, gnt, start, rdy;
    logic [7:0] addr, data;

        modport PCIS (input req, start, rdy, addr, clk,
                       output data, gnt,
                       export READ);
        //export from modport PCIS

        modport PCIM (input req, start, rdy, data, clk,
                      output addr, gnt,
                      import task READ(inout logic [7:0] addr) );
        //import in PCIM, the exported task from PCIS

    endinterface

module Memory (mIntf.PCIS memIntf );
    //task defined in module 'Memory' and exported from
    //modport PCIS via 'mIntf'
    task memIntf.READ (inout rAddr);
        begin
            @(posedge memIntf.clk)
            if (memIntf.req == 1) rAddr = rAddr + 'hff;
        end
    endtask

    always @(posedge memIntf.clk)
        memIntf.gnt <= memIntf.req && memIntf.rdy;

endmodule

```

```
module CPU (mIntf.PCIM CPUIntf );
    logic [7:0] addr;

    always @ (CPUIntf.clk)
        if (CPUIntf.gnt == 1'b1)
            CPUIntf.READ(addr) ; //task imported from modport PCIM

    endmodule

module TOP;
    logic clk;

    initial begin
        clk = 0;
        forever #5; clk = !clk;
    end

    initial #100 $finish;

    mIntf mI(clk);

    Memory mem(.memIntf(mI)); //export the READ task
    CPU cpu(.CPUIntf(mI));    //import the READ task

endmodule
```

In this example, we export the READ task from modport PCIS. Since it is exported, it needs to be defined in a module that connects to the modport PCIS. In our case, module memory is connected to modport PCIS. Hence we define the task READ in that module. Then we import the same task READ into the modport PCIM. This task is now available into the module that is connected to modport PCIM. This module is CPU. Hence CPU can reference task READ from within it.

## 11.4 Parameterized Interface

Just like a SystemVerilog module, you can parameterize an interface as well, allows for configurable and reusable interface definitions. The following example shows how to use parameters in an interface:

```

interface mIntf #(addrWidth = 8, dataWidth = 8) (input
logic clk);
    logic req, gnt, start, rdy;
    logic [addrWidth-1:0] addr;
    logic [dataWidth-1:0] data;

modport PCIS (input req, start, rdy, addr, clk, gnt,
              output data,
import task READ (input logic [addrWidth-1:0] ReadAddr) );
//import into the module that uses modport PCIS

modport PCIM (input req, start, rdy, data, clk, gnt,
              output addr,
import task WRITE( ) );
//import into the module that uses modport PCIM

task READ(input logic [addrWidth-1:0] ReadAddr);
    @(posedge clk) if (req == 1) ReadAddr = addr;
endtask

task WRITE;
    @(negedge clk) if (gnt == 1) data = 'h ff;
endtask

endinterface

module Memory (interface memIntf);
    logic [15:0] raddr;

    always @(posedge memIntf.clk)
        memIntf.gnt <= memIntf.req && memIntf.rdy;

    always @(memIntf.rdy)
        memIntf.READ(raddr); //task imported from modport PCIS

endmodule

module CPU (interface CPUIntf);

    always @(CPUIntf.clk)
        if (CPUIntf.gnt == 1'b1)
            CPUIntf.WRITE; //task imported from modport PCIM

```

```

endmodule

module TOP;
  logic clk = 0;

  mIntf mI(clk); //parameters take on default values #(8,8)
  mIntf #(16,16) mIw(clk); //New parameter values #(16,16)

  Memory mem(.memIntf(mI.PCIS)); //8 bit wide Memory
  CPU cpu(.CPUIntf(mI.PCIM)); //8 bit wide CPU

  Memory memW(.memIntf(mIw.PCIS)); //16 bit wide Memory
  CPU cpuW(.CPUIntf(mIw.PCIM)); //16 bit wide CPU
endmodule

```

The example is similar to what we have seen previously, except that this time we use two parameters in the interface header: `addrWidth` and `dataWidth`. We then use these parameters to size the addr and data buses. These parameters can be overridden when we instantiate the interface. This way we can customize the interface directly when we instantiate it and do not have to change the definition of interface itself.

In module TOP, we instantiate the interface twice:

```

mIntf mI(clk); //parameters take on default values #(8,8)
mIntf #(16,16) mIw(clk); //New parameter values #(16,16)

```

In the instance “`mI`,” we leave the parameters to their default values (#8,8). But in the instance “`mIw`,” we redefine the `addrWidth` and `dataWidth` parameters to be of value #(16,16). This way the modules that are tied to this instance will now have wider `addrWidth` and `dataWidth`. Examine the code carefully to see what is going on.

You can also use a “type” parameter and override it during interface instantiation. For example, you have an interface header as follows:

```

interface mIntf #(addrWidth = 8, dataWidth = 8, type DTYPE = int)
  (input logic clk);

```

You can change the “`type DTYPE = int`”-type parameter when you instantiate the interface, as follows:

```

mIntf #(.DTYPE(real)) mI(clk);

```

## 11.5 Clocking Block in an Interface

Please refer to Chap. 19 for complete description of a clocking block. This section is simply to state that you can use a clocking block in an interface to synchronize the timing of I/O from the interface.

If you want to specify the timing of synchronous signals relative to the clocks, you need to use a clocking block within an interface. Signals in a clocking block are driven or sampled synchronously, ensuring that your testbench interacts with the signals at the correct synchronous points in time. The main benefit of clocking blocks is that you can put all the detailed timing information in here and not clutter your testbench with it.

A detailed example of clocking block in an interface is provided in Sect. 19.1.

Here’s another example, simply to show how the clocking block clocking event can be used in your testbench:

```
interface busIntf (input bit clk);
    logic request, grant, rst;

    clocking cb @ (posedge clk);
        output request;
        input grant;
    endclocking

    //modport master (output request, rst,
    //                  input grant);

    modport master (clocking cb, output rst);

    modport target (input request, rst, clk,
                    output grant);
endinterface

module busMaster (busIntf.master busIf);
endmodule

module testbench;
    logic clk;

    initial begin
        clk = 0;
        forever #10 clk = !clk;
    end
```

```

initial
$monitor($stime,,, "request = %b",mI.cb.request);

busIntf mI(clk); //Instantiate interface
busMaster u1(.busIf(mI)); //Connect Interface

initial begin
@mI.cb; //@clocking block clock edge
mI.cb.request <= 1;
@mI.cb;
mI.cb.request <= 0;
#20 $finish;
end
endmodule

```

*Simulation log:*

```

# run -all
# 0 request = x
# 10 request = 1
# 30 request = 0
# ** Note: $finish : testbench.sv(45)

```

In this example, we define a clocking block called “cb” in the interface “busIntf.” The clocking block drives and samples signals @(posedge clk). Once you have defined a clocking block, your testbench can wait for the clocking expression “@mI.cb” (as in our testbench) rather than having to specify the exact clock and the edge.

We use the clocking block “cb” in the modport definition of “master” modport. And the clocking block “cb” has two signals: “output request” and “input grant.” So, when the “master” modport uses the “clocking cb” as its port, the “master” modport now treats “request” and “grant” as synchronous signals. The clocking block “cb” declares that the signals are active on the posedge of clock. The signal directions are relative to the modport where they are used. In other words, “request” is a synchronous output in the modport “master,” and “grant” is a synchronous input. But the signals “rst” of the “master” modport remain asynchronous.

In the module “testbench,” we do not use the explicit “@posedge clk” to drive signals. Instead, we simply use “@mI.cb” as a clocking edge. Note that in order to access the signals of the clocking block, you have to use the hierarchical name: the interface instance name and the clocking block name as in “mI.cb.request.”

Note that an interface can contain multiple clocking blocks, one per clock domain, as there is a single clock expression in each block.

# Chapter 12

## Operators



**Introduction** This chapter describes all available operators of the language, including assignment, increment/decrement, arithmetic, relational, equality, logical, bitwise, shift, conditional, concatenation, replication, streaming, wildcard equality, unary reduction, etc. operators.

SystemVerilog offers a rich set of operators. We will study operators and also the operands used with these operators to build expressions. Here is a list of operators (SystemVerilog-LRM). In Table 12.1, unary operator means the operator works on one operand, while binary operator means it works on two operands. Also, an integral data type represents a single basic integer data type, packed array, packed structure, packed union, enum variable, or time variable.

### 12.1 Assignment Operators

We are familiar with the simple assignment operator `=`, but as you notice in Table 12.1, there are many other assignment operators available, namely, `+ =`, `- =`, `* =`, `/ =`, `% =`, `& =`, `| =`, `^ =`, `<<=`, `>>=`, `<<<=`, and `>>>=`.

Here are some examples:

```
A += 2; //same as A = A+2;  
A -= 2; //same as A = A-2;  
A *= 4; //same as A = A*4;  
A >>>= 3; //same as A = A >>> 3;  
A <<= 2; //same as A = A << 2;
```

The same thought applies to the rest of the assignment operators.

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_12](https://doi.org/10.1007/978-3-030-71319-5_12)) contains supplementary material, which is available to authorized users.

## 12.2 Increment and Decrement Operators

These operators are `++i`, `--i`, `i++`, and `i--`. Let us see how these works. There is a subtle difference between `++i` and `i++` and between `--i` and `i--`. Here is an example that highlights the difference:

```
module OP;
    int a, b, c, d;

    initial
        begin
            a = 10;
            b = a ++; //Assign a to b, then increment a
            $display("b = a ++ :: a = %0d b = %0d",a,b);

            a = 20;
            b = ++ a; //increment a, then assign to b
            $display("b = ++ a :: a = %0d b = %0d",a,b);

            a = 10;
            b = a --; //Assign a to b, then decrement a
            $display("b = a -- :: a = %0d b = %0d",a,b);

            a = 20;
            b = -- a; //decrement a, then assign to b
            $display("b = -- a :: a = %0d b = %0d",a,b);

            a = 10; c = 10;
            b = ++a + ++c;
            //Increment a and c; then assign a+c to b
            $display("b = ++a + ++c :: a = %0d c = %0d b = %0d",a,c,b);

            a = 10; c = 10;
            b = a++ + c++;
            //Assign a+c to b first; then increment a and c
            $display("b = a++ + c++ :: a = %0d c = %0d b = %0d",a,c,b);

        end
endmodule
```

*Simulation log:*

```
b = a ++ :: a = 11 b = 10
b = ++ a :: a = 21 b = 21
b = a -- :: a = 9 b = 10
b = -- a :: a = 19 b = 19
```

```
b = ++a + ++c :: a = 11 c = 11 b = 22  
b = a++ + c++ :: a = 11 c = 11 b = 20
```

## VCS Simulation Report

The interesting part to note is the difference between `i++` and `++i` and `i--` and `i-` when it comes to using them in expression. So, in the following:

```
a = 10;  
b = a ++; //Assign a to b, then increment a
```

`a++` means that you assign “`a`” to “`b`” and then increment “`a`. So, first we assign `a=10` to “`b`” (so, `b=10`). And then increment “`a`” (so, `a = 11`). This is an important point to note. In contrast, in the following, we first increment “`a`” (so, `a = 21`) and then assign it to “`b`” (so, `b = 21`). Hence the simulation log shows different results for `a++` and `++a`:

**Table 12.1** SystemVerilog operators

Operator	Description	Operand data types
=	Binary assignment operator	Any
+ - -= /= *=	Binary arithmetic assignment operators	Integral, real, shortreal
%=	Binary arithmetic modulus assignment operator	Integral
&=  = ^=	Binary bitwise assignment operators	Integral
>>= <<=	Binary logical shift assignment operators	Integral
>>>= <<<=	Binary arithmetic shift assignment operators	Integral
?:	Conditional operator	Any
+ -	Unary arithmetic operators	Integral, real, shortreal
!	Unary logical negation operator	Integral, real, shortreal
~ & ~&   ~  ^ ~^	Unary logical reduction operators	Integral
+ - * / **	Binary arithmetic operators	Integral, real, shortreal
%	Binary arithmetic modulus operator	Integral
&   ^ ~^ ~^	Binary bitwise operators	Integral
>> <<	Binary logical shift operators	Integral
>>> <<<	Binary arithmetic shift operators	Integral
&&	Binary logical operators	Integral, real, shortreal
-> <->		
< <= > >=	Binary relational operators	Integral, real, shortreal
==!=	Binary case equality operators	Any except real and shortreal
==!=	Binary logical equality operators	Any
==? !=?	Binary wildcard equality operators	Integral
++--	Unary increment, decrement operators	Integral, real, shortreal
inside	Binary set membership operator	Singular for the left operand
dist	Binary distribution operator	Integral
{ } {{}}	Concatenation, replication operators	Integral
{<<{} } {>>{}}	Stream operators	Integral

```
a = 20;
b = ++ a; //increment a, then assign to b
```

The similar argument applies to `a--` and `--a` as described with comments in the code.

The same argument also applies to the following where “`a`” and “`c`” are incremented first and then their sum `a + c` is assigned to “`b`”:

```
a = 10; c = 10;
b = ++a + ++c;
//Increment a and c; then assign a + c to b
```

## 12.3 Arithmetic Operators

Table 12.2 shows the arithmetic operators available in the language.

Operators, plus, minus, and multiply, are obvious and we will not go into their detail. But division and modulo operators are interesting. The following example shows different types of numbers (integer, real) as well as positive and negative indices and their effect on these operators:

```
module OP;
  integer a, b, c, d;
  real e, f, g, h;

  initial begin

    /*** DIVISION ***/

    $display("\n DIVISION \n");
  end
endmodule
```

**Table 12.2** Arithmetic operators

Operator	Description
<code>a + b</code>	a plus b
<code>a - b</code>	a minus b
<code>a * b</code>	a times b
<code>a / b</code>	a divided by b
<code>a % b</code>	a modulo b
<code>a ** b</code>	a to the power of b

```
a = 10; b= 3;
c = a/b; //c=3
$display("c = 10/3 :: a = %0d b = %0d c = %0d", a , b, c);
           //Integer division truncates any fractional part to 0

a = 3.4; b = 2.3; //Truncate a and b since they are integers
c = a/b; //c=1
$display("integer c = 3.4/2.3 :: a = %0f b = %0f c = %0f",
         a , b, c);

e = 3.4; f = 2.3; //No truncation since e and f are real
g = e/f; //g=1.478
$display("real g = 3.4/2.3 :: e = %0f f = %0f g = %0f",
         e, f, g);

/** A TO THE POWER of B ***/

$display("\nA to the Power of B \n");

a = 3.4; b = 2.3; //Truncate a and b since they are integers
c = a**b; //c=9
$display("integer c = 3.4**2.3 :: a = %0f b = %0f c = %0f",
         a , b, c);

e = 3.4; f = 2.3; //No truncation since e and f are real
g = e**f; //g=16.687
$display("real g = 3.4**2.3 :: e = %0f f = %0f g = %0f",
         e, f, g);

e = 9; f = 0.5; //square root
g = e**f; //g=3.0
$display("real g = 9**.5 :: e = %0f f = %0f g = %0f",
         e, f, g);

a = 0; b= 3; //0**3 = 0
c = a**b;
$display("c = 0**3 :: a = %0d b = %0d c = %0d", a , b, c);

a = 0; b= -3; //0**-3 = x
c = a**b;
$display("c = 0**-3 :: a = %0d b = %0d c = %0d", a , b, c);

// -2**3 = -8 : takes on the sign of first operand
```

```

a = -2; b= 3;
c = a**b;
$display("c = -2**3 :: a = %0d b = %0d c = %0d", a , b, c);

/** A modulo B (A % B) **/


$display("\n A modulo B\n");

a = 10; b= 3; //10 % 3 = 1
c = a%b;
$display("c = 10 modulo 3 :: a = %0d b = %0d c = %0d",
a , b, c);

a = -10; b= 3; //-10 % 3 = -1
           //Result takes the sign of the first operand
c = a%b;
$display("c = -10 modulo 3 :: a = %0d b = %0d c = %0d",
a , b, c);

a = 10; b= -3; //10 % -3 = 1
           //Result takes the sign of the first operand
c = a%b;
$display("c = 10 modulo -3 :: a = %0d b = %0d c = %0d",
a , b, c);

a = 0; b= 3; //0 % 3 = 0
c = a%b;
$display("c = 0 modulo 3 :: a = %0d b = %0d c = %0d",
a , b, c);

a = 3; b= 0; //3 % 0 = x
c = a%b;
$display("c = 3 modulo 0 :: a = %0d b = %0d c = %0d",
a , b, c);

end
endmodule

```

*Simulation log:*

### DIVISION

```

c = 10/3 :: a = 10 b = 3 c = 3
integer c = 3.4/2.3 :: a = 3.000000 b = 2.000000 c = 1.000000
real g = 3.4/2.3 :: e = 3.400000 f = 2.300000 g = 1.478261

```

### A to the Power of B

```
integer c = 3.4**2.3 :: a = 3.000000 b = 2.000000 c = 9.000000
real g = 3.4**2.3 :: e = 3.400000 f = 2.300000 g = 16.687893
real g = 9**.5 :: e = 9.000000 f = 0.500000 g = 3.000000
c = 0**3 :: a = 0 b = 3 c = 0
c = 0**-3 :: a = 0 b = -3 c = x
c = -2**3 :: a = -2 b = 3 c = -8
```

### A modulo B

```
c = 10 modulo 3 :: a = 10 b = 3 c = 1
c = -10 modulo 3 :: a = -10 b = 3 c = -1
c = 10 modulo -3 :: a = 10 b = -3 c = 1
c = 0 modulo 3 :: a = 0 b = 3 c = 0
c = 3 modulo 0 :: a = 3 b = 0 c = x
```

### V C S   S i m u l a t i o n   R e p o r t

Let us look at some cases of interest:

```
a = 0; b= -3; //0**-3 = x
c = a**b;
```

Here, “b” is a negative number and “a” is zero. You cannot have a power of a negative number, and hence the result is an “x”:

```
a = -2; b= 3; //-2**3 = -8 :
c = a**b;
```

Here, a = -2 (a negative number) and b = 3. In such a case, the result takes on the sign of the first operand and hence the answer is -8.

Similarly, the following modulo also takes on the sign of the first operand. Hence, the result is -1:

```
a = -10; b= 3; // -10 % 3 = -1
c = a%b;
```

Lastly:

```
a = 3; b= 0; //3 % 0 = x
c = a%b;
```

Here, the modulo divisor is a zero and hence the results are indeterminate. So, c=x.

## 12.4 Relational Operators

Table 12.3 shows the relational operators available in the language.

We have all used relational operators in different programming languages. They return a Boolean pass/fail result. The result will be 0 if comparison fails and 1 if it succeeds. Let us look at an example with some corner cases:

```
module OP;

integer a, b, c, d;
initial begin
    a = 3; b = 2;
    $display("3 > 2 = boolean %0d", a > b);
    $display("3 < 2 = boolean %0d", a < b);
    $display("3 >= 2 = boolean %0d", a >= b);
    $display("3 <= 2 = boolean %0d", a <= b);

    a = 1'b1; b=2; //OR a = 1'b0;
    $display("\nRelational operation with 'x' results in 'x'\n");
    $display(" 'x > 2 = boolean %0d", a > b);
    $display(" 'x < 2 = boolean %0d", a < b);
    $display(" 'x >= 2 = boolean %0d", a >= b);
    $display(" 'x <= 2 = boolean %0d", a <= b);

    $display("\n");
    a = -1; b = -2;
    $display("'"-1 > -2 = boolean %0d", a > b);

    $display("\n");
    a=2; b=3;
    $display("Following two are the same");
    $display("a=2 b=3 : a < b - 1 = %0d", a < b-1);
    $display("a=2 b=3 : a < (b - 1) = %0d", a < (b-1));

    $display("\n");
    $display("Following two are NOT the same");
    $display("a=2 b=3 : b - (1 < a) = %0d", b - (1 < a));
    $display("a=2 b=3 : b - 1 < a = %0d", b - 1 < a);
end
endmodule
```

*Simulation log:*

3 > 2 = boolean 1

**Table 12.3** Relational operators

Operator	Description
$a < b$	a less than b
$a > b$	a greater than b
$a \leq b$	a less than or equal to b
$a \geq b$	a greater than b

$3 < 2 = \text{boolean } 0$

$3 \geq 2 = \text{boolean } 1$

$3 \leq 2 = \text{boolean } 0$

Relational operation with 'x' results in 'x'

$'x > 2 = \text{boolean } x$

$'x < 2 = \text{boolean } x$

$'x \geq 2 = \text{boolean } x$

$'x \leq 2 = \text{boolean } x$

$'-1 > -2 = \text{boolean } 1$

Following two are the same

$a=2\ b=3 : a < b - 1 = 0$

$a=2\ b=3 : a < (b - 1) = 0$

Following two are NOT the same

$a=2\ b=3 : b - (1 < a) = 2$

$a=2\ b=3 : b - 1 < a = 0$

#### V C S   S i m u l a t i o n   R e p o r t

The following is interesting in this example. Note that all the relational operators have the same precedence. Relational operators have lower precedence than arithmetic operators. However, the placement of parenthesis will result different results.

So, the following two are equivalent. In both cases, the first b is subtracted by 1 and then logically compared with "a." See the code/simulation log to see this case:

$a < b - 1$

$a < (b - 1)$

But the following two are not equivalent:

$b - (1 < a)$

$b - 1 < a$

Here, in the first case, 1 is compared with "a" and then the Boolean result subtracted from "b." In the second case, 1 is subtracted from "b" first and then the result is compared with "a." See the code/simulation log to see the difference.

**Table 12.4** Equality operators

Operator	Description
a === b	a equal to b. Compare “x” and “z” as well. Also known as case equality operator. Three “=” signs
a !== b	a not equal to b. Compare “x” and “z” as well. Also known as case inequality operator
a == b	a equal b. “x” and “z” in comparison will produce “x.” Also known as logical equality
a != b	a not equal to b. “x” and “z” in comparison will produce an “x” Also known as logical inequality

## 12.5 Equality Operators

Table 12.4 shows the equality operators offered by the language.

The operators are straightforward to understand. All four equality operators shall have the same precedence. These four operators compare operands bit for bit. As with the relational operators, the result will be 0 if comparison fails and 1 if it succeeds.

Note the difference between the case equality/inequality operators and the logical equality/inequality operators. Case equality version compares with “x” and “z,” while the logical equality operators will result in an “x” when compared with “x” or “z.” Let us look at an example to better understand this concept:

```
module OP;

logic [15:0] A1, A2;

initial begin
    $display("\n NO 'x' in data");
    A1='hff; A2='hf0;
    $display("A1='hff A2='hf0 : A1 == A2= boolean %0d",A1== A2);
    $display("A1='hff A2='hf0 : A1 != A2= boolean %0d",A1!= A2);
    $display("A1='hff A2='hf0 : A1 === A2= boolean %0d",A1=== A2);
    $display("A1='hff A2='hf0 : A1 !== A2= boolean %0d",A1!== A2);

    $display("\n WITH 'x' in data");
    A1='hx5f; A2='hx5f;
    $display("A1='hx5f A2='hx5f : A1 == A2= boolean %0d",A1== A2);
    $display("A1='hx5f A2='hx5f : A1 != A2= boolean %0d",A1!= A2);
    $display("A1='hx5f A2='hx5f : A1 === A2= boolean %0d",
            A1 === A2);
    $display("A1='hx5f A2='hx5f : A1 !== A2= boolean %0d", A1!== A2);
end
endmodule
```

*Simulation log:*

NO 'x' in data

A1='hff A2='hf0 : A1 == A2= boolean 0

A1='hff A2='hf0 : A1 != A2= boolean 1

A1='hff A2='hf0 : A1 === A2= boolean 0

A1='hff A2='hf0 : A1 !== A2= boolean 1

WITH 'x' in data

A1='hxf A2='hxf : A1 == A2= boolean x

A1='hxf A2='hxf : A1 != A2= boolean x

A1='hxf A2='hxf : A1 === A2= boolean 1

A1='hxf A2='hxf : A1 !== A2= boolean 0

V C S   S i m u l a t i o n   R e p o r t

This example shows the difference between logical and case equality operators. First, we assign to A1 and A2 known data. In this case, there is no difference in results between logical and case equality operators.

However, in the second set of operations, we assign A1='hxf and A2='hxf, meaning data with “x” in them (this could be “z” as well). Now, the results differ. Logical equality does not compare “x” with an “x,” and hence the result is unknown “x.” But the case equality operators compare “x” with an “x” and the results are known. The simulation log shows this difference.

Here are the truth tables for logical equality and case equality operators. Tables 12.5 and 12.6 assume 1-bit operands.

As you notice, the case equality results will never be “X.”

Tables 12.7 and 12.8 show examples of equality between vectors.

The case to note in this example is “4'b0011 == 4'b0XX1.” You may expect the result to be 1'bX. But since there is 1-bit (the last bit) that is different between LHS operand and RHS operand, the result will be false (1'b0).

**Table 12.5** Logical equality truth: 1-bit operands

Logical Equality

==	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

**Table 12.6** Case equality truth: 1-bit operands

Case Equality

==	0	1	X	Z
0	1	0	0	0
1	0	1	0	0
X	0	0	1	0
Z	0	0	0	1

**Table 12.7** Logical equality: vector operands**Logical Equality**

4'b0011 == 4'b0110	1'b0
4'b0011 == 4'b0XX1	1'bX
4'b0110 == 4'b0XX1	1'b <b>0</b>
4'b0XX1 == 4'b0XX1	1'bX

**Table 12.8** Case equality: vector operands

<b>Case Equality</b>	
4'b0011 === 4'b0110	1'b0
4'b0011 === 4'b0XX1	1'b0
4'b0110 === 4'b0XX1	1'b0
4'b0XX1 === 4'b0XX1	1'b1

**12.5.1 Wildcard Equality Operators**

Table 12.9 shows the wildcard equality operators.

The wildcard equality operator (==?) and inequality operator (!=? ) treat X and Z values in a given bit

position of their right operand as a wildcard. *X and Z values in the left operand are not treated as wildcards.* A wildcard bit matches any bit value (0, 1, Z, or X) in the corresponding bit of the left operand being compared against it. Any other bits are compared as for the logical equality and logical inequality operators.

Here is an example:

```
module OP;
  logic [15:0] A1, A2;

  initial begin
    A1='hff; A2='hxx;
    $display("A1='hff  A2='hxx : A1==? A2 = boolean %0d",
             A1==? A2);
    $display("A1='hff  A2='hxx : A1!=? A2 = boolean %0d",
             A1!=? A2);

    A1='hxx; A2='hff;
    $display("A1='hxx  A2='hff : A1==? A2 = boolean %0d",
             A1==? A2);
    $display("A1='hxx  A2='hff : A1!=? A2 = boolean %0d",
             A1!=? A2);

  end
endmodule
```

**Table 12.9** Wildcard equality operators

Operator	Description
a ==? b	a equals b, “x” and “z” values in b act as wildcards
a !=? b	a does not equal b, “x” and “z” values in b act as wildcards

*Simulation log:*

```
A1='hff A2='hxx : A1==? A2 = boolean 1
A1='hff A2='hxx : A1!=? A2 = boolean 0
A1='hxx A2='hff : A1==? A2 = boolean x
A1='hxx A2='hff : A1!=? A2 = boolean x
V C S  Simulation Report
```

In this example, first we assign A1= 'hff and A2 = 'hxx, and then we compare A1with A2 using wildcard equality operators. Since A2 is on the right-hand side (RHS) of the comparison, the “x” in it is a wildcard. This means that it will compare any of 1,0,x,z in A1. That being the case, the wildcard ==? will pass, since 'hff is compared with 'hxx where is “x” can be any of 0,1,x,z.

However, if you have an “x” (or a “z”) in the left-hand side (LHS) variable, it is not considered a wildcard. Hence, when A1= 'hxx and A2 = 'hff, the comparison is equivalent to logical equality comparison. And since there is an “x” in A1, the comparison will result in an “x.” Study the code/simulation log to see what is going on.

## 12.6 Logical Operators

Table 12.10 describes the logical operators available in the language.

The result of any of the logical operators is Boolean 1 if true and 0 if false, and “x” is the result if ambiguous. The precedence of && is greater than that of ||.

**Table 12.10** Logical operators

Operator	Description									
&&	Logical AND									
	Logical OR									
->	Logical implication Logically equivalent to (!expression1    expression2)									
<->	Logical equivalence Logically equivalent to ((expression1 -> expression2) && (expression2 -> expression1)) Truth table: <table style="margin-left: auto; margin-right: auto;"> <tr> <th>A &lt;-&gt; B</th> <th>B=false</th> <th>B=True</th> </tr> <tr> <th>A = false</th> <td>true</td> <td>false</td> </tr> <tr> <th>A = true</th> <td>false</td> <td>true</td> </tr> </table>	A <-> B	B=false	B=True	A = false	true	false	A = true	false	true
A <-> B	B=false	B=True								
A = false	true	false								
A = true	false	true								
!	Logical negation									

`A = b && c; //A is set to 0, if "b" is 0 (regardless of what "c" is).`  
`A = b || c; //A is set to 1, if "b" is 1 (regardless of what "c" is).`

Let us look at an example of logical implication and logical equivalence:

```
module OP;
  logic [15:0] A1, A2;
  int i1;

  initial begin
    A1 = 2; //Any number greater than 0
    A2 = 255; //Any number greater than 0

    i1 = A1 -> A2; //logical implication
    $display("A1 = %0d    A2 = %0d A1 -> A2 = %0d",A1,A2,i1);

    i1 = !A1 || A2; //equivalent to A1 -> A2
    $display("A1 = %0d    A2 = %0d !A1 || A2 = %0d",A1,A2,i1);

    i1 = A1 <-> A2;
    $display("A1 = %0d    A2 = %0d A1 <-> A2 = %0d",A1,A2,i1);

    $display("\n");

    A1 = 2;
    A2 = 0;

    i1 = A1 -> A2;
    $display("A1 = %0d    A2 = %0d A1 -> A2 = %0d",A1,A2,i1);

    i1 = A1 <-> A2;
    $display("A1 = %0d    A2 = %0d A1 <-> A2 = %0d",A1,A2,i1);

  end
endmodule
```

*Simulation log:*

```
A1 = 2 A2 = 255 A1 -> A2 = 1
A1 = 2 A2 = 255 !A1 || A2 = 1
A1 = 2 A2 = 255 A1 <-> A2 = 1
```

```
A1 = 2 A2 = 0 A1 -> A2 = 0
A1 = 2 A2 = 0 A1 <-> A2 = 0
```

V C S   S i m u l a t i o n   R e p o r t

The way this works is whenever both “A1” and “A2” are non-zero numbers (i.e., true); both the logical implication and logical equivalence result will be *true* (1). Whenever *both* “A1” and “A2” are false, the logical equivalence will be *true*. Whenever *either* of “A1” or “A2” is a zero, the logical equivalence will be *zero* (0). That is because a *true* implies a *true* but a *false* does not imply a *true*.

## 12.7 Bitwise Operators

As you have guessed, bitwise operators perform bitwise manipulation of the operands. The operator combines 1-bit in one operand with corresponding bit in the other operand to *produce 1 bit result*.

Here are the truth tables for each of the operators. Table 12.11 describes bitwise binary AND (&) operator.

The bitwise binary  $\sim\&$  (NAND) is the reverse operator.

Table 12.12 describes bitwise binary OR (l) operator.

The bitwise binary  $\sim l$  (NOR) is the reverse operator.

Table 12.13 describes bitwise binary exclusive OR (^) operator.

Table 12.14 describes the bitwise binary exclusive ( $^{\sim}$ ,  $\sim^{\wedge}$ ) NOR operator.

Table 12.15 describes the binary unary (~) operator.

Here is an example:

**Table 12.11** Bitwise binary AND (&) operator

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

**Table 12.12** Bitwise binary OR (l) operator

	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

**Table 12.13** Bitwise binary exclusive OR (^) operator

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

**Table 12.14** Bitwise binary exclusive NOR operator

$\sim\wedge$	<b>0</b>	<b>1</b>	<b>x</b>	<b>z</b>
<b>0</b>	1	0	x	x
<b>1</b>	0	1	x	x
<b>x</b>	x	x	x	x
<b>z</b>	x	x	x	x

**Table 12.15** Bitwise unary negation ( $\sim$ ) operator

<b><math>\sim</math></b>	
<b>0</b>	<b>1</b>
<b>1</b>	<b>0</b>
<b>x</b>	<b>x</b>
<b>z</b>	<b>x</b>

```

module OP;

logic [31:0] a1, a2;
int i1;

initial begin
a1 = 'h00_00_00_ff;
a2 = 'hf0_f0_f0_f0;

i1 = a1 & a2;
$display("a1 = %h a2 = %h a1 & a2 = %h",a1,a2,i1);

i1 = a1 | a2;
$display("a1 = %h a2 = %h a1 | a2 = %h",a1,a2,i1);

i1 = a1 ^ a2;
$display("a1 = %h a2 = %h a1 ^ a2 = %h",a1,a2,i1);

i1 = a1  $\sim\wedge$  a2;
$display("a1 = %h a2 = %h a1  $\sim\wedge$  a2 = %h",a1,a2,i1);

i1 = a1  $\sim\sim$  a2;
$display("a1 = %h a2 = %h a1  $\sim\sim$  a2 = %h",a1,a2,i1);

i1 = ~a2;
$display("a2 = %h ~a2 = %h",a2,i1);

end
endmodule

```

*Simulation log:*

```
a1 = 000000ff a2 = f0f0f0f0 a1 & a2 = 000000f0
a1 = 000000ff a2 = f0f0f0f0 a1 | a2 = f0f0f0ff
a1 = 000000ff a2 = f0f0f0f0 a1 ^ a2 = f0f0f0ff
a1 = 000000ff a2 = f0f0f0f0 a1 ~^ a2 = 0f0f0ff0
a1 = 000000ff a2 = f0f0f0f0 a1 ^~ l2 = 0f0f0ff0
a2 = f0f0f0f0 ~a2 = 0f0f0f0f
```

V C S   S i m u l a t i o n   R e p o r t

## 12.8 Unary Reduction Operators

These operators perform a bitwise *operation on a single operand to produce a single bit result*. The operation will first apply between the first bit of the operand and the second bit of the operand. It will then take the result of this operation and apply to the third bit. And so on.

There are six operators: reduction AND, reduction OR, reduction XOR, reduction NAND, reduction NOR, and reduction XNOR. For reduction NAND, reduction NOR, and reduction XNOR operators, the result will be computed by inverting the result of the reduction AND, reduction OR, and reduction XOR operation, respectively. So, here are the truth tables of reduction AND, OR, and XOR.

Table 12.16 describes the reduction unary AND (&) operator.

Table 12.17 describes the reduction unary OR (|) operator.

Table 12.18 describes the reduction exclusive OR (^) operator.

Let us look at an example:

```
module OP;
  logic [7:0] A1;
  int i1;

  initial begin
    A1 = 8'b 10110111;

    i1 = & i1;
    $display("A1 = 10110111 : & i1 = %0d", i1);

    i1 = | i1;
    $display("A1 = 10110111 : | i1 = %0d", i1);

    i1 = ^ i1;
    $display("A1 = 10110111 : ^ i1 = %0d", i1);

    i1 = ~& i1;
    $display("A1 = 10110111 : ~& i1 = %0d", i1);
```

```

i1 = ~| i1;
$display("A1 = 10110111 : ~| i1 = %0d", i1);

i1 = ~^ i1;
$display("A1 = 10110111 : ~^ i1 = %0d", i1);

end
endmodule

```

*Simulation log:*

```

run -all;
# KERNEL: A1 = 10110111 : & i1 = 0
# KERNEL: A1 = 10110111 : | i1 = 1
# KERNEL: A1 = 10110111 : ^ i1 = 0
# KERNEL: A1 = 10110111 : ~& i1 = 1
# KERNEL: A1 = 10110111 : ~| i1 = 0
# KERNEL: A1 = 10110111 : ~^ i1 = 1
# KERNEL: Simulation has finished.

```

**Table 12.16** Reduction unary AND operator

& reduction	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

**Table 12.17** Reduction unary OR operator

reduction	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

**Table 12.18** Reduction unary exclusive OR operator

^ reduction	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

## 12.9 Shift Operators

There are two types of shift operators, logical shift (`<<` (shift left) and `>>` (shift right)) and arithmetic shift (`<<<` (left) and `>>>` (right)). The left shift operators (`<<` and `<<<`) shift their left operand to the left by the number of bits given by the right operand. The vacated bits are filled with zero. Similarly, the right shift operators (`>>` and `>>>`) will shift the left operand to the right by the number of bits given by the right operand, and the vacated bits are filled with zero. The arithmetic right shift fills the vacated bit positions with zeros if the result type is unsigned. It fills the vacated bit positions with the value of the most significant (i.e., sign) bit of the left operand if the result type is signed. If the right operand has an `x` or `z` value, then the result will be unknown. For example:

```
//left operand 'a' is left shifted by right operand '2'  
result = a << 2;
```

Here is an example:

```
module OP;  
logic [7:0] A1, i1;  
  
initial begin  
A1 = 8'b 10110111;  
  
i1 = A1 << 2; //left shift by 2  
$display("A1 = %b : << 2 = %b", A1,i1);  
  
i1 = A1 >> 4; //right shift by 4  
$display("A1 = %b : >> 4 = %b", A1, i1);  
  
A1= -3;  
i1 = A1 <<< 2; //arithmetic left shift by 2  
//Note A1 is negative  
$display("A1 = %b : <<< 2 = %b", A1, i1);  
  
A1= -2;  
i1 = A1 >>> 4 ; //arithmetic right shift by 4  
//Note A1 is negative  
$display("A1 = %b : >>> 4 = %b", A1, i1);  
  
end  
endmodule
```

*Simulation log:*

```
A1 = 10110111 : << 2 = 11011100
A1 = 10110111 : >> 4 = 00001011
A1 = 11111101 : <<< 2 = 11110100
A1 = 11111110 : >>> 4 = 00001111
```

### V C S   S i m u l a t i o n   R e p o r t

In the first case, the number “A1” is 8'b10110111. You left shift it by 2 and fill in the left bits with zero to get 8'b11011100. The same applies to right shift; “A1” = 8'b10110111 is right shifted by 4, and right bits are filled with zero to get 8'b00001011.

For the arithmetic shift, we have chosen negative numbers to make it interesting. In the arithmetic left shift, A1 = -3 which means its two’s complement is 101. You sign extend this to get A1 = 8'b11111101. Then left shift is by 2 to get 8'b11110100. Similarly, when “A1” = -2, its two complement is 110 which when sign extended gives us A1=8'b1111110. You then right shift it by 4 to get i1=8'b00001111.

## 12.10 Conditional Operators

Conditional operator takes on the following form:

```
condition ? expression1 : expression2;
```

If condition is true, expression1 is executed (expression2 is non-consequential). If the condition is false, expression2 is executed (expression1 is non-consequential). If the condition is unknown (x or z), then both expressions are evaluated and compared for logical equivalence (Sect. 12.6). Here is a simple example:

```
wire [15:0] PCIdata = enb ? 'hff : 'z;
```

Drive 'hff onto PCIdata bus if “enb” is true, drive 'z onto the PCIdata if “enb” is false. What if “enb” is unknown? Please see the example below to see how that works:

```
module OP;
  logic [7:0] data1;
  logic enb;

  initial begin

    enb = 1'b1;
    data1 = enb ? 'hff : 'z; //data1 = 'hff
    $display("enb = %0d data1 = %h", enb,data1);
```

```

enb = 1'b0;
data1 = enb ? 'hff : 'z; //data1 = 'z
$display("enb = %0d data1 = %h", enb,data1);

enb = 1'bx;

//data1 = 'x. The two expressions are not equivalent
data1 = enb ? 'hff : 'z;
$display("enb = %0d data1 = %h", enb,data1);

enb = 1'bx;
data1 = enb ? 'hff : 'hff;
           //the two expressions are logically equivalent
$display("enb = %0d data1 = %h", enb,data1);

enb = 1'bx;
data1 = enb ? 'hff : 'h00;
           //data1 = 'x. The two expressions are not equivalent
$display("enb = %0d data1 = %h", enb,data1);
end
endmodule

```

*Simulation log:*

```

enb = 1 data1 = ff
enb = 0 data1 = zz
enb = x data1 = xx
enb = x data1 = ff
enb = x data1 = xx

```

V C S   S i m u l a t i o n   R e p o r t

Here is the explanation:

```

enb = 1'b1; data1 = enb ? 'hff : 'z;
//data1 = 'hff; 'enb' = 1. So, expression1 ('hff) is selected

enb = 1'b0; data1 = enb ? 'hff : 'z;
//data1 = 'z ; 'enb' = 0. So, expression2 ('hz) is selected.

enb = 1'bx; data1 = enb ? 'hff : 'z;
//data1 = 'x; enb = 'x, so expression1 and expression2 are evaluated for logical equivalence. Since they are not equivalent, the result is 'x

enb = 1'bx; data1 = enb ? 'hff : 'hff;

```

```
//data1 = 'hff. Even though enb is 1'bx, the expression1 and expression2 are logically equivalent and hence the result is 'hff.
```

```
enb = 1'bx; data1 = enb ? 'hff : 'h00;
//data1 = 'x. Expression1 and expression2 are not logically equivalent, so the result is 1'bx.
```

A couple of points on how the data type of the result is determined. If both the first expression and second expression are of integral types (meaning the data types that can represent a single basic integer data type, packed array, packed structure, packed union, enum variable, or time variable), the operation proceeds as we have seen. If both expressions are real, then the resulting type is real. If one expression is real and the other expression is shortreal or integral, the other expression is cast to real, and the resulting type is real. If one expression is shortreal and the other expression is integral, the integral expression is cast to shortreal, and the resulting type is shortreal.

## 12.11 Concatenation Operators

When you join together the bits resulting from one or more expressions, it is called concatenation of expressions. You will see examples of such concatenation throughout the book. Concatenation is expressed using the brace characters {and}, with comma separated list of expressions. For example, the following is a concatenation of a[2:0], b[1:0], i1, and 2'b01:

```
{a[2:0], b[1:0], i1, 2'b01};
```

This is equivalent to:

```
{a[2], a[1], a[0], b[1], b[0], i1, 1'b0, 1'b1};
```

The concatenation is treated as a packed vector of bits. Here is an example:

```
module OP;
  logic [7:0] data1, data2, data3;
  logic A1, A2, A3;
  logic enb;
  string s1;
```

```
initial begin
    data1 = {4'b 0101, 4'b 1010};
    $display("data1 = %b",data1);

    A1 = 1'b0;
    A2 = 1'b1;
    A3 = 1'b0;
    enb = 1'b0;

    data1 = {A1, A2, A3, enb, 4'b 1010};
    $display("data1 = %b",data1);

    {A1, A2, A3} = 3'b101;

    data3 = 8'b 0;
    data2 = 8'b 0000_0001;

    data1 = {data2 + data3}[7:0];
    $display("data1 = %b",data1);

    s1 = {"hi", " ", "later"};
    $display("string s1 = %p",s1);
end
endmodule
```

*Simulation log:*

```
data1 = 01011010
data1 = 01001010
data1 = 00000001
string s1 = "hi later"
V C S  S i m u l a t i o n  R e p o r t
```

## 12.12 Replication Operators

This is also a concatenation operator only that you can have multiple concatenations (i.e., a replication) with this operator. It is expressed by a concatenation preceded by a non-negative, non-x (or z) constant expression. The other difference with regular concatenation is that the replication concatenation cannot appear on the left-hand side of an assignment. Here is an example:

```

module OP;
  logic [7:0] data1, data2, data3;
  logic A1, A2, A3;
  logic enb;
  string s1;

  initial begin
    data1 = {8{1'b0}}; //data1 = 00000000
    $display("data1 = %b",data1);

    enb = 1;
    data1 = {8{enb}}; //data1 = 11111111
    $display("data1 = %b",data1);

    data1 = { {4{enb}} , 4'b0}; //data1 = 11110000
    $display("data1 = %b",data1);

    data1 = { {0{1'b1}} , 4'b1111};
    //{0{1'b1}} is ignored; data1=00001111
    $display("data1 = %b",data1);
  end
endmodule

```

*Simulation log:*

```

data1 = 00000000
data1 = 11111111
data1 = 11110000
data1 = 00001111
  V C S  S i m u l a t i o n  R e p o r t

```

The following is interesting:

```
data1 = { {0{1'b1}} , 4'b1111};
```

Here we are using the replication operator of “0” (zero). A replication with a zero-replication constant is considered to have a size of zero and is ignored. Hence, we get the following result:

```
data1 = 00001111
```

## 12.13 Streaming Operators (pack/unpack)

The streaming operators (<< or >>) perform packing and unpacking of bit-stream types into a sequence of bits in a user-specified order. For example, if you have an array of bytes and you want to turn it into an int, or an array of ints that you want to

turn into an array of bytes, or if you have a class instance that you want to turn into a stream of bits, then streaming comes in handy.

Here is the format:

```
streaming_concatenation ::= { stream_operator [ slice_size ]
    stream expression, stream expression, ... }
```

where stream operator is `<<` or `>>` and slice\_size is of simple type or a constant integral.

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 groups of any number of bits. The default is 1-bit at a time, if a size is not specified. Unpacking occurs when a streaming operator is used on the left-hand side of an assignment.

`>>` causes *blocks of data* to be streamed in left-to-right order, while `<<` causes blocks of data to be streamed in right-to-left order. Left-to-right streaming using `>>` will cause the slice\_size to be ignored and no re-ordering performed. Right-to-left streaming using `<<` will reverse the order of blocks in the stream, preserving the order of bits within each block.

When used in the left-hand side, the streaming operators perform the reverse operation, i.e., unpack a stream of bits into one or more variables. For right-to-left streaming using `<<`, the stream is sliced into blocks with the specified number of bits, starting with the right-most bit.

If the data being packed contains any 4-state types, the result of a pack operation is a 4-state stream; otherwise, the result of a pack is a 2-state stream.

The slice\_size determines the size of each block, measured in bits. If a slice\_size is not specified, the default is 1. If a type is used, the block size will be the number of bits in that type. If a constant integral expression is used, it cannot be zero or negative.

Note: A bit-stream type is any integral, packed, or string type and unpacked array structures or classes of these types. They can also be dynamically sized arrays (dynamic, associative, and queues).

But before we go further, what is “left” and what is “right”? Seems obvious but consider the following packed array:

```
bit [7:0] pArray = 8'b 0100_0101;
```

Since this is a packed array, `pArray[0]` will be the right most bit (“1” in our case). But if you have an unpacked array:

```
bit upArray [ ] = '{0, 1, 0 ,1};
```

The `upArray[0]` will be the left most bit “0.” So, you need to be careful in coding with packed and unpacked arrays, or the results may not make sense.

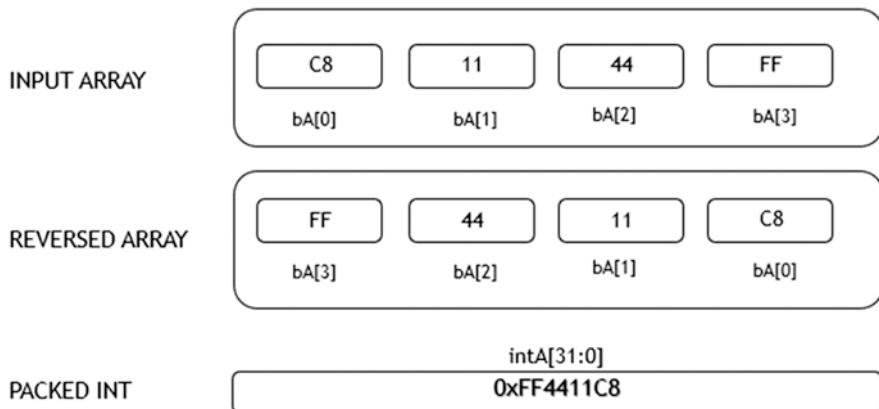


Fig. 12.1 Byte Array elements packed into an “int”

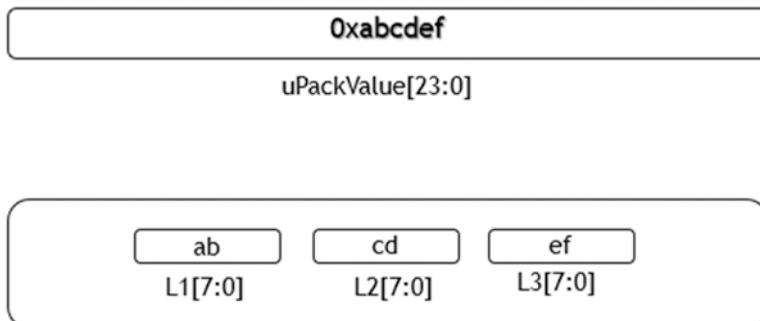


Fig. 12.2 Unpacking of bits

### 12.13.1 *Packing of Bits*

Here is a simple example of packing bytes into an int. Four individual bytes in memory will be packed into a 32-bit int in memory:

```
module byteToInt;
initial begin
  //four individual bytes
  static byte b1 = 8'h00;
  static byte b2 = 8'h11;
  static byte b3 = 8'h22;
  static byte b4 = 8'h33;

  //packed into an int
  static int intA = {>>{b1, b2, b3, b4}};
end
```

```
//Left-to-Right packing

$display("intA = 0x%h", intA);
end
endmodule
```

*Simulation log:*

intA = 0x00112233

V C S   S i m u l a t i o n   R e p o r t

Similarly, we can reverse the order of an array's elements and then pack them into a single value:

```
module tarray;
initial begin
    //array of four bytes
    static bit [7:0] bA[4] = '{ 8'hC8, 8'h11, 8'h44, 8'hFF };

    //reversed and packed into an int
    static int      intA = {<<8 {bA} }; //Right-to-Left packing

    $display("intA = 0x%h", intA);
end
endmodule
```

*Simulation log:*

intA = 0xff4411c8

V C S   S i m u l a t i o n   R e p o r t

Note that packing is done in blocks of 8-bits.

Here is the graphic representation of the code (Fig. 12.1).

In the above example, let us see how the following line works:

```
int      intA = {<<8 {bA} };
```

Here right-to-left streaming means that it will reverse the order of elements in the input array. bA[4] means the elements are bA[0] = C8, bA[1] = 11, bA[2] = 44, and bA[3] = FF. The “<<” operator will first reverse the order of these elements and then pack them resulting in the following:

```
intA = 0xff4411c8
```

Here is another example showing different slice sizes and their effect on reversal of data and packing. This example shows how “packing” works. We will later see an example on how “unpacking” works:

```

module stream;

string str = { "A", "B", "C", "D" };
string str2;
bit [7:0] i1, i2, i3;

initial begin

    //slice_size ignored - no re-ordering
    str2 = { >> byte {str} }; //Left-to-Right
    $display(">> byte: str = %s str2 = %s",str,str2);

    //slice_size (block size) = byte - reverse ordering
    //reverse byte block. Stream from right to left
    str2 = { << byte {str} }; //Right-to-Left
    $display("<< byte: str = %s str2 = %s",str,str2);

    //slice_size (block size) = 16 - reverse ordering
    //reverse 16-bit block
    str2 = { << 16 {str} }; //slice_size (block size) = 16
    $display("<< 16: str = %s str2 = %s",str,str2);

    //slice_size (block size) = 32 - no re-ordering
    str2 = { << 32 {str} };
    $display("<< 32: str = %s str2 = %s",str,str2);

    //reverse bits in a byte
    i1 = 8'b1101_0011;
    i2 = { << {i1} };
    $display("<< : i1 = %b i2 = %b",i1,i2);

    //reverse nibbles in a byte
    i1 = 8'b1111_0000;
    i2 = { << 4 {i1} };
    $display("<< 4: i1 = %b i2 = %b",i1,i2);
end
endmodule

```

*Simulation log:*

```

>> byte: str = ABCD str2 = ABCD
<< byte: str = ABCD str2 = DCBA
<< 16: str = ABCD str2 = CDAB

```

```
<< 32: str = ABCD str2 = ABCD
<< : i1 = 11010011 i2 = 11001011
<< 4: i1 = 11110000 i2 = 00001111
VCS Simulation Report
```

An interesting point in this code.

Consider the following:

```
str2 = { << 32 {str} };
```

In this case, we are streaming right to left a 4-byte (32-bit) string. Since the slice size is 32-bits and so is the size of “str,” there cannot be any reversal. Reversal is the same as original order. Hence, the answer in simulation log is:

```
<< 32: str = ABCD str2 = ABCD
```

Also, let us understand the following:

```
str2 = { << 16 {str} };
```

This means that it will create a stream taking the right most 16-bit chunk of “str” (which is [15:0]) first, followed by the chunk to the left ([31:16]), effectively giving you (str[15:0], str[31:0]) resulting in the following result:

```
<< 16: str = ABCD str2 = CDAB
```

Here is an example of packing/converting between a word queue and a byte queue:

```
module queue;
  bit [15:0] wordQ [$] = {'h abcd, 'h5678 };
  bit [7:0] byteQ [$];

  initial begin
    byteQ = { >> {wordQ} }; //pack word queue into byte queue
    $display("byteQ = %p", byteQ);

    wordQ = { >> {byteQ} }; //pack byte queue into word queue
    $display("wordQ = %p", wordQ);
  end
endmodule
```

*Simulation log:*

```
byteQ = {'hab, 'hcd, 'h56, 'h78}
```

```
wordQ = {'habcd, 'h5678}
```

```
VCS Simulation Report
```

This is an interesting example. byteQ is an 8-bit queue and wordQ is a 16-bit queue. When we do the following:

```
byteQ = { >> {wordQ} };
```

We are packing a word width variable into a byte width variable. Since byteQ is byte wide, the word will be converted to a byte as part of packing operation. This is shown in simulation log:

```
byteQ = {'hab, 'hcd, 'h56, 'h78}
```

Similarly, the following will take the generated byte queue and pack into a word queue:

```
wordQ = { >> {byteQ} };
```

as seen from the simulation log as:

```
wordQ = {'habcd, 'h5678}
```

### ***12.13.2 Unpacking of Bits***

Let us now see how unpacking works. Unpacking involves taking a packed array of bits and unpacking them into specified bit size. For example, taking a 32-bit word and unpacking it into four 8-bit bytes. Here is an example to do just that.

Note that, unlike packing of bits, for unpacking we have the streaming operator on the left-hand side of the statement as:

```
{<< or >> {bits or expression}} = bits or expression;
```

Let us look at an example. First the graphical representation (Fig. 12.2).

We take a 24-bit value '`h abcdef`' and unpack it into three 8-bit values, A1, A2, and A3. Here is the code:

```
module unpack;
  logic [7:0] A1, A2, A3;
  logic [23:0] uPackValue;

  initial begin

    uPackValue = 24'h abcdef;

    {>>{ A1, A2 ,A3 }} = uPackValue; //unpack
```

```

$display(" uPackValue = %h A1 = %h A2 = %h A3 = %h",
         uPackValue, A1, A2, A3);

{<<{ A1, A2 ,A3 }} = uPackValue; //unpack and reverse
$display(" uPackValue = %b A1 = %b A2 = %b A3 = %b",
         uPackValue, A1, A2, A3);

{<< 4 { A1, A2 ,A3 }} = uPackValue; //unpack and reverse
$display(" uPackValue = %h A1 = %h A2 = %h A3 = %h",
         uPackValue, A1, A2, A3);

end
endmodule

```

*Simulation log:*

```

uPackValue = abcdef A1 = ab A2 = cd A3 = ef
uPackValue = 10101011100110111101111 A1 = 11110111 A2 = 10110011 A3 =
11010101
uPackValue = abcdef A1 = fe A2 = dc A3 = ba
      V C S   S i m u l a t i o n   R e p o r t

```

We assign a 24-bit value '`'h abcdef` to the variable `uPackValue`. This is a packed value. We want to unpack it into three 8-bit values. This is done via the following statement using left-to-right streaming operator:

```
{>>{ A1, A2 ,A3 }} = uPackValue;
```

The simulation log shows the result of this operation. The simulation log shows that the 24-bit value has been unpacked into three 8-bit values:

```
uPackValue = abcdef A1 = ab A2 = cd A3 = ef
```

Similarly, we unpack and reverse the bit order by the following statement using right-to-left streaming operator:

```
{<<{ A1, A2 ,A3 }} = uPackValue;
```

The simulation log shows that the bits of `A1`, `A2`, and `A3` have been reversed (bit by bit). Reverse the entire `uPackValue` and then unpack into three 8-bit values:

```
uPackValue = 10101011100110111101111 A1 = 11110111 A2 = 10110011 A3 =
11010101
```

In this case, we take the last 8-bits of `uPackValue` (i.e., `11101111`), reverse it (because we are going right to left), and assign to `A1`. Hence, `A1 = 11110111` (bit-by-bit reversal). Similar analogy applies to `A2` and `A3` calculation.

Finally, we do the same as above but in slice\_size of 4 (meaning unpack in chunks of 4-bits):

```
{<< 4 { A1, A2 ,A3 } } = uPackValue;
```

uPackValue has been reversed in chunks of 4-bits as shown in the simulation log.  
Reverse the entire uPackValue, one nibble at a time:

uPackValue = abcdef A1 = fe A2 = dc A3 = ba

Here is an example of taking an 8-bit packed array and unpacking it into an unpacked array of eight locations:

```
module example_2;
initial begin
    static bit [7:0] p_array = 8'b1101_0011;
    static bit      up_array[8];
    {>>{up_array}} = p_array; //unpack p_array into up_array

    for (int i=0; i<8; i++)
        $display("up_array[%0d] = %b", i, up_array[i]);
end
endmodule
```

*Simulation log:*

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

VCS Simulation Report

Here is an example of unpacking an array into the fields of a structure ( (Amiq, consulting) ):

```
module arrayTostruct;
typedef struct {
    bit [3:0] address;
    bit [3:0] data;
} bus;
```

```
//dynamic array
bit [1:0] array[ ] = '{ 2'b10, 2'b01, 2'b11, 2'b00 };
bus bus1;

initial begin
  //unpack into structure
  {>>{bus1.address, bus1.data}} = array;

  $display("bus1 address = %b", bus1.address);
  $display("bus1 data     = %b", bus1.data);
end
endmodule
```

*Simulation log:*

bus1 address = 1001

bus1 data = 1100

V C S   S i m u l a t i o n   R e p o r t

## 12.14 “inside” Operator (Set Membership Operator)

The “inside” operator allows to check if a given value lies within a given range. It is very useful in constraint random verification which we will study in Chap. 13. The syntax is:

```
<expression> inside {expression or <values or range>}
```

The expression on the left-hand side of the inside operator is any singular expression. The right-hand side of “inside” operator is a comma-separated list of expressions or ranges. For example:

```
data inside {[5:10]} //inclusive of 5 and 10
data inside {1,2,3,4}
data inside {1,3,[5:10],11,12,[13:15]}
```

Here is a simple example:

```
module OP;
  bit [3:0] bus_data;

  initial begin
    for (int i = 0; i < 5; i++) begin
      bus_data = $urandom;
```

```

if (bus_data inside {1,2,[4:9],10})
    $display ("bus_data=%0d 'inside' {1,2,[4:9],10} ",
bus_data);
else
    $display ("bus_data=%0d outside {1,2,[4:9],10} ",
bus_data);
end
end
endmodule

```

*Simulation log:*

```

bus_data=4 'inside' {1,2,[4:9],10}
bus_data=1 'inside' {1,2,[4:9],10}
bus_data=9 'inside' {1,2,[4:9],10}
bus_data=3 outside {1,2,[4:9],10}
bus_data=13 outside {1,2,[4:9],10}

```

### VCS Simulation Report

As you notice in the simulation log, whenever bus\_data is within the {1,2,[4:9],10} range, it is considered “inside” that range; else it is outside.

Let us look at how SystemVerilog “constraint” can effectively use the “inside” operator. Constraint random verification is discussed in Chap. 13:

```

class myClass;
    rand bit [3:0]    cVar, cVar2;

    // Inverted inside: Constrain c_var to be outside 3 to 7
    constraint c_var { !(cVar inside {[3:7]}); } //OUTSIDE

    //inside: Constrain c_var2 to be inside 3 to 7
    constraint c_var2 { cVar2 inside {[3:7]}; } //INSIDE

endclass

module TOP;

initial begin
    myClass mC = new( ); //instantiate/construct class

    $display("Constraint cVar to be outside 3 to 7");
    repeat (5) begin
        mC.randomize( ); //randomize variables of class
        $display("mC.cVar = %0d", mC.cVar);
    end
endmodule

```

```

    end

    $display("Constraint cVar2 to be inside 3 to 7");
    repeat (5) begin
        mC.randomize();
        $display("mC.cVar2 = %0d", mC.cVar2);
    end

end
endmodule

```

*Simulation log:*

Constraint cVar to be outside 3 to 7

mC.cVar = 10

mC.cVar = 10

mC.cVar = 2

mC.cVar = 13

mC.cVar = 8

Constraint cVar2 to be inside 3 to 7

mC.cVar2 = 4

mC.cVar2 = 6

mC.cVar2 = 7

mC.cVar2 = 3

mC.cVar2 = 5

### V C S   S i m u l a t i o n   R e p o r t

We declare a class called myClass in which we declare *rand* (random) variables cVar and cVar2. “rand” means that we can randomize these variables. We then provide constraint on each of the “rand” variable. This way the class variables can be randomized, but these random values can be constrained to keep them in meaningful range.

Note that we randomize the variables two different ways. One way is to constraint them with values that fall *outside* of the constraint range (inverted “inside”). The other is to constraint the values to be *inside* the constraint range.

In module TOP, we use the “.randomize” method to randomize the variables of class myClass and print out the constrained values generated, as shown in the simulation log. Note the difference between the values generated for outside the constraint range vs. inside the constraint range.

# Chapter 13

## Constrained Random Test Generation and Verification



**Introduction** This chapter describes the constrained random verification methodology and discusses how to generate constrained random values and use them effectively for functional verification. The chapter discusses, among other things, rand/randc variables, randomization of arrays and queues, constraint blocks, weighted distribution, iterative constraints, soft constraints, randomization methods, system functions/methods, random stability, randcase/randsequence, productions, etc.

Constrained random verification (CRV) is a methodology that allows you to constrain your stimulus to better target a design function, thereby allowing you to reach your coverage goal faster with accuracy. From that sense, functional coverage and CRV go hand in hand. You check your functional coverage and see where the coverage holes are. You then constrain your stimulus to target those holes and improve coverage.

As part of verification strategy, you start with directed testing to target *directly* the features that you need to verify. But directed testing runs out of steam very soon. If you jump straight to random, you may or may not hit the corner cases of importance. Fully random can end up wasting a lot of simulation cycles without improving coverage. It may also exercise many invalid cases. That is where constrained random comes into picture. By specifying constraints, one can easily create tests that can find hard-to-reach corner cases. SystemVerilog allows users to specify constraints in a compact, declarative way. The constraints are then processed by a solver that generates random values that meet the constraints.

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_13](https://doi.org/10.1007/978-3-030-71319-5_13)) contains supplementary material, which is available to authorized users.

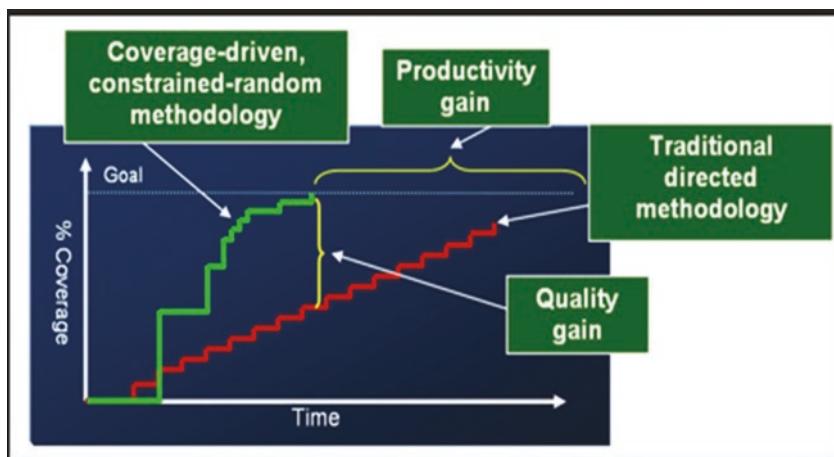
### 13.1 Productivity Gain with CRV

Figure 13.1 shows a chart used by EDA vendors to highlight the productivity and quality gain with the use of constrained random stimuli. The Y-axis is the coverage achieved, and the X-axis is the time it took to achieve that coverage. As you notice, directed testing may take you longer to reach your goal and at the expense of lengthy time in test development, debug, and simulation. This is because for every corner case you will be creating a new testcase and hope against hope that you will reach that corner case. A lot of trial and error will take place. The debug time will dramatically increase as the number of tests increases, not to mention the simulation and regression time.

In contrast, if you understand the logic that is not covered and constrain and randomize your stimuli to target that logic, you will not only get to the coverage goal faster but also will find some of those hidden corner cases that you had not even envisioned.

CRV is not new. What is new is that SystemVerilog has incorporated an exhaustive constraint solver that allows you to constrain your stimuli in a logical and organized way. The language semantics are easy to understand and easy to deploy.

Let us look at a simple CRV methodology that shows the need and importance of having coverage as an integral part of your verification methodology and CRV as part of that methodology to cover missing gaps in functional coverage.



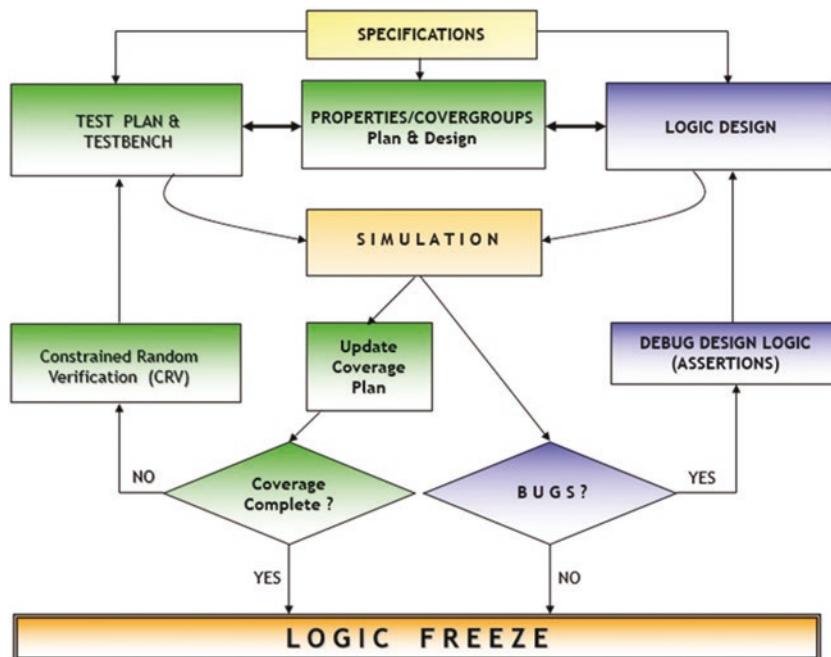
**Fig. 13.1** Advantages of constrained random verification

## 13.2 Constrained Random Verification (CRV) Methodology

Figure 13.2 shows the functional verification methodology in a nutshell. Traditionally, both design and verification teams start running with the specifications written by the architects of the design. The DV (design verification) team starts putting together a verification architecture/environment, test plan, and a testbench. Tests are written per the test plan and the verification cycle begins. The key component missing in this traditional methodology is planning for coverage (and assertions). Without a comprehensive coverage plan, the team has no idea how their tests and testbenches are performing. They use code coverage (if) at the most.

This is where CRV and coverage marry. The first step is to create a functional coverage plan (Chap. 15). Once the coverage plan is ready, you measure the coverage after each simulation (regression) run. This process can also be automated, and the major EDA vendors provide just such mechanisms.

If the coverage is not complete, you identify the holes in your coverage results. Since your directed tests are leaving holes in the coverage results (think corner cases), you now need to move onto constrained random stimuli. Constrained random allows you to narrow down your stimuli to those areas where coverage is lacking. Now, you design your stimulus with constraints. There are many ways to place constraints on your stimulus, which are explained in the coming sections.



**Fig. 13.2** Constrained random verification (CRV) methodology

After you constrain your stimulus, you go through the simulation cycle and repeat the coverage cycle and further identify remaining coverage holes. You further apply constraints to your stimulus and repeat the entire loop.

Functional coverage with CRV is an objective methodology in that you know objectively if you are done verifying your chip (as opposed to subjective measure where as soon as you stop finding bugs, you may stop simulations).

### 13.3 SystemVerilog Support for CRV

Constraint-driven test generation allows you to automatically (well almost) generate tests for functional verification. SystemVerilog offers a rich set of features toward the constrained random verification methodology. We will discuss each of these features in the coming sections. But at the high level, the following are some of the features that are supported:

- Constraints: Allows you to specify interesting subset of all possible stimuli with constraint blocks. Purely random stimulus may not allow you to focus on specific set of verification criteria. The constraint blocks allow for constraining randomness. The following features support constraints:
  - Constraints can be any SystemVerilog expression with variables and constants of integral type (e.g., bit, reg, logic, integer, enum, and packed struct).
  - The constraint solver will be able to handle a wide spectrum of equations, such as algebraic factoring, complex Boolean expressions, and mixed integer and bit expressions. We will see more examples on this.
  - “foreach” for constraining elements of an array.
  - Inline constraints.
  - Guarded constraints.
  - Conditional (if-else) and implication ( $\rightarrow$ ) constraints.
  - Rand case and rand sequence.
- Randomization: The following features support randomization:
  - System functions for randomization
  - Constrained and unconstrained randomization
  - Uniform and weighted distribution
  - Pre- and post-randomization
  - Random variables
- Dynamic constraints:
  - Disabling/enabling constraints
  - Disabling/enabling random variables
  - Overriding constraint blocks
- Random stability:
  - Thread stability
  - Object stability

## 13.4 Constraints

The random constraints are typically specified on top of a “class”-based object-oriented data abstraction. Such abstraction models the data to be randomized as objects that contain random variables and user-defined constraints. The constraints determine the legal values that can be assigned to the random variables. Objects are ideal for representing complex aggregate data types and complex protocols such as Ethernet packet generation. Constraints are expressions that need to be held true by the constraint solver when solving a randomization problem. Constraint expressions may include random variables, non-random state variables, operators, distributions, literals, and constants.

Using classes to model the data to be randomized is a powerful mechanism that enables the creation of generic, reusable objects containing random variables and constraints that can be later extended, inherited, constrained, overridden, enabled, disabled, and merged with or separated from other objects. The ease with which classes and their associated random variables and constraints can be manipulated makes classes an ideal vehicle for describing and manipulating random data and constraints.

Constraint programming is a powerful method that lets users build generic, *reusable objects* that can later be extended or constrained to perform specific functions.

SystemVerilog uses an object-oriented method for assigning random values to the (“rand” and “randc”) member variables of an object, subject to user-defined constraints.

The syntax for specifying constraints is:

```
constraint <constraint_identifier> <constraint_block>
```

Let us look at a simple example that uses layered constraints:

```
module top;

class PacketBase;
    rand bit [7:0] src;
    rand bit [5:0] len;
    rand bit [7:0] payld [ ];
    constraint payload_size {payld.size > 0; payld.size < 15;};
endclass

class EtherPacket extends PacketBase;
    constraint c1 {src inside {[8'h2A:8'h2F]};};
    constraint c2 {len inside {[0:31]}; }
    constraint payload_size {payld.size > 15; payld.size < 20;};
    constraint c3 {foreach (payld[i]) { (payld[i]) == 'haa;} }

    function show;
        $display("EtherPacket = %h %h %h", src, len, payld);
    endfunction
endclass
endmodule
```

```

endfunction:show
endclass

class stim_gen;

task run;
    EtherPacket e1;
    e1 = new( );

    for(int i = 0; i<8; i++) begin
        e1.randomize;
        e1.show;
    end

endtask

endclass: stim_gen

initial begin
    stim_gen st1;

    st1 = new( );
    st1.run;
end
endmodule

```

*Simulation log:*

```

# run -all
# EtherPacket = 2c 0a aa aa
# EtherPacket = 2f 0d aa aa
# EtherPacket = 2d 1d aa aa
# EtherPacket = 2b 06 aa aa
# EtherPacket = 2a 09 aa aa
# EtherPacket = 2c 0f aa aa
# EtherPacket = 2f 03 aa aa
# EtherPacket = 2d 1e aa aa
# exit

```

A class named PacketBase uses “rand” keyword for variables src, len, and payload. These are the variables that can be randomized. Then we provide a constraint on the “payld.” Constraints allow us to restrict the randomized values to what we really want to exercise, the valid cases to reach our coverage goals:

```
constraint payload_size {payld.size > 0; payld.size < 15;}
```

We then extend the PacketBase class to EtherPacket class. Here we constrain “src” and “len” and further constrain “payld” size and “payld” to ‘h aa. This shows that you can build layered constraints in extended classes. Each extended class can have its own constraints as well as override those in the base class:

```
constraint c1 {src inside {[8'h2A:8'h2F]};}
constraint c2 {len inside {[0:31]}; }
constraint payload_size {payld.size > 15; payld.size < 20; }
constraint c3 {foreach (payld[i]) { (payld[i]) == 'haa;}}
```

The class “stim\_gen” instantiates EtherPacket and uses the method “randomize( )” to randomize the “rand” variables of PacketBase (and extended class EtherPacket). Calling randomize( ) causes new values to be selected for all the random variables in an object so that all the constraints are true (satisfied). We do not have any unconstrained variables in this example. But if we did, unconstrained variables are assigned any value in their declared range. We call the following randomization call on object “e1” in class stim\_gen:

```
e1.randomize;
```

The randomize( ) method generates random values for all the active random variables of an object, subject to the active constraints. Variables declared with the rand or randc keyword will get random values on the object.randomize( ) method call. The randomize( ) method returns 1 if the randomization is successful, i.e., on randomization it is able to assign random values to all the random variables; otherwise, it returns 0.

The built-in class method randomize( ) is used to randomize class fields with rand/randc qualifiers according to predefined constraints. It can be called to recursively randomize all random variables of a class or to randomize specific variable(s) as well (either defined with the rand qualifier or not), keeping all predefined constraints satisfiable.

As you notice in the example, we further constrain “payload\_size” in extended class EtherPacket. Using such inheritance to build layered constraint systems enables the development of general-purpose models that can be constrained to perform application-specific functions.

Note also the use of “foreach” loop inside the “c3” constraint to constrain an array. The “foreach” loop iterates over the elements of the array, and its count is determined by payld.size (which is itself constrained):

```
constraint c3 {foreach (payld[i]) { (payld[i]) == 'haa; } }
```

Simulation log shows the constrained values of src, len, and payld.

Note that you *cannot* make assignments inside a constraint block since it only contains expressions. In other words, you cannot use “=” sign for assignments; you have to use the “==” equivalence operator.

Here is another simple example in which we restrict the values of “addr” between 60 and 61:

```
class aClass;
    rand bit [15:0] addr;
    constraint c1 {addr < 62; }
    constraint c2 {addr >= 60; }
endclass

module tb;
    initial begin
        aClass aC = new ();
        for (int i = 0; i < 10; i++) begin
            aC.randomize();
            $display ("i=%0d addr=%0d", i, aC.addr);
        end
    end
endmodule
```

#### *Simulation log:*

```
i=0 addr=60
i=1 addr=60
i=2 addr=60
i=3 addr=61
i=4 addr=61
i=5 addr=60
i=6 addr=61
i=7 addr=60
i=8 addr=61
i=9 addr=61
```

The example declares a rand variable “addr” and applies two constraints to it. The first constraint requires  $addr < 62$ , while the other requires  $addr \geq 60$ . This way we restrict “addr” to be between 60 and 61. This example illustrates how you can build layered constraints (constraint on constraint) to limit the values in a certain range.

Note that constraint blocks are not executed from top to bottom like procedural code but are all active at the same time.

Note also that you need to be careful in planning your constraints. You cannot contradict them among each other. For example, if in the above example, you end up doing the following:

```

class aClass;
  rand bit [15:0] addr;
  constraint c1 {addr < 62; }
  constraint c2 {addr >= 60; }
  constraint c3 {addr == 50; } //This contradicts above
constraints
endclass

```

This will generate a run-time error which will look like the following (Synopsys – VCS):

```

=====
Solver failed when solving following set of constraints

rand bit[15:0] addr; // rand_mode = ON

constraint c2      // (from this) (constraint_mode = ON) (test-
bench.sv:5)
{
  (addr >= 60);
}
constraint c3      // (from this) (constraint_mode = ON) (test-
bench.sv:6)
{
  (addr == 50);
}
=====
```

### ***13.4.1 Constraints: Turning On and OFF***

Let us look at another example that illustrates how to turn on/off the constraints and also how to use the “with” clause:

```

module crv;

class PCIBus;
  rand bit[15:0] addr;
  rand bit[31:0] data;

  constraint addrw {addr[7:0] == 'h01; }
  constraint dataaw {data[15:0] == 'hffff; }
```

```

endclass

logic [31:0] result;

initial begin
    PCIBus pBus = new ( ); //Instantiate PCIBus

    $display("\n pBus randomize");
    repeat (2) begin
        pBus.randomize; //randomize pBus
        $display ("addr = %h data = %h", pBus.addr, pBus.data);
    end

    $display("\n pBus randomize with constraint 'addrw' and 'dataw' turned OFF");
    pBus.addrw.constraint_mode(0); //turn OFF constraint
    pBus.dataw.constraint_mode(0); //turn OFF constraint
    repeat (2) begin
        pBus.randomize;
        $display ("addr = %h data = %h", pBus.addr, pBus.data);
    end

    $display("\n pBus randomize with constraint 'addrw' and 'dataw' turned ON");
    pBus.addrw.constraint_mode(1); //turn ON constraint
    pBus.dataw.constraint_mode(1); //turn ON constraint
    repeat (2) begin
        pBus.randomize;
        $display ("addr = %h data = %h", pBus.addr, pBus.data);
    end

    $display("\n constraint 'data' using the 'with' clause");
    repeat (3) begin
        pBus.randomize ( ) with {pBus.data >= 'hffff && pBus.data <= 'hf_ffff;};
        $display ("data = %h", pBus.data);
    end

end
endmodule

```

*Simulation log:*  
 pBus randomize

```
addr = 3501 data = ad77ffff
addr = f401 data = 3de0ffff
```

pBus randomize with constraint 'addrw' and 'dataaw' turned OFF  
 addr = 0966 data = acf08845  
 addr = 966a data = 2e4c76be

pBus randomize with constraint 'addrw' and 'dataaw' turned ON  
 addr = 0101 data = 2bbeffff  
 addr = e401 data = cb1cffff

constraint 'data' using the 'with' clause  
 data = 0000ffff  
 data = 0002ffff  
 data = 0005ffff

#### V C S   S i m u l a t i o n   R e p o r t

In this example, we declare a class called “PCIBus” with two random variables “addr” and “data.” We also constrain these two variables as shown in the code. In the initial block, we first randomize the instance “pBus” and see its effect on “addr” and “data”:

```
pBus.randomize;
```

The result of this randomization is shown in the simulation log. As you notice, addr[7:0] remains 'h01 as per the constraint and the data[15:0] remains 'hffff:

```
pBus randomize
addr = 3501 data = ad77ffff
addr = f401 data = 3de0ffff
```

We then use the method “constraint\_mode ( )” to turn off the constraints in class PCIBus as shown below:

```
pBus.addrw.constraint_mode(0); //turn OFF constraint
pBus.dataaw.constraint_mode(0); //turn OFF constraint
```

Once we turn off the constraints, you will notice in the simulation log that addr[7:0] is no longer restricted to 'h01 and so is the data not restricted to 'hffff. Here is the simulation log:

pBus randomize with constraint 'addrw' and 'dataaw' turned OFF  
 addr = 0966 data = acf08845  
 addr = 966a data = 2e4c76be

We now turn on the constraints:

```
pBus.addrw.constraint_mode(1); //turn ON constraint
pBus.dataaw.constraint_mode(1); //turn ON constraint
```

Since the constraints are on again, `addr[7:0]` will again be restricted to 'h01 and data to 'hffff:

```
pBus randomize with constraint 'addrw' and 'dataw' turned ON
addr = 0101 data = 2bbeffff
addr = e401 data = cb1cffff
```

Finally, we use the “with” clause to further constrain “data” using the following:

```
pBus.randomize ( ) with {data >= 'hffff && data <= 'hf_ffff; };
```

You will see in the simulation log that “data” is now constrained with this new constraint which is layered on top of the original constraint in class PCIBus. Here is the simulation log that shows this constraint:

```
constraint 'data' using the 'with' clause
data = 0000ffff
data = 0002ffff
data = 0005ffff
```

The `constraint_mode( )` method can be used to enable or disable any named constraint block in an object. The ability to enable or disable constraints allows users to design constraint hierarchies. In these hierarchies, the lowest-level constraints can represent physical limits grouped by common properties into named constraint blocks, which can be independently enabled or disabled.

Note that there is also “static” constraints. The format is:

```
static constraint <constraint_identifier> <constraint_block>
```

If a constraint block is declared as static, then calls to `constraint_mode( )` will affect all instances of the specified constraint in all objects. Thus, if a static constraint is set to *off*, it is off for all instances of that particular class.

Here is a recap to some of the important properties of constraints:

- Constraint blocks are not executed from top to bottom like procedural code but are all active at the same time.
- Constraints can be any SystemVerilog expression with variables and constants of integral type (e.g., bit, reg, logic, integer, enum, and packed struct).
- The constraint solver will be able to handle a wide spectrum of equations, such as algebraic factoring, complex Boolean expressions, and mixed integer and bit expressions. We will see more examples on this.
- You need to be careful in planning your constraints. You cannot contradict them among each other; else you will get a run-time error. If a solution exists, the constraint solver will find it. The solver can fail only when the problem is over-constrained and there is no combination of random values that satisfy the constraints.

- Constraints support only 2-state values. The 4-state values (X or Z) or 4-state operators (e.g., ===, !==) are illegal and will result in an error.
- If randomize( ) fails, the constraints are infeasible, and the random variables retain their previous values.
- The randomize( ) method is built-in and cannot be overridden.

## 13.5 Random Variables (rand and randc)

The class variables which get random values on randomization are called random variables. In order to make variables as random variables, class variables need to be declared using the *rand* and *randc* type-modifier keywords.

You can declare the following as “rand” or “randc”:

- Singular variables and any integral types
- Arrays and queues
- Arrays size
- Object handles

Variable declared with “rand” keyword are standard random variables. Their values are *uniformly* distributed over their range. For example:

```
rand bit [3:0] length;
```

This is a 4-bit unsigned variable with a range of 0 to 15. If unconstrained, this variable will be assigned any value in the range of 0 to 15 with *equal probability*. So, in this example, the probability of the same value repeating on successive calls to randomize is 1/16.

In contrast, “randc” are random-cyclic variables that cycle through all the value in a random permutation of their declared range. For example:

```
randc bit [1:0] length;
```

The variable “length” can take on values 0, 1, 2, and 3. Randomize computes an initial random permutation of the range values of “length” and then returns those values in the order of successive calls. After it returns the last element of a permutation, it repeats the process by computing a new random permutation. Here is how these permutations will work:

```
0 -> 3 -> 2 -> 1 (initial)
2 -> 1 -> 3 -> 0 (next permutation)
2 -> 0 -> 1 -> 3 (next permutation)
```

Note that the semantics of random-cyclic (randc) variables will be solved *before* other random variables. A set of constraints that include both “rand” and “randc” variables will be solved so that the “randc” variables are solved first.

Here is an example distinguishing “rand” from “randc”:

```
class aClass;
    rand bit [1:0] addr;
    randc bit [1:0] rc;
endclass

module tb;
    initial begin
        aClass ac = new ( );
        for (int i = 0; i < 12; i++) begin
            ac.randomize( );
            $display ("i=%0d addr=%b rc=%b", i, ac.addr, ac.rc);
        end
    end
endmodule
```

*Simulation log:*

```
i=0 addr=11 rc=01
i=1 addr=10 rc=00
i=2 addr=01 rc=10
i=3 addr=01 rc=11
i=4 addr=01 rc=10
i=5 addr=11 rc=01
i=6 addr=00 rc=00
i=7 addr=10 rc=11
i=8 addr=01 rc=00
i=9 addr=10 rc=11
i=10 addr=11 rc=10
i=11 addr=11 rc=01
```

V C S   S i m u l a t i o n   R e p o r t

Both “addr” and “rc” are two bit variables. “addr” is rand, while “rc” is randc. We simply randomize their values in module “tb.” Since these are 2-bit variables, there are four combination sequences that will be iterated by the “randc” variable “rc.” The values within a sequence are not repeated as you see in the simulation log:

```
01 -> 00 -> 10 -> 11 // non-repeated numbers – cyclic permutation
10 -> 01 -> 00 -> 11 // non-repeated numbers – cyclic permutation
00 -> 11 -> 10 -> 01 // non-repeated numbers – cyclic permutation
```

In contrast, if you look at the random values generated by “rand addr,” they are not cyclic and repeat within the sequence:

```
11 -> 10 -> 01 -> 01 //repeated number – noncyclic permutation
01 -> 11 -> 00 -> 10
01 -> 10 -> 11 -> 11 //repeated number – noncyclic permutation
```

### 13.5.1 Static Random Variables

Random variables declared as static are shared by all instances of the class in which they are declared. Each time the randomize( ) method is called, the static variable is changed in every class instance. Here is an example:

```
class StaticRandom;
    //static random variable - shared by all instances
    rand static integer Var1;

    rand int Var2;
endclass

module TOP;
    StaticRandom s1 = new;
    StaticRandom s2 = new;

initial begin
    repeat(2)
        begin
            void'(s1.randomize());
            $display("s1.Var1 : %d s1.Var2 : %d : s2.Var1 : %d s2.Var2 : %0d :",
                    s1.Var1,s1.Var2,s2.Var1,s2.Var2);
        end
    end
endmodule
```

*Simulation log:*

```
s1.Var1 : 409176739 s1.Var2 : 1499859087 : s2.Var1 : 409176739 s2.Var2 : 0 :
s1.Var1 : -1430829098 s1.Var2 : 100214679 : s2.Var1 : -1430829098 s2.Var2 : 0 :
V C S   S i m u l a t i o n   R e p o r t
```

In this example, we declare Var1 as a static random variable, while leave Var2 as a normal random variable. In module TOP, we randomize only the “s1” instance of “StaticRandom” class (s1.randomize( )). We do – not – randomize the “s2” instance. Now, when we display the values of Var1 and Var2 of both instances, we notice that the value of s2.Var1 randomizes even though we did not randomize “s2.” That is because Var1 is static random.

In the simulation log, you see that s1.Var1 is randomized and so is s2.Var1 – with the same value as s1.Var1 – since Var1 is static:

```
s1.Var1 : 409176739 == s2.Var1 : 409176739,
s1.Var1 : -1430829098 == s2.Var1 : -1430829098.
```

But s2.Var2 (which is non-static) does not get randomized since, again, we are calling only s1.randomize(). Its value stays at default 0.

### 13.5.2 Randomizing Arrays and Queues

You can randomize dynamic arrays, associative arrays, and queues. They can be declared “rand” or “randc” which will cause all the array’s elements to be randomized. All the elements in the array are randomized, overwriting any previous data. You can also constrain individual array elements. Here is an example:

```

class aClass;
    rand bit [1:0] addr [ ] = {1,2,3,4}; //dynamic array

    //associative array
    rand bit [7:0] St [string] = {'{"Peter":26, "Paul":24};

    rand bit [7:0] dq2[$] = { 3, 2, 7, 1 }; //queue
    rand bit [2:0] len;
    rand bit [7:0] data [ ]; //dynamic array

    //constraint size of dynamic array 'data'
    constraint length { data.size == len; }

    //constraint addr[0] of dynamic array 'addr'
    constraint addrC {addr[0] == 0; }

    //constraint dq2[0] of queue 'dq2'
    constraint dq2C {dq2[0] == 'hff; }

endclass

module tb;
    initial begin
        aClass aC = new ( );

        for (int i = 0; i < 4; i++) begin
            aC.randomize();
            $display ("addr=%p St=%p dq2=%p", aC.addr, aC.St, aC.dq2);
        end

        $display("\n");
        for (int i = 0; i < 4; i++) begin
            aC.randomize( );
            $display ("len=%0d data=%p", aC.len, aC.data);
        end
    end
endmodule

```

*Simulation log:*

```

addr={'h0, 'h3, 'h3, 'h1} St={"Paul":'h8a, "Peter":'hc4} dq2=['hff, 'h73, 'h42, 'h49}
addr={'h0, 'h2, 'h3, 'h2} St={"Paul":'h91, "Peter":'h83} dq2=['hff, 'h3d, 'hdb, 'hfe}
addr={'h0, 'h1, 'h2, 'h3} St={"Paul":'hec, "Peter":'h99} dq2=['hff, 'hd2, 'hac, 'hb4}
addr={'h0, 'h1, 'h3, 'h1} St={"Paul":'h93, "Peter":'h37} dq2=['hff, 'h6a, 'h61, 'h84}

len=1 data=['hb]
len=2 data=['h7b, 'h95]
len=1 data=['h0]
len=6 data=['hda, 'hea, 'hf6, 'he7, 'hd1, 'h9a]

V C S   S i m u l a t i o n   R e p o r t

```

In class “aClass,” we declare two dynamic arrays, “addr” and “data,” one associative array of type string (“St”) and a queue called “dq2.” All are declared as “rand.” We then constrain individual element of dynamic array “addr” and also individual element of queue “dq2”:

```

//constraint addr[0] of dynamic array 'addr'
constraint addrC {addr[0] == 0; }

//constraint dq2[0] of queue 'dq2'
constraint dq2C {dq2[0] == 'hff; }

```

In addition, we declare a rand variable “len” and use its random value to constrain the size of the dynamic array “data”:

```

rand bit [2:0] len;
rand bit [7:0] data [];
//constraint size of dynamic array 'data'
constraint length { data.size == len; }

```

These constraints will hold when these arrays and queues are randomized. Here is how to read the simulation log.

The randomized values of “addr,” “St,” and “dq2” are as shown in the first part of the simulation log. Note that the addr[0] was constrained to “0” and dq2[0] was constrained to ‘hff’. Hence, you see that these values are not randomized. Other individual elements are randomized as expected. Note how the associative array elements are randomized.

In the second part of the simulation log, we see randomized value of “len.” This “len” value will size the dynamic array “data.” As shown in the simulation log, the “len” keeps varying and so does the size of array “data.” Hence, you see different sized “data” array with its individual elements randomized.

When a queue is resized by randomize( ), elements are inserted or deleted at the back (i.e., right side) of the queue as necessary to produce the new queue size; any new elements inserted take on the default value of the element type. That is, the resize grows or shrinks the array.

### 13.5.3 Randomizing Object Handles

You can declare an object handle as “rand.” In that case, all of that object’s variables and constraints are randomized. Randomization will not modify the actual object handle. Also, object handles cannot be declared “randc.” Here is an example:

```

class child;
    rand bit [7:0] data;
endclass

class parent extends child;
    rand bit [7:0] addr;
    rand child ch = new ( ); //object handle is 'rand'
endclass

module tb;
    int i;
    parent pp = new ( );

    initial begin
        for (i=0; i < 4; i++) begin
            pp.randomize( );
            $display("Parent addr = %h Child data = %h", pp.addr,
pp.ch.data);
        end
    end
endmodule

```

*Simulation log:*

Parent addr = 8f Child data = 11

Parent addr = 97 Child data = 16

Parent addr = de Child data = f3

Parent addr = 4f Child data = b5

V C S   S i m u l a t i o n   R e p o r t

The class “child” has a rand variable “addr.” Class “parent” extends class “child” and adds a rand variable called “data.” Now, we instantiate the class “child” in class “parent” with object handle named “ch.” We then randomize the object handle “ch” as shown in the code. Next, when we instantiate class “parent” in module “tb” and randomize it ( pp.randomize( ) ), it will not only randomize its properties but also the properties of class “child.” Hence, you see in the simulation log that both “addr” (of class “parent”) and “data” (of class “child”) are randomized. Obviously, if “ch” is not declared rand, its properties will not be randomized.

## 13.6 Constraint Blocks

As we saw in the preceding sections, we use a constraint block to constrain the values of random variables. For example, the following is a constraint block from one of the examples above:

```
constraint addrC {addr[0] == 0;}
```

In this section, we will go into plenty of details on different features available in constraint blocks. We will discuss the following:

- External constraint blocks
- Constraint inheritance (we've already seen this feature in preceding examples)
- “inside” operator (Sect. 12.14).
- Weighted distribution
- Unique constraints
- Implication
- If-else construct
- Iterative constraints (*foreach*)
- Functions in constraints
- Soft constraints

### 13.6.1 External Constraint Blocks

You can specify a constraint as an “extern” constraint in a class. Once you declare a constraint as “extern” you must provide its definition in an external constraint block, else you will get an error. Note that both an “external” constraint and an internal (implicit) constraint (that we have seen so far) can be defined by a constraint block *external* to the class. For either form, it is an error if more than one external constraint block is provided for any given prototype, and it is an error if a constraint block of the same name as a prototype appears in the same class declaration. Here is an example:

```
class externC;
  randc bit [7:0] bx;

  constraint cxG;           //internal constraint
  extern constraint eXS;   //external constraint
endclass

constraint externC::cxG { bx > 10; }
constraint externC::eXS { bx < 20; }
```

```

module tb;
    int i;

    externC pp = new ( );

    initial begin
        for (i=0; i < 4; i++) begin
            pp.randomize( );
            $display("bx = %d", pp.bx );
        end
    end
endmodule

```

*Simulation log:*

```

bx = 14
bx = 17
bx = 18
bx = 15

```

V C S   S i m u l a t i o n   R e p o r t

In this example, there are two constraints specified in class “externC,” but neither of them has a constraint\_block associated with them. One is internal constraint, while the other is “extern” constraint:

```

constraint cXG;           //internal constraint
extern constraint eXS;   //external constraint

```

We then provide the constraint definition of these two constraints outside of the class “externC”:

```

constraint externC::cXG { bx > 10; }
constraint externC::eXS { bx < 20; }

```

As noted above both the internal and external constraints can have their constraint blocks defined outside of the class, except that for “extern” constraint, you must specify its definition outside the class. Else you will get an error as shown (Synopsys – VCS):

```

Error-[SV_MEECD] Missing explicit external constraint def
testbench.sv, 6
$unit, "constraint eXS";

```

The explicit external constraint 'eXS' declared in the class 'externC' is not defined.

Provide a definition of the constraint body 'eXS' or remove the explicit

external constraint declaration 'eXS' from the class declaration 'externC'.

For internal constraint, if you do not specify an external definition, it will be considered an empty constraint.

The simulation log shows that both constraints on the variable “bx” are met.

### **13.6.2 Weighted Distribution**

As we know, constraints provide control on randomization from which user can control the values of randomization. There are many ways to control these values. One of them is weighted distribution. Weighted distribution creates distributions in a constraint block such that some values are chosen more often than others. Weighted distribution operator is *dist*, and it takes a list of values and weights separated by either “:=” or “:/” operators. The “:=” operator specifies that the weight is the same for every specified value in a range and the “:/” operator specifies that the weight is to be equally divided among all the values. The syntax for these two operators is:

Value := <weight>  
Value :/ <weight>

The values and weights can be constants or variables. The “value” can be single or a range. The default weight of an unspecified value is 1. Also, it is not necessary to have the sum of weights to be 100.

For example:

Data dist { 10 := 8, [11:13] := 10 };

In this case the weighted distribution is as follows. Equal distribution for a range of values:

Data == 10, weight 8  
Data == 11, weight 10  
Data == 12, weight 10  
Data == 13, weight 10  
And

Data dist { 10 :/ 8, [11:13] :/ 10 };

In this case the weighted distribution is as follows. The :/ assigns the specified weight to the item, or if the item is a range, it specifies weight/N where N is the number of values in the range:

Data == 10, weight 8  
Data == 11, weight 10/3  
Data == 12, weight 10/3  
Data == 13, weight 10/3

Let us now look at a working example:

```

class distClass;
  rand bit [2:0] data;
  constraint distr { data dist { 0:=30, [1:3]:=70, 4:=50,
5:=20}; }
endclass

module top;

distClass dc = new ();

initial begin
  for (int i = 0; i < 20; i++) begin
    dc.randomize();
    $display ("data=%0d",dc.data);
  end
end
endmodule

```

*Simulation log:*

```

data=5
data=1
data=1
data=3
data=3
data=2
data=1
data=4
data=1
data=3
data=3
data=4
data=4
data=2
data=3
data=0
data=2
data=2
data=3
data=2
data=1
data=4
data=0

```

V C S   S i m u l a t i o n   R e p o r t

The weights are distributed as follows:

```
data == 0, weight 30
data == 1,2,3 weight 70
data == 4, weight 50
data == 5, weight 20
```

The distribution in this example shows that data range [1:3] has the highest distribution weight of 70. Hence in the simulation log, you see more values of 1, 2, and 3 as the values generated and a few “4” since its weight is lower at “50.” But you see only two “0” since its weight is the second lowest 30 and only one “5” since its weight is the lowest.

Now, let us look at an example where we contrast “`:=`” with “`:/`” operators. We extend the above example with “`:/`” operator to see side-by-side comparison:

```
class distClass;
  rand bit [2:0] data, addr;
  constraint distr { data dist { 0:=30, [1:3]:=70, 4:=50,
5:=20}; } //      := operator
  constraint distr1 { addr dist { 0:/30, [1:3]:/70, 4:/50,
5:/20}; } //      :/ operator
endclass

module top;

  distClass dc = new ( );

  initial begin
    for (int i = 0; i < 20; i++) begin
      dc.randomize();
      $display ("data=%0d addr=%0d",dc.data, dc.addr);
    end
  end
endmodule
```

*Simulation log:*

```
data=5 addr=1
data=1 addr=3
data=1 addr=4
data=3 addr=5
data=2 addr=2
data=1 addr=1
data=4 addr=4
data=1 addr=4
data=3 addr=1
```

```

data=4 addr=4
data=4 addr=0
data=2 addr=5
data=3 addr=4
data=0 addr=0
data=2 addr=0
data=3 addr=2
data=2 addr=1
data=1 addr=5
data=4 addr=0
data=0 addr=0

```

### VCS Simulation Report

The distribution of “data” is identical to what we saw in the previous example but contrast that with the distribution of “addr.” The “addr” weight is distributed as follows:

```

addr == 0, weight 30
addr == 1,2,3 weight 70/3 ( = 23.3 )
addr == 4, weight 50
addr == 5, weight 20

```

So, the weight of `addr == 1, 2, 3` has now reduced from 70 to 23.3. Hence, the weight of other values is now higher. Hence, you see more values of “4” followed by values of “0,” followed by lesser values of 1, 2, and 3 and lastly the least number of values of 5. Contrast the distribution of “data” with “addr,” and you will see the difference between the two operators.

A couple of final points. A “dist” operation cannot be applied to “randc” variables. A “dist” operation requires that the expression contains at least one “rand” variable.

### 13.6.3 “unique” Constraint

When you do not want any two members of a group have the same value after randomization, you need to use the “unique” constraint. “unique” is a keyword. “unique” will generate unique values of a constrained rand variable. It also applies to arrays. It will generate unique elements in an array, may it be dynamic, associative, or queue. Here is an example. It shows the difference between randomization with “uniqueness” and one without it:

```

class distClass;
  rand bit [1:0] data1, data2, data3, data4;
  rand bit [1:0] addr1, addr2, addr3, addr4;
  constraint distr { unique {data1, data2, data3, data4}; }
                    // 'unique' random

```

```

constraint distr1 { {addr1, addr2, addr3, addr4}; }
                  //non-unique random
endclass

module top;

distClass dc = new ();

initial begin
    for (int i = 0; i < 5; i++) begin
        dc.randomize();
        $display ("data1=%0d data2=%0d data3=%0d data4=%0d",dc.
data1, dc.data2, dc.data3, dc.data4);
        $display ("addr1=%0d addr2=%0d addr3=%0d addr4=%0d",dc.
addr1, dc.addr2, dc.addr3, dc.addr4);
        $display("\n");
    end
end
endmodule

```

*Simulation log:*

```

data1=3 data2=1 data3=0 data4=2
addr1=3 addr2=2 addr3=3 addr4=1

data1=2 data2=1 data3=0 data4=3
addr1=1 addr2=1 addr3=0 addr4=1

data1=1 data2=0 data3=3 data4=2
addr1=2 addr2=2 addr3=0 addr4=1

data1=0 data2=1 data3=2 data4=3
addr1=3 addr2=1 addr3=3 addr4=2

data1=2 data2=1 data3=3 data4=0
addr1=3 addr2=1 addr3=3 addr4=2

```

## V C S S i m u l a t i o n R e p o r t

In this example, we have two types of randomization, one with “unique” keyword for unique randomization and one without:

```

constraint distr { unique {data1, data2, data3, data4}; }
//'unique' random
constraint distr1 { {addr1, addr2, addr3, addr4}; }
//non-unique random

```

data1 through data4 will be randomized but will not repeat a value within a set. But addr1 through addr4 uses the regular randomization as we have seen so far.

In the simulation log, you will notice that with each randomization, the data1 through data4 does not have any repeat values, while addr1 through addr4 show repeat values. This shows that unique can indeed be powerful when generating non-repeat random values.

Here is an example of how to generate unique values of dynamic, associative, and queue arrays:

```

class aClass;
    rand bit [1:0] addr [ ] = {1,2,3,4}; //dynamic array
    rand bit [7:0] St [string] = {'{"Peter":26, "Paul":24};
                                    //associative array
    rand bit [7:0] dq2[$] = { 3, 2, 7, 1 }; //queue

    constraint addrC {unique {addr};}
    constraint strC {unique {St};}
    constraint dq2C {unique {dq2};}

endclass

module tb;
    initial begin
        aClass aC = new ( );
        for (int i = 0; i < 4; i++) begin
            aC.randomize();
            $display ("addr=%p St=%p dq2=%p", aC.addr, aC.St, aC.dq2);
        end
    end
endmodule

```

#### *Simulation log:*

```

addr={'h3,'h1,'h0,'h2} St='{"Paul":'h97, "Peter":'heb} dq2='{'hc3,'hbc,'h5a,'h5f}
addr={'h2,'h1,'h0,'h3} St='{"Paul":'h37, "Peter":'h7} dq2='{'h80,'ha5,'h49,'hec}
addr={'h1,'h0,'h3,'h2} St='{"Paul":'h91, "Peter":'h44} dq2='{'h10,'h14,'he3,'hff}
addr={'h0,'h1,'h2,'h3} St='{"Paul":'h2f, "Peter":'hbb} dq2='{'h41,'ha6,'h56,'h49}

```

#### V C S   S i m u l a t i o n   R e p o r t

In the example, the dynamic array addr, associative array “St,” and queue “dq2” are randomized with “unique” operator. As you notice from the simulation log, none of the array elements are repeated in each individual randomization set.

### 13.6.4 Implication and If-Else

You can further constrain randomization using the implication operator (`->`) and if-else construct. It is a conditional relationship between two variables. The implication operator is used to declare an expression that implies a constraint. For example:

```
constraint dataC { (data == 'hff) -> (addr == 0); }
```

which is equivalent to:

```
if (data == 'hff) addr = 0;
```

Here is an example showing both the implication operator and the if-else ways of constraining values:

```
class dataP;
    rand bit [7:0] data;
    string    burst_mode ;
    constraint burst1 { (burst_mode == "short") -> (data < 16); }
                           //implication
    constraint burstIf { if (burst_mode == "long") data > 16;
                         else if (burst_mode == "garbage") data == 0; }
endclass

module top;
    initial begin
        dataP d1;
        d1 = new();

        d1.burst_mode = "short";
        repeat(4) begin
            d1.randomize();
            $display("\t Short burst_mode = %s data = %0d",d1.burst_
mode,d1.data);
        end

        $display("\n");
        d1.burst_mode = "long";
        repeat(4) begin
            d1.randomize();
            $display("\t Long burst_mode = %s data = %0d",d1.burst_-
mode,d1.data);
        end
    end

```

```

$display("\n");
d1.burst_mode = "garbage";
repeat(4) begin
  d1.randomize();
  $display("\t garbage burst_mode = %s data = %0d",d1.burst_
mode,d1.data);
end
end
endmodule

```

*Simulation log:*

Short burst\_mode = short data = 13

Short burst\_mode = short data = 6

Short burst\_mode = short data = 1

Short burst\_mode = short data = 2

Long burst\_mode = long data = 36

Long burst\_mode = long data = 109

Long burst\_mode = long data = 18

Long burst\_mode = long data = 47

garbage burst\_mode = garbage data = 0

#### V C S   S i m u l a t i o n   R e p o r t

In this example, we use the implication ( $\rightarrow$ ) operator to imply constraint on “data”:

```
constraint burst1 { (burst_mode == "short") -> (data < 16); }
```

Whenever `burst_mode == "short"`, data should be less than 16. In the module “top,” we assign the string `burst_mode` that value “short,” and as expected we get data values less than 16 as shown in the first part of simulation log. Similarly, with the if-else condition, if `burst_mode == long`, we see the data values to be greater than 16 as shown in the simulation log; else if `burst_mode == garbage`, we see that data values are constrained to zero.

### 13.6.5 Iterative Constraint (`foreach`)

SystemVerilog provides support for “`foreach`” loop inside a constraint so that arrays can be constrained. The `foreach` construct specifies iteration over the elements of an array. Its argument is an identifier that designates any type of array (fixed-size,

dynamic, associative, or queue) followed by a list of loop variables enclosed in square brackets. Each loop variable corresponds to one of the dimensions of the array.

Here is the syntax:

```
constraint    constraint_name    {foreach    (variable[iterator])
variable[iterator] < constraint condition >}
```

Here is an example:

```
class bus;
  rand byte addr [4];
  rand byte data [4];

constraint avalues {foreach( addr[i] ) addr[i] inside {1,2,3,4}; }
constraint dvalues {foreach( data[j] ) data[j] == 16 * j; }

endclass

module constr_iteration;
  initial begin
    bus b1;
    b1 = new();

    b1.randomize();

    foreach(b1.addr[i]) $display("\t addr[%0d] = %0d", i,
b1.addr[i]);
    foreach(b1.data[i]) $display("\t data[%0d] = %0d", i, b1.
data[i]);

  end
endmodule
```

#### *Simulation log:*

```
addr[0] = 4
addr[1] = 1
addr[2] = 3
addr[3] = 2
data[0] = 0
data[1] = 16
data[2] = 32
```

```
data[3] = 48
```

### VCS Simulation Report

In this example, we declare two arrays, namely, `addr[4]` and `data[4]`. In the constraint block, we constrain each element of these arrays with a “foreach” loop. The loop will go through each of the four elements of the arrays and constrain them as specified.

We see the effect of this randomization when we loop through each element of the array in module “top.” The simulation log shows the correctly constrained values for each element of the array.

Here’s another example, showing how the size of arrays are determined and used by “foreach” loop for constraining the arrays:

```
class arrayClass;
    rand bit[7:0] dyn_array [ ];
    rand bit[7:0] queue [$];

    // Assign size of the queue
    constraint qsize { queue.size( ) == 3; }

    // Constrain each element of both the arrays
    constraint arrays { foreach (dyn_array[i])
        dyn_array[i] == i*10;
        foreach (queue[i])
            queue[i] == i*20;
    }

    function new ();
        dyn_array = new[4];           // Assign size of dynamic array
    endfunction
endclass

module tb;

arrayClass ac = new;

initial begin
    ac.randomize();
    $display ("dyn_array = %p \nqueue = %p", ac.dyn_array, ac.queue);
end
endmodule
```

*Simulation log:*

```
# run -all
```

```
# dyn_array = '{0, 10, 20, 30}
# queue = '{0, 20, 40}
# exit
```

In this example, we define a dynamic array (“dyn\_array”) and a queue (“queue”). The size of each is unknown at their time of declaration. But we need to know their sizes to constrain them using a “foreach” loop. For the “queue” we constrain `queue.size( ) == 3;`, and for the “dyn\_array,” we assign it a size when we instantiate it in the constructor of the class “arrayClass.” Then we constrain both in the constraint “arrays” using “foreach” loops. Simulation log shows the constrained values of the dynamic array and the queue.

### **13.6.6 Array Reduction Methods for Constraint**

Array reduction methods (discussed in Sect. 3.5.3) can produce a single value from an unpacked array of integral values. This can be used within a constraint to allow the expression to be considered during randomization.

The reduction methods apply specified operation between each element of the unpacked array. For example:

```
rand int a[10];
a.sum == a[0] + a[1] + ...
a.product == a[0] * a[1] * ...
a.and == a[0] & a[1] & ...
a.or == a[0] | a[1] | ...
a.xor == a[0] ^ a[1] ^ ...
```

Here’s an example showing the use of array reduction method “sum” in a constraint:

```
class a_reduction;
  rand bit[7:0] Darray [7];

  constraint arraySum { Darray.sum( ) with (int'(item)) == 30; }
endclass

module tb;

  a_reduction ar = new;

  initial begin
```

```

ar.randomize( );
$display ("Darray = %p", ar.Darray);
end
endmodule

```

*Simulation log:*

```

# run -all
# Darray ='{4, 2, 6, 4, 0, 14, 0}
# exit

```

In this example, we use the array reduction method “sum” to constrain “Darray”:

```
constraint arraySum { Darray.sum( ) with (int'(item)) == 30; }
```

This is equivalent to:

```
int'(Darray[0]) + int'(Darray[1]) + ... + int'(Darray[6]) == 30
```

Simulation log shows that the constraint assigns values to Darray elements such that the sum of all its elements is = 30.

### 13.6.7 Functions in Constraints

There are times when you need to specify constraints based on complex constraint expressions. So far, we have seen simple constraints that can be specified with a single expression. But this expression would become overly complicated, if not impossible, without the availability of a “function” to model the complex condition.

Here is an example (SystemVerilog-LRM):

```

function int count_ones ( bit [9:0] w );
    for( count_ones = 0; w != 0; w = w >> 1 )
        count_ones += w & 1'bl;
endfunction

class func;
    rand bit [9:0] length, v;
    //use function 'count_ones'
    constraint C1 { length == count_ones(v); }
endclass

module top;
    initial begin
        func fc = new();

```

```

for (int i=0; i<4; i++) begin
    fc.randomize( );
    $display("v = %b; length=%0d",fc.v,fc.length);
end
end
endmodule

```

*Simulation log:*

```

ncsim> run
v = 1101110001; length=6
v = 1101111011; length=8
v = 0111100000; length=4
v = 1001011011; length=6
ncsim: *W,RNQUIE: Simulation is complete.

```

In this example, we use a function(`count_ones`) to model a loop that counts the number of the ones in a given input value. We then use this function to specify a constraint in class “func”:

```
constraint C1 { length == count_ones(v); }
```

If we were to model this constraint without the use of a function, it will look like:

```

constraint C1
{
    length == ((v>>9)&1) + ((v>>8)&1) + ((v>>7)&1) + ((v>>6)&1) +
    ((v>>5)&1) +
    ((v>>4)&1) + ((v>>3)&1) + ((v>>2)&1) + ((v>>1)&1) + ((v>>0)&1);
}

```

As you can see, the constraint expression becomes complex to model. There can be even more complex situations which would be impossible to model without the facility of using a function.

We then randomize the class “func” and see that the constraint with `count_ones` works and provides us results for randomized values of “v.” The “length” is the number of 1’s in “v.”

### 13.6.8 Soft Constraints

The constraints we have seen so far are considered *hard* constraints, meaning if there is a contradiction among the constraints, the solver will fail and so will your simulation. In contrast, when there is no solution that satisfies all active hard constraints simultaneously with a constraint defined as *soft*, the solver will discard that soft constraint and find a solution that satisfies the remaining constraints. Soft

constraints only express a preference for one solution over another; they are discarded when they are contradicted by other more important constraints. Regular (hard) constraints must always be satisfied; they are never discarded and are thus considered to be of the same highest priority.

Soft constraints are declared using the keyword *soft*.

One of the places constraints end up having conflict is when a constraint is defined in a class and then overridden with an inline constraint in a module, for example. The author may want to override the class constraint with the one in the inline constraint, but with hard constraints, it is not possible. You can declare the class constraint as soft constraint and then override it with hard inline constraint.

Let us see some examples. First an example with conflicting constraints, that will result in run-time error:

```
class sft;
    rand bit [3:0] data;
    constraint data_range { data > 'hf; }
    constraint data_rangel { data <= 'ha; } //conflicting constraint
endclass

module soft_constr;
    initial begin
        sft sf;
        sf = new();

        repeat(4) begin
            sf.randomize();
            $display("\data = %0h",sf.data);
        end
    end
endmodule
```

As you notice in the example, the two constraints are conflicting. One is asking for data to be  $> 'hf$ , while the other is restricting data to be  $\leq 'ha$ . In this case, you will get the following run-time error (Synopsys – VCS):

---

Solver failed when solving the following set of constraints:

```
rand bit[3:0] data; // rand_mode = ON
constraint data_range // (from this) (constraint_mode = ON) (testbench.sv:5)
{
    (data > 4'hf);
}
```

Error-[CNST-CIF] Constraints inconsistency failure

testbench.sv, 16

Constraints are inconsistent and cannot be solved.

Please check the inconsistent constraints being printed above and rewrite them.

---

Now, let us look at the same example but with the following constraint changed to soft (no other change is made to the previous example):

```
constraint data_range { soft data > 'hf; }
```

Note the keyword “soft” added in the constraint. This will allow the constraint solver to override this constraint and use other non-conflicting constraints to constrain the randomized values. So, here is the same example but with the “soft” constraint:

```
class sft;
    rand bit [3:0] data;

    // 'soft' constraint
    constraint data_range { soft data > 'hf; }

    constraint data_range1 { data <= 'ha; }
endclass

module soft_constr;
    initial begin
        sft sf;
        sf = new();

        repeat(4) begin
            sf.randomize();
            $display("\data = %0h", sf.data);
        end
    end
endmodule
```

With “constraint data\_range { **soft** data > 'hf; }” as soft constraint, the solver will ignore this constraint and use the hard constraint “constraint data\_range1 { data <= 'ha; }.” There will not be a run-time error, and you will get the following randomized values according to the hard constraint.

*Simulation log:*

data = 6

data = 3

data = 0

```
data = 1
V C S S i m u l a t i o n R e p o r t
```

The data values are <= 'ha as required by the hard constraint.

Let us now look at the same example but with a conflict between the class constraint and an inline constraint. This is where soft constraints are most useful. You declare a generic constraint as a soft constraint allowing inline constraints to override the class constraint:

```
class sft;
    rand bit [3:0] data;
    constraint data_range { data > 'hf; } //class constraint
endclass

module soft_constr;
    initial begin
        sft sf;
        sf = new();

        repeat(4) begin
            sf.randomize() with { data <= 'ha; }; //inline constraint
            $display("\data = %0h",sf.data);
        end
    end
endmodule
```

As you notice in the example, the class constraint and the inline constraint contradict, and since both are default hard constraints, the solver will fail giving the same error message as seen before. Here is the error message (Synopsys – VCS):

---

Solver failed when solving the following set of constraints:

```
rand bit[3:0] data; // rand_mode = ON
constraint data_range // (from this) (constraint_mode = ON) (testbench.sv:5)
{
    (data > 4'hf);
}
Error-[CNST-CIF] Constraints inconsistency failure
testbench.sv, 15
```

Constraints are inconsistent and cannot be solved.

Please check the inconsistent constraints being printed above and rewrite them.

---

Let us now convert the class constraint to a soft constraint, as shown in the example below (same as the previous one but with a soft constraint):

```

class sft;
  rand bit [3:0] data;

//class constraint converted to 'soft' constraint
constraint data_range { soft data > 'hf; }

endclass

module soft_constr;
  initial begin
    sft sf;
    sf = new( );

    repeat(4) begin

      //inline constraint - hard
      sf.randomize( ) with { data <= 'ha;};
      $display("\data = %0h",sf.data);
    end
  end
endmodule

```

With the class constraint converted to “soft,” the solver will be able to constrain with the hard constraint and will pass giving us the following simulation log:

```

data = 6
data = 3
data = 0
data = 1
V C S   S i m u l a t i o n   R e p o r t

```

Finally, you can also “disable soft” constraint. The keywords to disable are *disable soft*. This allows for all lower priority soft constraints that reference the given random variable to be discarded. Here is an example:

```
constraint data_range { disable soft data; }
```

## 13.7 Randomization Methods

There are three randomization methods available in constrained random verification:

- Randomize( )
  - Virtual function int randomize( );

- Pre-randomize
  - Function void pre\_randomize( );
- Post-randomize
  - Function void post\_randomize( );

We have used randomize( ) method throughout this section. To recap, every class has a built-in randomize( ) method that is a virtual function that generates random values for all the random variables (rand and randc) in the object subject to provided constraints.

The randomize( ) method returns a 1 if successful, otherwise returns a 0 as in the following example. I should mention that the return value of randomize() should always be tested. You can use assert(randomize()), but sometimes users globally turn off all assertions, and unfortunately the call to randomize does not get executed. In that case, use if (randomize()) else \$error(...);. Some tools automatically generate an error message when the solver fails, but it will not stop the simulation and you wind up with an un-modified object:

```
sft sf;
sf = new( );
int success = sf.randomize( );
if (success == 1) ...
```

### **13.7.1 Pre-randomization and Post-randomization**

On calling randomize( ), the pre\_randomize( ) and post\_randomize( ) get called before and after randomize( ) call, respectively. But user can override these two methods. The pre\_randomize function can be used to set pre-conditions before the object randomization. The post\_randomization function can be used to check and perform post-conditions after the object randomization. When randomize( ) is invoked, it first invokes pre\_randomize( ) on obj and also all of its random object members that are enabled. After the new random values are computed and assigned, randomize( ) invokes post\_randomize( ) on obj and also all of its random object members that are enabled.

User can override these built-in functions to set any pre-conditions before the randomize( ) call. If the class is an extended class and no user-defined implementation of pre\_randomize( ) exists, then pre\_randomize() will automatically call super. pre\_randomize( ). Same applies to post\_randomize( ) method.

Here is an example, where we override the pre- and post-randomization methods with user-defined ones:

```
class sft;
  rand bit [3:0] data, addr;
  constraint data_range1 { data <= 'ha; }
```

```

function void pre_randomize ( );
    $display ("\n User-defined pre_randomize called before random-
ize( )");
    $display("data = %0h",data);
    $display("addr = %0h",addr);

    addr.rand_mode(0); //disable addr randomization
endfunction

function void post_randomize ( );
    $display ("\n User-defined post_randomize called after random-
ize( )");
    $display("data = %0h",data);
    $display("addr = %0h",addr);
endfunction

endclass

module PrePost_constr;
initial begin
    sft sf;
    sf = new();

    repeat (2) sf.randomize( );
end
endmodule

```

*Simulation log:*

User-defined pre\_randomize called before randomize()

data = 0

addr = 0

User-defined post\_randomize called after randomize()

data = 3

addr = 0

User-defined pre\_randomize called before randomize()

data = 3

addr = 0

User-defined post\_randomize called after randomize()

data = 9

addr = 0

V C S   S i m u l a t i o n   R e p o r t

In this example, we override the built-in functions/methods pre\_randomize() and post\_randomize(). In both functions we use a \$display to show that we are

indeed in pre- or post-randomize( ) state. In the pre\_randomize( ) function, we also disable randomization of “addr” variable (addr.rand\_mode(0));. This is to show that you can have any such pre-condition prior to invocation of randomize( ) method. As you notice from the simulation log, the user-defined \$display show up pre- and post-randomization. Also, since “addr” randomization is disabled in pre\_randomize( ) function, we do not see any randomized value on “addr” as shown in the simulation log (you only see the initial value 0 on “addr”). That proves that conditions set in pre\_randomize do get exercised before randomize( ) is invoked.

Note that randomize( ) is built-in and cannot be overridden.

Let us see an example of how pre\_randomize( ) and post\_randomize( ) work with extended classes. In what order do they get called?

```

class parent;

function void pre_randomize;
    $display(" PARENT PRE_RANDOMIZATION ");
endfunction

function void post_randomize;
    $display(" PARENT POST_RANDOMIZATION ");
endfunction

endclass

class Ex1 extends parent;

function void pre_randomize;
    $display(" Ex1 PRE_RANDOMIZATION ");
endfunction

function void post_randomize;
    $display(" Ex1 POST_RANDOMIZATION ");
endfunction

endclass

class Ex2 extends parent;

function void pre_randomize;
    super.pre_randomize( ); //Call to super.

```

```

$display(" Ex2 PRE_RANDOMIZATION ");
endfunction

function void post_randomize;
super.post_randomize( ); //call to super.
$display(" Ex2 POST_RANDOMIZATION ");
endfunction

endclass

module TOP;

parent P = new( );
Ex1 E1 = new( );
Ex2 E2 = new( );

initial
begin
void'(P.randomize( ));
void'(E1.randomize( ));
void'(E2.randomize( ));
end
endmodule

```

*Simulation log:*

```

PARENT PRE_RANDOMIZATION
PARENT POST_RANDOMIZATION
Ex1 PRE_RANDOMIZATION
Ex1 POST_RANDOMIZATION
PARENT PRE_RANDOMIZATION
Ex2 PRE_RANDOMIZATION
PARENT POST_RANDOMIZATION
Ex2 POST_RANDOMIZATION
V C S   S i m u l a t i o n   R e p o r t

```

Three classes are defined: “parent” class and “Ex1” and “Ex2” which are extended from “parent.” Each class has a pre\_randomize( ) and post\_randomize( ) function in it. Note that “Ex2” calls super.pre\_randomize( ) first in its pre\_randomize( ) function. Similarly, it also calls post\_randomize first in its post\_randomize function.

When (in the TOP module) we call P.randomize( ), the pre- and post-randomize functions in the class “parent” are exercised as shown in the simulation log:

PARENT PRE\_RANDOMIZATION  
 PARENT POST\_RANDOMIZATION

When we call E1.randomize( ) (E1 is the instance of class Ex1), it will exercise its pre- and post-randomization functions, as shown in the simulation log:

Ex1 PRE\_RANDOMIZATION  
 Ex1 POST\_RANDOMIZATION

When we call E2.randomize( ) (E2 is the instance of class Ex2), we first call the super.pre\_randomize( ), which results in the following simulation log. First, the pre\_randomize( ) function of the “super,” i.e., the “parent” class, is exercised:

PARENT PRE\_RANDOMIZATION  
 Ex2 PRE\_RANDOMIZATION

Similarly, in the post\_randomize( ) function of E2, we first call super.post\_randomize( ) which will exercise the post\_randomize( ) function of the “parent” class. This results in the following simulation log:

PARENT POST\_RANDOMIZATION  
 Ex2 POST\_RANDOMIZATION

As mentioned before, if the class is a derived class and no user-defined implementation of pre\_randomize( ) and post\_randomize( ) exists in the derived class, then pre\_randomize( ) and post\_randomize( ) will automatically invoke super.pre\_randomize( ) and super.post\_randomize( ), respectively. Here's an example:

```
class parent;

function void pre_randomize;
  $display(" PARENT PRE_RANDOMIZATION ");
endfunction

function void post_randomize;
  $display(" PARENT POST_RANDOMIZATION ");
endfunction
endclass

class Ex1 extends parent;
endclass

module TOP;
  Ex1 E1 = new();
  initial
    if (E1.randomize()) $display ("PASS");
endmodule
```

*Simulation log:*

```
PARENT PRE_RANDOMIZATION
PARENT POST_RANDOMIZATION
PASS
```

**V C S   S i m u l a t i o n   R e p o r t**

Class “Ex1” extends class “parent” but does not define pre\_randomize( ) or post\_randomize( ) functions. Hence, when E1 (which is the instance of class “Ex1”) is randomized, it first calls the pre- and post-randomize functions of its “parent” class which in turn executes its pre- and post-randomize functions. This is shown in the simulation log.

Note that pre- and post-randomization functions allow you to model any functionality that a function allows. I have shown only \$display to show how they get called and exercised. But you can have a meaningful functionality that you want to model before randomization (pre) and after randomization (post).

### **13.7.2 Local Scope Resolution (*local::*)**

When “randomize( ) with” method is called on an object, it refers to both the class properties and variables local to the method call. If there is a variable declared both in a class and also in the scope where randomize( ) is called, one needs to distinguish between the two, as in the following example (SystemVerilog-LRM):

```
class C;
    rand integer x;
endclass

function int F(C obj, integer x);
    F = obj.randomize( ) with { x < local::x; };
endfunction
```

In this example, the first “x” binds to the variable “x” in the class C, while the local::x binds to the input of the function F.

## **13.8 rand\_mode( ): Disabling Random Variables**

As we saw in a previous example, rand\_mode( ) allows us to disable randomization on a random variable. When the randomization on a “rand” or “randc” variable is disabled, the variable acts as if it were not a “rand” or “randc” variable.

When <variable>.rand\_mode(0) is invoked, it will set the specified variables to an inactive state so that they are not randomized by subsequent randomize( ) calls.

When <variable>.rand\_mode(1) is invoked, it will set the specified variable to an active state so that they are randomized by subsequent randomize( ) calls.

Here is an example:

```
class rmode;
    rand bit [3:0] data, addr;
    constraint data_rangel { data <= 'ha; }

    function void disp();
        $display("data = %0h",data);
        $display("addr = %0h",addr);
    endfunction
endclass

module TOP;
    initial begin
        rmode sf;
        sf = new();

        $display("\n rand_mode(0) - Individual");
        sf.data.rand_mode(0); //Turn OFF data random mode
        sf.addr.rand_mode(0); //Turn OFF addr random mode

        repeat (2) begin sf.randomize();
            sf.disp;
        end

        $display("\n rand_mode(1) - Individual");
        sf.data.rand_mode(1); //Turn ON data random mode
        sf.addr.rand_mode(1); //Turn ON addr random mode

        repeat (2) begin sf.randomize();
            sf.disp;
        end

        $display("\n rand_mode(0) - ALL");
        sf.rand_mode(0); //Turn OFF ALL random variables

        repeat (2) begin
            sf.randomize();
            sf.disp;
        end

        $display("\n rand_mode(1) - ALL");
        sf.rand_mode(1); //Turn ON ALL random variables
```

```

repeat (2) begin
    sf.randomize();
    sf.disp;
end

end
endmodule

```

*Simulation log:*

rand\_mode(0) – Individual

data = 0

addr = 0

data = 0

addr = 0

rand\_mode(1) – Individual

data = 3

addr = 9

data = 9

addr = b

rand\_mode(0) – ALL

data = 9

addr = b

data = 9

addr = b

rand\_mode(1) – ALL

data = 4

addr = a

data = 8

addr = 7

### V C S   S i m u l a t i o n   R e p o r t

In class “rmode” we declare two rand variables, data and addr. In module TOP, we first turn *off* the randomness of both these variables individually using rand\_mode(0). The simulation log shows that none of the randomization takes place:

rand\_mode(0) – Individual

data = 0

addr = 0

data = 0

addr = 0

Next, we turn *on* the randomness of both variables individually, and the simulation log shows that both data and addr now have random values:

rand\_mode(1) – Individual

```
data = 3
addr = 9
data = 9
addr = b
```

Now, we turn off randomness of all variables by directly turning *off* rand\_mode for the object “sf” itself. This will retain the last values of data and addr, as shown in the simulation log. The last data value (= 9) and addr value (= b) will reflect on data and addr when randomization is turned off:

```
rand_mode(0) - ALL
data = 9
addr = b
data = 9
addr = b
```

Finally, we turn *on* randomness of all variables directly turning *on* rand\_mode of the object “sf.” The randomness is shown in the simulation log:

```
rand_mode(1) - ALL
data = 4
addr = a
data = 8
addr = 7
```

A compiler error will be issued if the specified variable does not exist within the class hierarchy or it exists but is not declared as rand or randc. This is rather obvious.

Also, when called as a function, rand\_mode( ) returns the current active state of the specified random variable. It returns 1 if the variable is active (*on*) (i.e., the variables are randomized) and 0 if the variable is inactive (*off*) (i.e., the variables are not randomized). Here is a snippet:

```
if (sf.data.rand_mode( ) == 1) $display("data is random");
```

### 13.9 constraint\_mode( ): Control Constraints

Analogous to rand\_mode( ), there is also the constraint\_mode( ) that turns *on/off* constraints. The constraint\_mode( ) method can be used to control whether a constraint is active or inactive. When a constraint is inactive, it is not considered by the randomize( ) method. All constraints are initially active.

When <constraint\_identifier>.constraint\_mode(0) is invoked, it will set the specified constraint to an inactive state so that it is not evaluated by subsequent randomize( ) calls.

When <constraint\_identifier>.constraint\_mode(1) is invoked, it will set the specified constraint to an active state so that it is evaluated by subsequent randomize( ) calls.

When called as a function, constraint\_mode( ) returns the current active state of the specified constraint block. It returns 1 if the constraint is active (*on*) and 0 if the constraint is inactive (*off*).

Here is an example:

```
class sft;
  rand bit [3:0] data, addr;
  constraint data_range1 { data <= 'h4; }
  constraint addr_range1 { addr > 'ha; }
endclass

module soft_constr;
  initial begin
    sft sf;
    sf = new();

    sf.data_range1.constraint_mode(0);
    sf.addr_range1.constraint_mode(0);

    repeat (4) begin
      sf.randomize();
      $display("addr = %h data = %h",sf.addr, sf.data);
    end

    sf.constraint_mode (1); //Turn ON all constraints

    repeat (4) begin
      sf.randomize();
      $display("addr = %h data = %h",sf.addr, sf.data);
    end
  end
endmodule
```

#### *Simulation log:*

addr = f data = 5

addr = 3 data = e

addr = 8 data = 6

addr = a data = c

addr = d data = 3

```
addr = d data = 4
addr = e data = 2
addr = b data = 4
```

### V C S   S i m u l a t i o n   R e p o r t

In this example, we have two constraints in class “sft,” namely, `data_range1` and `addr_range1`. The `data_range1` constraints restrict “`data`” to `<= 'h4`, while the `addr_range1` restricts “`addr`” to `> 'ha`.

In the module `TOP`, we turn off both these constraints. As you see from the simulation log, “`addr`” does not conform to its constraint anymore (“`addr`” is not necessarily “`> 'ha`” anymore). Similarly, “`data`” does not conform to its constraint anymore (“`data`” is not necessarily “`<= 'h4`” anymore). Then we turn *on* all the constraints (`sf.constraint_mode(1)`). As you can see in the second part of the log, “`addr`” and “`data`” now conform to their constraints.

## 13.10 randomize( ) with Arguments: In-Line Random Variable Control

So far we have seen the use of `randomize( )` without any arguments. But you can indeed use `randomize( )` with arguments to control the set of random variables and state variables (non-random variables) within a class instance. As we have seen, when the `randomize( )` method is called with no arguments, it assigns new values to *all* random variables in an object – those declared as `rand` or `randc` – so that *all* of the constraints are satisfied. When `randomize( )` is called with arguments, those arguments designate the complete set of random variables within that object; all other variables in the object are considered state non-random variables. This is useful when you want to randomize non-`rand` variables, as we will see in the example below.

Arguments are limited to the names of properties of the calling object; expressions are not allowed.

It is important to note that calling `randomize( )` with arguments allows changing the random mode of any class property, *even those not declared as rand or randc*. Yes, it allows you to randomize even those properties that are *not* declared random.

Here is an example:

```
class sft;
    rand bit [3:0] a,b;
    bit [3:0] c,d;
    constraint a_range1 { a <= 'h4; }
    constraint b_range1 { b > 'ha; }
endclass

module soft_constr;
```

```
initial begin
    sft sf;
    sf = new( );

repeat (2) begin

//randomize all rand variables in 'sf' - but only a, b will be
//randomized since c, d are not 'rand'
    sf.randomize( );

    $display("randomize a, b : a=%h b=%h c=%h d=%h",sf.a, sf.b,
sf.c, sf.d);
    end

repeat (2) begin

// randomize only non-rand variables 'c' and 'd'
// a, b, will not be randomized
    sf.randomize(c, d);

    $display("randomize c, d : a=%h b=%h c=%h d=%h",sf.a, sf.b,
sf.c, sf.d);
    end

repeat (2) begin
    sf.randomize(a); //randomize only rand variable 'a' .
                    // b, c, d will not be randomized
    $display("randomize a : a=%h b=%h c=%h d=%h",sf.a, sf.b,
sf.c, sf.d);
    end

repeat (2) begin
    sf.randomize(b); //randomize only rand variable 'b'
                    // a, c, d will not be
randomized
    $display("randomize b : a=%h b=%h c=%h d=%h",sf.a, sf.b,
sf.c, sf.d);
    end

repeat (2) begin
    sf.randomize(c); //randomize only non-rand variable 'c'
                    /// a, b, d will not be
randomized
```

```

$display("randomize c : a=%h b=%h c=%h d=%h",sf.a, sf.b,
sf.c, sf.d);
end

repeat (2) begin
    sf.randomize(d); //randomize only non-rand variable 'd'
                      // a , b, c will not be
randomized
    $display("randomize d : a=%h b=%h c=%h d=%h",sf.a, sf.b,
sf.c, sf.d);
end

if (sf.randomize(null) == 1);
    $display("randomize(null) : a=%h b=%h c=%h d=%h",sf.a, sf.b,
sf.c, sf.d);
end

endmodule

```

*Simulation log:*

```

randomize a,b : a=1 b=f c=0 d=0
randomize a,b : a=4 b=c c=0 d=0

randomize c,d : a=4 b=c c=a d=6
randomize c,d : a=4 b=c c=7 d=2

randomize a : a=3 b=c c=7 d=2
randomize a : a=4 b=c c=7 d=2

randomize b : a=4 b=d c=7 d=2
randomize b : a=4 b=f c=7 d=2

randomize c : a=4 b=f c=0 d=2
randomize c : a=4 b=f c=9 d=2

randomize d : a=4 b=f c=9 d=5
randomize d : a=4 b=f c=9 d=0
randomize(null): a=4 b=f c=9 d=0

```

### V C S   S i m u l a t i o n   R e p o r t

Let us study the example, one subset at time. First, we randomize( ) all the rand variable of class instance “sf.” This will randomize the variables “a” and “b.” Variables “c” and “d” will not be randomized since they are not declared “rand” (or “randc”). That is shown in simulation log as follows:

```
randomize a, b : a=1 b=f c=0 d=0
```

```
randomize a, b : a=4 b=c c=0 d=0
```

Next, we randomize( ) variables “c” and “d.” Note that “c” and “d” are *not* declared rand in the class “sft.” Also *note that since we are calling randomize(c, d), it will only randomize “c” and “d” and not “a” and “b.”* As you see in the simulation log, “a” and “b” retain their old values, while “c” and “d” take on random values:

```
randomize c,d : a=4 b=c c=a d=6  
randomize c,d : a=4 b=c c=7 d=2
```

Next, we randomize(a). This means we randomize only “a.” “b,” “c,” and “d” are not randomized (even though “b” is rand). In the simulation log, you see that only “a” is randomized; others retain their previous non-randomized values:

```
randomize a : a=3 b=c c=7 d=2  
randomize a : a=4 b=c c=7 d=2
```

Next, we randomize(b). This means we randomize only “b.” “a,” “c,” and “d” are not randomized (even though “a” is rand). In the simulation log, you see that only “b” is randomized; others retain their previous non-randomized values:

```
randomize b : a=4 b=d c=7 d=2  
randomize b : a=4 b=f c=7 d=2
```

Next, we randomize(c). This means we randomize only “c,” and “a,” “b,” and “d” are not randomized (even though “a” and “b” are rand). In the simulation log, you see that only “c” is randomized; others retain their previous non-randomized values:

```
randomize c : a=4 b=f c=0 d=2  
randomize c : a=4 b=f c=9 d=2
```

Finally, we randomize(d). This means we randomize only “d,” and “a,” “b,” and “c” are not randomized (even though “a” and “b” are rand). In the simulation log, you see that only “d” is randomized; others retain their previous non-randomized values:

```
randomize d : a=4 b=f c=9 d=5  
randomize d : a=4 b=f c=9 d=0
```

Note also that you can call randomize(null). Such a call accepts the special argument *null* to indicate no random variables exist in the class. In other words, all class members behave as non-random state variables. This causes the randomize( ) method to behave as a checker instead of a generator. A checker evaluates all constraints and simply returns 1 if all constraints are satisfied and 0 otherwise.

You can see in the simulation log that randomize(null) turns off randomization of all the variables:

```
randomize(null): a=4 b=f c=9 d=0
```

## 13.11 Random Number Generation System Functions and Methods

SystemVerilog provides the following system functions and methods to further augment the constrained random verification methodology:

```
$urandom()
$urandom_range()
srandom()
get_randstate()
set_randstate()
```

Let us look at each in detail. But first what is a Random Number Generator?

### 13.11.1 Random Number Generator (RNG)

The element responsible for generating random values in SystemVerilog is called Random Number Generator, abbreviated RNG. Each thread, package, module instance, program instance, interface instance, or class instance has a built-in RNG. Thread, module, program, interface, and package RNGs are used to select random values for \$urandom( ) (as well as \$urandom\_range( ), randomize( ), rand-sequence, randcase, and shuffle( )) and to initialize the RNGs of child threads or child class instances. A class instance RNG is used exclusively to select the values returned by the class's predefined randomize( ) method (Ahmed Yehia, Mentor Graphics).

Whenever an RNG is used either for selecting a random value or for initializing another RNG, it will “change state” so that the next random number it generates is different. Therefore, the random value that a specific randomization call generates depends on the number of times the RNG has been used and on its initialization. Initializing the RNG, in turn, depends on the number of times its parent RNG has been used and on the parent RNG initialization. The topmost RNG is always a module, program, interface, or package RNG, and all of these RNGs are initialized to the same value, which is chosen by the simulator according to the simulation seed. For a constraint, given randomize( ) call, the process is essentially the same up to the point where the object is allocated. Once the object is allocated, it gets its own RNG which, unlike package, module, program, interface or thread RNGs, changes state only when randomize() is called. Therefore, from instantiation point onward, the only instructions that affect the results of a given randomize( ) call are earlier randomize( ) calls.

Further discussion on RNG is scattered throughout this chapter, under various topics.

### **13.11.2 \$urandom( ) and \$urandom\_range( )**

\$urandom( ) function provides a mechanism to generate pseudo-random values. It will return an unsigned 32-bit random number when it is called. Here is the syntax:

```
function int unsigned $urandom [ (int seed) ];
```

Note that “seed” is an optional argument that determines the sequence of random numbers generated. This is for the predictability of random number generation. In other words, the same sequence of random numbers will be generated every time the same seed is used. “seed” is very important for regression runs where each run needs to work with the same sequence of random numbers. Providing a new seed every time you run simulation will give you a different set of random numbers.

There is also \$urandom\_range function that returns an unsigned integer within the specified range. Its syntax is:

```
function int unsigned $urandom_range(int unsigned maxval, int
unsigned minval);
```

If maxval is less than minval, then the number generator automatically reverses the arguments so that the first argument is larger than the second argument. If you provide only 1 value, as in \$urandom(15), then you will get numbers in the 15...0 range.

Here is an example:

```
module random;

integer in1, in2, sinl;
bit [63:0] bil;
int seed,il;

initial begin

$display("\n$urandom( ) - same seed");
seed=1234;
repeat (4) begin //same seed through each iteration
    in1 = $urandom(seed); //with seed
    in2 = $urandom & 'h0000_00ff; //Mask top 24-bits
    bil = {$urandom, $urandom}; //64-bit random number
    $display("in1=%0h in2=%0h bil=%0h",in1,in2,bil);
    #1;
end
```

```

$display("\n$urandom( ) - changing seed");
seed=1234;
repeat (4) begin
    seed=seed+1; //different seed through each iteration
    in1 = $urandom(seed); //with seed
    in2 = $urandom & 'h0000_00ff; //Mask top 24-bits
    bi1 = {$urandom, $urandom}; //64-bit random number
    $display("in1=%0h in2=%0h bi1=%0h",in1,in2,bi1);
    #1;
end

$display("\n$urandom_range( )");
repeat (4) begin
    in1 = $urandom_range(15,7); //max_value > min_value
    in2 = $urandom_range(0,15); //max_value < min_value
    bi1 = $urandom_range(7); //No min_value. So, 7...0
    $display("in1=%0d in2=%0d bi1=%0d",in1,in2,bi1);
end

process::self().srandom(1234);
in1 = $urandom; // seed set by srandom
$display("in1=%h",in1);

end
endmodule

```

*Simulation log:*

```

$urandom( ) - same seed
in1=d473f645 in2=ce bi1=46d3933a17f96ab4
in1=d473f645 in2=ce bi1=46d3933a17f96ab4
in1=d473f645 in2=ce bi1=46d3933a17f96ab4
in1=d473f645 in2=ce bi1=46d3933a17f96ab4

```

```

$urandom( ) - changing seed
in1=f7c14f9 in2=8e bi1=d4d64b41c99e7e59
in1=4ae225b5 in2=ef bi1=59fa737b1b1b34a0
in1=30310970 in2=18 bi1=cfb535dbb9987038
in1=8de5b5bf in2=4c bi1=f16b5c1f826512c5

```

```

$urandom_range()
in1=12 in2=2 bi1=4
in1=8 in2=9 bi1=7
in1=11 in2=1 bi1=0
in1=14 in2=1 bi1=7
in1=d473f645

```

### VCS Simulation Report

In this example, we are using \$urandom with a seed. We first declare a seed=1234; (can be any “int”) and do *not* change it through the iterations of the loop. The idea is that you will get the same random number with the same seed. Since we do not change the seed, the random numbers are the same for all four iterations of the loop.

Then in the second loop, we increment the seed and generate different numbers using this seed. Each new seed will generate a new set of random numbers:

```
in1 = $urandom(seed);
```

will generate random numbers based on “seed” as shown in the simulation log (“in1”). We then use a masking set of bits to show that you can effectively create restricted random numbers:

```
in2 = $urandom & 'h0000_00ff; //Mask top 24-bits
```

This is shown in the simulation log (“in2”), where only the eight least significant bits are produced.

If we want to generate a 64-bit random value, we do the following (note that \$urandom only returns 32-bit numbers):

```
bi1 = {$urandom, $urandom}; //64-bit random number
```

This is shown in simulation log for “bi1.”

We then use \$urandom\_range to generate random numbers in a given range:

```
in1 = $urandom_range(15,7); // maxval > minval
```

In this case, the maxval is greater than minval. So, the numbers will be generated between 15 and 7, as shown in simulation log (“in1”). We then use the function with maxval less than minval:

```
in2 = $urandom_range(0,15); // maxval < minval
```

In this case, the simulator will reverse the order of these values and generate numbers between 15 and 0. This is shown in simulation log with “in2.”

Lastly, we provide only 1 value to \$urandom\_range. This value will be regarded as the maxval and the minval will be considered 0. Hence, for \$urandom\_range(7), we will get values from 7 to 0 (inclusive of 7 and 0). This is shown in simulation log for “bi1.”

### 13.11.3 *srandom( ), get\_randstate( ), and set\_randstate( )*

Let us continue with the previous discussion on RNG. Instead of depending on the absolute execution path for a thread or on the ordering of an object construction, the RNG of a given thread or an object can be manually set to a specific known state. This makes the execution path up to a point “do not care.” This is known as manual seeding, which is a powerful tool to guarantee random stability upon minimal code changes. Manual seeding can be performed using methods such as:

*srandom( )*: Takes an integer argument acting as the seed. Once called on a process id or a class object, it manually sets the process (or object) RNG to a specific known state, making any subsequent random results depend only on the relative execution path from the manual seeding point onward.

*get\_randstate( )/set\_randstate( )*: Used together to shield some code from subsequent randomization operations. You can *get\_randstate* and store it in some variable and use that value to *set\_randstate* to start RNG from that state regardless of other randomization that took place between *get\_randstate* and *set\_randstate*. We will see examples below.

*srandom( )*: Allows you to specify the seed that is used by an object or process. *srandom( )* takes an integer argument acting as the seed. The *srandom( )* method initializes the current Random Number Generator (RNG) of objects or threads using the value of the seed. Each object maintains its own internal RNG, which is used exclusively by its *randomize( )* method. This allows objects to be randomized independently of each other and of calls to other system randomization functions.

When an object is created, its RNG is seeded using the next value from the RNG of the thread that creates the object. This process is called hierarchical object seeding.

When a thread is created, its RNG state is initialized using the next random value from the parent thread as a seed.

So, sometimes it is desirable to manually seed an object’s RNG using the *srandom( )* method.

The syntax of *srandom( )* is:

```
function void srandom (int seed);
```

Here is a simple example on how to set *srandom( )*. The example shows how to use *srandom( )* to set *srandom(seed)* and also to see that given the same seed, you will get the same random value:

```
module TOP;
class bus;
    rand logic [7:0] x;
endclass

int d2,d3,d4;

bus b1;

initial begin
    b1 = new;

    b1.srandom(1234); //provide seed for RNG of 'b1'

    if (b1.randomize()); //if randomize() is successful
        d2 = b1.x; //assign value of the variable x to d2
    $display("d2 = %0h",d2);

    b1.srandom(2345); //provide a new seed

    if (b1.randomize());
        d3 = b1.x;
    $display("d3 = %0h",d3);

    b1.srandom(1234); //repeat the earlier seed

    if (b1.randomize());
        d4 = b1.x;
    $display("d4 = %0h",d4);

    if((d2 == d4) && (d2 != d3))
        $display("test passed");
    else
        $display("test failed");

end
endmodule
```

*Simulation log:*

```
d2 = 65
d3 = c6
d4 = 65
test passed
```

### VCS Simulation Report

In this example, we declare a rand variable “x” in class “bus.” Then, in the initial block, we first use *random(1234)* as the seed for RNG of “b1.” This initializes the current Random Number Generator (RNG) of “b1” using the value of the seed. We then randomize instance “b1” and store the random value of “x” to a variable “d2.” We then use a different seed using *random(2345)* to seed the RNG and change randomization of “x.” We store this new random value of “x” into “d3.” We then reuse *random(1234)*, i.e., use the same seed as before to initialize the current RNG of “b1.” We then randomize “b1” to get the value of “x” and store it in “d4.” This first value of “x” (“d2”) should be the same as the new value of “x” (“d4”). This is shown in the simulation log:

#### *get\_randstate( ) / set\_randstate( ):*

In SystemVerilog the initial seed is a 32-bit integer that gets converted to a much larger internal Random Number Generator (RNG) value that you can get with *get\_randstate( )*. The *get\_randstate()* method retrieves the current state of an object’s RNG. The syntax of *get\_randstate( )* is:

```
function string get_randstate( );
```

The *get\_randstate( )* method returns a copy of the internal state of the RNG associated with the given object. The RNG state is a string of unspecified length and format. The length and contents of the string are implementation dependent.

The *set\_randstate( )* method sets the state of an object’s RNG. The *set\_randstate( )* method copies the given state into the internal state of an object’s RNG. The RNG state is a string of unspecified length and format. The place to use *set\_randstate( )* is with a value string returned by an earlier *get\_randstate( )*. You do this when you need to preserve the RNG from being disturbed by some other operation.

Calling *set\_randstate( )* with a string value that was not obtained from *get\_randstate( )* is undefined.

Here is the syntax of *set\_randstate( )*:

```
function void set_randstate (string state);
```

Here is an example of usage of *get\_randstate( )* and *set\_randstate( )*:

```
class crand;
  rand bit [31:0] addr;
endclass
```

```

module p1;
  int in1;
  typedef bit [7:0] chars[];

  string state;
  crand c3 = new( );
  initial begin
    state = c3.get_randstate( );
    $displayh("Initial RNG State = %0p", chars'(state));

    c3.srandom(1234); //Manually Change the seed of RNG

$      d      i      s      p      l      a      y      h      (      "      A      f      t      e      r
srandom: RNG state = %0p", chars'(c3.get_randstate()));

    void'(c3.randomize( )); //
    $displayh("After randomize( ): RNG state = %0p", chars'(c3.
get_randstate()));

    c3.set_randstate(state); //set_randstate using 'state'
    $displayh("After set_randstate: RNG state = %0p", chars'(c3.
get_randstate()));

  end
endmodule

```

*Simulation log:*

```

# run -all
# Initial RNG State = 4d 53 34 31 34 32 35 65 37 63 39 66 61 66 66 30 37 37 36 35
   63 39 61 38 32 61 38 31 65 32 39 33 34 62
# After srandom: RNG state = 4d 53 37 32 64 66 64 66 32 35 39 37 66 37 64 30 37
   37 39 66 32 65 35 36 30 32 63 39 36 30 61 31 65 35
# After randomize(): RNG state = 4d 53 39 37 66 37 64 30 37 37 64 62 39 34 35 65
   39 31 31 34 34 39 38 62 37 62 32 33 64 61 31 61 65 37
# After set_randstate: RNG state = 4d 53 34 31 34 32 35 65 37 63 39 66 61 66 66
   30 37 37 36 35 63 39 61 38 32 61 38 31 65 32 39 33 34 62
# exit

```

In this example, we first create instance “c3” of class “crand.” We then get the RNG value of this object “c3” using `get_randstate()` method and store it in string “state”:

```
state = c3.get_randstate();
$displayh("Initial RNG State = %0p", chars'(state));
```

The simulation log shows this initial RNG state as follows:

```
Initial RNG State = 4d 53 34 31 34 32 35 65 37 63 39 66 61 66 66 30 37 37 36 35
63 39 61 38 32 61 38 31 65 32 39 33 34 62
```

We then manually seed “c3” using srandom(1234) and again get the current state of an “c3”’s RNG value:

```
c3.srandom(1234); //Manually Change the seed of RNG
$ d i s p l a y h ("A f t e r
srandom: RNG state = %0p", chars'(c3.get_randstate()));
```

This is shown in simulation log as:

```
After srandom: RNG state = 4d 53 37 32 64 66 64 66 32 35 39 37 66 37 64 30 37
37 39 66 32 65 35 36 30 32 63 39 36 30 61 31 65 35
```

Note that this RNG value is different from the one we got as the initial RNG value. “srandom” changed the state of RNG.

We then randomize “c3” and get the new RNG value of “c3”:

```
void'(c3.randomize()); //
$displayh("After randomize(): RNG state = %0p", chars'(c3.
get_randstate()));
```

The simulation log shows this RNG state as follows:

```
After randomize(): RNG state = 4d 53 39 37 66 37 64 30 37 37 64 62 39 34 35 65
39 31 31 34 34 39 38 62 37 62 32 33 64 61 31 61 65 37
```

Note that this RNG value is different from the previous two RNG values.

We then set the randstate using the value we got from an earlier get\_randstate which was stored in “state”:

```
c3.set_randstate(state); //set_randstate from 'state'
$displayh("After set_randstate: RNG state = %0p", chars'(c3.
get_randstate()));
```

This is to allow us to reset the RNG state to a previous RNG state. The set\_randstate( ) method sets the state of “c3”’s RNG. The set\_randstate( ) method copies the given state into the internal state of an “c3”’s RNG.

Note that the value obtained after “set\_randstate” is the same as the initial RNG value that we got before we called srandom( ) or randomize( ). So, this way no matter how the RNG was going about generating different RNG values for different

threads, we can still reset it to a previous value. Here is the display that shows that this RNG value is the same as “Initial RNG value”:

```
After set_randstate: RNG state = 4d 53 34 31 34 32 35 65 37 63 39 66 61 66 66 30  
37 37 36 35 63 39 61 38 32 61 38 31 65 32 39 33 34 62
```

So, this example shows how to get the RNG value of an object (or a thread) and how to set it back to the value received from a previous `get_randstate()`.

## 13.12 Random Stability

Random stability means how RNG is generated and how it gets affected by code changes in threads and object instances. It means that the random value returned by a thread or an object is independent of the RNG in other threads and objects. Random stability applies to how RNG is generated in relation to `$urandom`, `$urandom_range`, `randcase`, `rand` sequence (Sect. 13.14), `srandom` (Sect. 13.11.3) and `randomize()`.

There are two types of random stability properties, thread stability, and object stability (SystemVerilog-LRM).

Thread stability has the following properties:

- Each thread has an independent RNG for all randomization system calls invoked from that thread.
- When a new dynamic thread is created, its RNG is seeded with the next random value from its parent thread.

Object stability has the following properties:

- Each class instance (object) has an independent RNG for all randomization methods in the class.
- When an object is created using `new`, its RNG is seeded with the next random value from the thread that creates the object.
- Object stability is preserved when object and thread creation and random number generation are done in the same order as before. In order to maintain random number stability, new objects, threads, and random numbers can be created after existing objects are created.

Note that the same stimulus sequence cannot be produced on different simulators as the SystemVerilog – 2017 LRM does not restrict the EDA vendors to implement specific constraint solver. Each simulator will most likely give you different random numbers for the same thread or the object. `$urandom`, `$urandom_range`, and `randomize()` may give you different results on different simulators. There is also the question of order of thread creation and execution that will affect the RNG values generated by different vendors.

Let us see how thread and object stability works when we change (add/delete/rearrange) threads and objects. Here are a couple of examples where we simply rearrange the order of objects and see how that affects the random values generated.

Example 1:

```
class aClass;
    rand bit [2:0] addr;
endclass

class bClass;
    rand bit [2:0] data;
endclass

module tb;
    initial begin
        aClass ac = new (); //aClass instantiated first
        bClass bc = new (); //bClass instantiated next

        for (int i = 0; i < 5; i++) begin
            ac.randomize();
            $display ("i=%0d addr=%0d", i, ac.addr);
        end

        for (int i = 0; i < 5; i++) begin
            bc.randomize();
            $display ("i=%0d data=%0d", i, bc.data);
        end
    end
endmodule
```

*Simulation log:*

```
i=0 addr=3
i=1 addr=6
i=2 addr=1
i=3 addr=1
i=4 addr=5
i=0 data=5
i=1 data=3
i=2 data=1
i=3 data=6
i=4 data=6
VCS Simulation Report
```

A very simple example with two classes aClass and bClass each with a random variable, addr and data, respectively. Now, in module tb, we first instantiated aClass and then instantiated bClass. This is important to note because in Example 2, we will simply reverse the order of instantiate and see what happens. The simulation log shows the random values generated for addr and data.

#### Example 2:

In this example, we simply reverse the order of instantiation of aClass and bClass. Here is the new model:

```
class aClass;
    rand bit [2:0] addr;
endclass

class bClass;
    rand bit [2:0] data;
endclass

module tb;
    initial begin
        bClass bc = new (); //bClass instantiated first
        aClass ac = new (); //aClass instantiated next

        for (int i = 0; i < 5; i++) begin
            ac.randomize();
            $display ("i=%0d addr=%0d", i, ac.addr);
        end

        for (int i = 0; i < 5; i++) begin
            bc.randomize();
            $display ("i=%0d data=%0d", i, bc.data);
        end
    end
endmodule
```

Note that the only difference between Example 1 and Example 2 is that in Example 1 we had the following order of instantiation:

```
aClass ac = new (); //aClass instantiated first
bClass bc = new (); //bClass instantiated next
```

But in Example2 we reverse the order as follows:

```
bClass bC = new ( ); //bClass instantiated first
aClass aC = new ( ); //aClass instantiated next
```

With this new order, here is the simulation log.

*Simulation log:*

```
i=0 addr=5
i=1 addr=3
i=2 addr=1
i=3 addr=6
i=4 addr=6
i=0 data=3
i=1 data=6
i=2 data=1
i=3 data=1
i=4 data=5
```

#### V C S   S i m u l a t i o n   R e p o r t

Now, carefully note the random values on “addr” and “data” in each example. In Example 1, whatever values were randomized on “addr” now appear on “data” in Example 2. And whatever values were randomized on “data” in Example 1 now appear on “addr” in Example 2. So, the random values generated for “addr” and “data” changed because we changed the order of instantiation of aClass and bClass.

What happened? All we did was change the order of instantiation. Random stability comes into picture. Normally, stimulus generated in a thread or object is independent of the other stimulus.

*BUT this is valid as long as the order of the threads or the objects is not disturbed. If a new object is created, make sure that they are added at the end of previous objects. Very important point to note, if you want to produce consistent results from every new simulation run. This applies to threads also. If you are going to add a new thread (e.g., a new begin-end block), make sure that it is added after all the pre-existing threads. Else, RNG will generate different random values. While debugging to produce the same simulation, we should make sure that calls to RNG are not disturbed.*

Program and thread stability can be achieved as long as thread creation and random number generation are done in the same order as before. When adding new threads to an existing test, they can be added at the end of a code block in order to maintain random number stability of previously created work.

As an exercise, change the order of for loop in Examples 1 and 2. In one example, declare the for loop for aC.randomize followed by a for loop for bC.randomize. See the results. Then reverse the order of declaration of these for loops and see how the random values change.

### 13.13 Randcase

Just as in a normal “case” statement, the “randcase” is a case statement that randomly selects one of its branches. But in randcase there is also the notion of assigning random weights to randcase branches. You assign a weight to each branch of randcase statement and then use that weight to assign a probability with which that branch will be executed.

Here is an example:

```
module TOP;
    int x;

    initial begin
        repeat (10) begin
            randcase
                5 : x = 1; //weight : item = value;
                15 : x = 2;
                25 : x = 3;
                10 : x = 4;
            endcase
            $display("x = %0d", x);
        end
    end
endmodule
```

*Simulation log:*

```
x = 4
x = 3
x = 1
x = 2
x = 3
x = 3
x = 1
x = 2
x = 3
x = 3
```

V C S   S i m u l a t i o n   R e p o r t

In this example, we declare a randcase and assign different weights to its branches. The sum of all weights is 55. So, the probability of execution of branch with weight 5 is 5/55 (0.09). The probability of branch with weight 15 is 15/55 (0.27). Similarly, the probability for branch with weight 25 is 25/55 (0.45) and for

branch with weight 10 is 10/50 (0.2). So, weight 25 has the highest probability, and weight 5 has the lowest probability. This is evident from simulation log where “x = 3” with weight 25 occurs more often than other expressions.

Let us take this example a bit further and prove that probability distribution in the above example is indeed as we discussed:

```
module TOP;
    int x;
    real c1, c2, c3, c4;
    real p1;

    initial begin
        c1=0; c2=0; c3=0; c4=0;
        repeat (1000) begin
            randcase
                5 : x = 1;
                15 : x = 2;
                25 : x = 3;
                10 : x = 4;
            endcase
            if (x == 1) c1 = c1+1;
            if (x == 2) c2 = c2+1;
            if (x == 3) c3 = c3+1;
            if (x == 4) c4 = c4+1;
        end
        $display("c1 = %0d c2 = %0d c3 = %0d c4 = %0d",
                 c1, c2, c3, c4);

        $display("Probability of x = 1; is %0f", c1/1000);
        $display("Probability of x = 2; is %0f", c2/1000);
        $display("Probability of x = 3; is %0f", c3/1000);
        $display("Probability of x = 4; is %0f", c4/1000);
    end
endmodule
```

This example is identical to the one before it, in terms of the weighted distribution of randcase branches. We are simply counting the number of times any of the case expressions (x=1; x=2; x=3; x=4) get executed. We then calculate the probability by dividing the count by 1000 (which is the repeat loop count). Here is the simulation log.

*Simulation log:*

c1 = 88 c2 = 260 c3 = 455 c4 = 197

Probability of x = 1; is 0.088000

Probability of x = 2; is 0.260000

Probability of x = 3; is 0.455000

Probability of x = 4; is 0.197000

### V C S   S i m u l a t i o n   R e p o r t

The simulation log shows that the probabilities are close to what we had predicted. The sum of all weights is 55. X = 1 and has weight of 5. So, its probability is  $5/55 = 0.09$ . X = 2 and has a weight of 15, so its probability is  $0.27$  ( $15/55$ ). X = 3 has the weight 25, so its probability is  $0.45$  ( $25/55$ ), and X = 4 has the weight of 10, so its probability is  $.2$  ( $10/55$ ).

## 13.14 randsequence

The randsequence generator is useful for randomly generating structured sequences of stimulus as CPU instructions, cache transactions, network traffic generation, etc.

Constrained randomization is an acceptable approach, but is it the most optimal approach for all types of stimulus that is required for verification? For example, generation of processor instruction sequences. An effort to build such a sequence of instructions starting from basic random variables and then using constraints to create valid instruction sequence is rather difficult and time-consuming. And, if additional instructions are introduced later, making changes is not easy. That is where randsequence comes into picture. If the possible sequences are represented in the form of a tree or graphs, then randsequence allows one to formally capture such a structure.

Also, as one goes from block-level verification to the SoC level, the level of randomness keeps reducing; in other words, the level of constraints increases. As we go from block- to SoC-level verification, strict sequencing becomes critical, and randomization has to be done keeping this sequencing requirement in mind.

The features of randsequence are described in terms of the so-called productions. Each production contains a name and a list of production items. These productions are further classified as terminal or non-terminal. A terminal production is completely specified by its code block. It is an indivisible item that

needs no further definition than its associated code block. A non-terminal production is further divided into terminal (or further non-terminal) productions. What does all this mean?

Let us look at an example, since that is the best way to understand productions:

```
module TOP( );
initial
repeat(20)
randsequence( prodM ) //TOP level production
//prodM is non-terminal production
```

```

prodM : do1 | do2 | do3 | poppush;
//poppush is non-terminal production
poppush : pop | push;

pop : {$display("\t pop");}; //terminal production
push : {$display("\t push");}; //terminal production
do1 : {$display("do1");}; //terminal production
do2 : {$display("do2");}; //terminal production
do3 : {$display("do3");}; //terminal production
endsequence
endmodule

```

*Simulation log:*

```

do1
do2
do1
do1
    pop
do2
do3
do3
    pop
    pop
do1
    pop
do2
    pop
do1
    pop
do1
do3
do2
do1
do3
push

```

V C S S i m u l a t i o n R e p o r t

The production “prodM” is defined in terms of four non-terminal productions, namely, “do1,” “do2,” “do3,” and “poppush.” The “poppush” is further decomposed into two productions “pop” and “push.” Then, do1, do2, do3, pop, and push are declared as terminal since their \$display statements are self-contained code blocks.

When the first (top) production (“prodM”) is generated, it is decomposed into its productions, which specify a random choice between do1, do2, do3, and poppush sequences. Similarly, poppush production is further divided into downward productions of pop and push sequences. All other productions are terminals; they are completely specified by their code block, which in the example displays the production

name. Thus, the grammar leads to possible outcomes, as shown in the simulation log.

When the randsequence statement is executed, it generates a grammar-driven stream of random productions. As each production is generated, the side effects of executing its associated code blocks produce the desired stimulus. The randsequence keyword can be followed by an optional production name (inside the parentheses – “prodM” in our example) that designates the name of the top-level production. If unspecified, the first production becomes the top-level production.

### 13.14.1 Random Production Weights and If-Else Statement

Just as we saw in weighted distribution in a constraint block, we can assign weights to production lists. The probability that a particular production list is generated is proportional to its specified weight.

The operator is `:=` to assign weights. It assigns the weight specified by the *weight\_specification* to its production list. Here is a simple example:

```
bus : addr := 4 | data := 2 ;
```

This defines a production “bus” in terms of two weighted production lists, “addr” and “data.” “addr” will be generated with 67% (4/6) probability, and “data” will be generated with 33% (2/6) probability.

If no weight is specified, the production uses a weight of 1. If only some of the weights are specified, then the unspecified weights will use a weight of 1.

Let us look at a working example. I have also included how if-else works with productions:

```
module TOP( );
    int cpop, cpush, cdol, cdo2, cdo3;

initial begin
repeat(100)begin
    randsequence( prodM ) //TOP level production

        //weighted production
    prodM : dol := 1 | do2 := 2 | do3 := 3 | poppush;
    poppush : pop := 1 | push := 2; //weighted production

    pop : {cpop = cpop+1;
        if (cpop > 5)
            $display ("if: cpop=%0d",cpop);
        else $display("else branch: do nothing");};
```

```

push : {cpush = cpush+1;
        dispPush;}; //invoke a task

do1 : {cd01 = cd01+1;};
do2 : {cd02 = cd02+1;};
do3 : {cd03 = cd03+1;};
endsequence
end
$display("\n pop=%0d push=%0d do1=%0d do2=%0d do3=%0d", cpop,
cpush, cd01, cd02, cd03);
end

task dispPush;
begin
  if (cpush >= 13) $display ("if: cpush=%0d", cpush);
end
endtask
endmodule

```

In this example, for the productions “prodM” and “poppush,” we assign weights to assign probabilities of their generation:

```

prodM : do1 := 1 | do2 := 2 | do3 := 3 | poppush; //weighted production
poppush : pop := 1 | push := 2; //weighted production

```

You notice that different weights are assigned in each non-terminal productions. “do3” has higher weight than “do2” which has higher weight than “do1.” So, “do3” will be generated more often than “do2” which will occur more often than “do1.” Similarly, “push” has a higher weight than “pop,” and so “push” will occur more often than “pop.” These weights determine the probabilities with which the terminal productions occur. We will see this effect in the simulation log.

Then in each production, we increase a count that determines how many times that particular production was generated. This is to assess the probability of each production. Here is the code:

```

pop : {cpop = cpop+1;
       if (cpop > 5)
         $display ("if: cpop=%0d", cpop);
       else $display("else branch: do nothing");};

push : {cpush = cpush+1;
        dispPush;};

do1 : {cd01 = cd01+1;};

```

```

do2 : {cd02 = cd02+1;};
do3 : {cd03 = cd03+1;};

...
task dispPush;
begin
  if (cpush >= 13) $display ("if: cpush=%0d", cpush);
end

```

Note that in the above code, we are also using an if-else statement with production “pop.” This is a very simple if-else, but you can have complex hierarchical “if-else” to express your functionality. Also, with the production “push,” we are invoking a task “dispPush.” This is another way to add complex functionality to a production. We then display the count of each production.

Here is the *simulation log*:

```

else branch: do nothing
else branch: do nothing
else branch: do nothing
if: cpush=13
else branch: do nothing
else branch: do nothing
if: cpop=6
pop=6 push=13 do1=12 do2=22 do3=47

```

#### V C S   S i m u l a t i o n   R e p o r t

As you notice the output of “if-else” is correct and so does the task “dispPush” works correctly. Note the count of each of the production. “pop” has a weight of 1, while “push” has a weight of 2. Hence, “push” will be generated more often than “pop.” So, in the simulation log, you see *pop=6 push=13*. Similarly, “do1” has a weight of 1. “do2” has a weight of 2 and “do3” has a weight of 3. Thus, do3 will occur more often than “do2” which will occur more often than “do1.” Simulation log shows this – *do1=12 do2=22 do3=47*.

### **13.14.2 Repeat Production Statement**

The repeat statement repeats a given production repeat number of times. Here is the syntax:

```
repeat (expression) production_item
```

The value specified by the “expression” is the number of times that corresponding production will be generated. The expression must be a non-negative integral value.

Here is an example:

```
module TOP( );
    int cdo1, cdo2, cdo3;

initial begin
    repeat(20)begin
        randsequence( prodM ) //TOP level production
        prodM : do1 := 1 | do2 := 2 | do3 := 3;

        do1 : {repeat ($urandom_range(2,10)) cdisp;}; //repeat
        //do1 : {cd01 = cd01+1;}//uncomment and see what happens
        do2 : {cd02 = cd02+1;};
        do3 : {cd03 = cd03+1;};
        endsequence
    end
    $display("\n do1=%0d do2=%0d do3=%0d",cd01,cd02,cd03);
end

task cdisp;
    begin
        cd01 = cd01+1;
    end
endtask
endmodule
```

In this example, we use a *repeat* expression for production “do1”:

```
do1 : {repeat ($urandom_range(2,10)) cdisp;}; //repeat
```

We are using \$urandom\_range expression, which means that every time we hit “do1,” that “cdisp” will be repeated a random number of times (between 2 and 10) depending on the value returned by \$urandom\_range. The rest of the code is similar to preceding examples.

*Simulation log:*

```
do1=16 do2=7 do3=10
```

```
V C S S i m u l a t i o n R e p o r t
```

do1=16 means that every time we hit “do1” production, a random number was generated, and “cdisp” was executed that many times, totaling 16 times. If we comment the line:

```
//do1 : {repeat ($urandom_range(2,10)) cdisp;};
```

And replace it with:

```
do1 : {cd01 = cd01+1;};
```

we will see the following result:

```
do1=3 do2=7 do3=10
```

```
V C S   S i m u l a t i o n   R e p o r t
```

So, “do1” was hit only three times. Compare that to the result where “do1 = 16.” You will know the reason.

### 13.14.3 *rand join*

Also known as interleaving productions, “rand join” is used to randomly interleave production sequences. The syntax is:

```
rand join [(expression) } production_item production_item ...
```

Best explained with an example (partly taken from (SystemVerilog-LRM)):

```
module TOP( );
initial begin
repeat(20)begin
  randsequence( prodM )

  prodM : rand join do1 do2; //rand join - top production

  do1 : A B; //non-terminal production
  do2 : C D; //non-terminal production
  A : {$write ("A");}; //terminal productions
  B : {$write ("B");};
  C : {$write ("C");};
  D : {$write ("D");};

  endsequence
  $display("\n");
end
end
endmodule
```

In this example, we use “rand join” in the top production “prodM.” This means that it will randomize “do1” and “do2” to randomly interleave these production sequences. Let us see the results we get.

*Simulation log* (note that there are four terminal productions: A, B, C, and D):

ACBD

CADB

ACBD

ABCD

CDAB

ACBD

CDAB

ABCD

CDAB

CABD

CABD

CDAB

ABCD

ABCD

ACBD

CADB

CADB

CADB

ACDB

CDAB

#### V C S   S i m u l a t i o n   R e p o r t

So, we see random sequence patterns among the four terminal productions (A, B, C, D). Note that B always comes after A and D always comes after C. Can you figure out why? If you look at the sequences, you will see that the following unique sequences are interleaved while maintaining the relative order of each sequence:

A B C D

A C B D

A C D B

C D A B

C A B D  
C A D B

Exercise: Remove “rand join” from the statement `prodM : rand join do1 do2;` and see what happens? Will you get randomized sequences?

Now, let us look at a real-life example. Recall the (older!) parallel bus PCI. PCI has many different cycle types. And one needs to generate sequences in any random order to exercise all possible permutations. PCI has following cycles:

MemRead – Memory Read  
MemRMult – Memory Read Multiple  
MemReadLine – Memory Read Line  
MemWrite - MemWrite  
MemWrInv – MemWrite Invalid  
IORead – IO Read  
IOWrite – IO Write  
ConfRead – Configuration Read  
ConfWrite – Configuration Write

The key to exercising such a bus is to be able to randomly fire one type of cycle after another randomly picked cycle. For example, MemRead -> MemWrite -> IORead -> ConfRead -> IOWrite and so on. Simply verifying each cycle individually will not suffice. So, there is a random tree of events that we need to execute. Trying to do this simply with random variables and constraints will be very difficult, cumbersome, and error-prone. That is where “randsequence” comes into picture as we see in the following example:

```
module PCI_TOP( );
begin
    initial
        repeat(10) begin
            randsequence( PCI ) //TOP level production
            PCI : rand join MemReadOP MemWriteOP IOOP ConfOP ;
            MemReadOP : MemRead MemRMult MemReadLine ;
            MemWriteOP : MemWrite MemWrInv;
            IOOP : IORead IOWrite;
            ConfOP : ConfRead ConfWrite;

            MemRead : {$write("MemRead -> ")};
            MemRMult : {$write("MemRMult -> ")};
            MemReadLine : {$write("MemReadLine ->")};

            MemWrite : {$write("MemWrite -> ")};
        end
    end
endmodule
```

```

MemWrInv : {$write("MemWrInv -> ");};

IORRead : {$write("IORRead -> ");};
IOWrite : {$write("IOWrite -> ");};

ConfRead : {$write("ConfRead -> ");};
ConfWrite : {$write("ConfWrite -> ");};
endsequence
$display("\n");
end
endmodule

```

As you see from the simulation log below, we are executing random sequence of PCI cycles. We are using a repeat loop of only 10, but you can increase the loop count to make sure that you have covered full randomness of all the cycles.

*Simulation log:*

```

MemRead -> MemWrite -> MemRMult -> MemReadLine ->ConfRead ->
IORRead -> IOWrite -> MemWrInv -> ConfWrite ->

ConfRead -> IORRead -> ConfWrite -> IOWrite -> MemRead -> MemWrite ->
MemRMult -> MemWrInv -> MemReadLine ->

MemRead -> MemRMult -> IORRead -> MemWrite -> MemReadLine ->IOWrite ->
ConfRead -> ConfWrite -> MemWrInv ->

IORRead -> MemRead -> IOWrite -> MemWrite -> MemWrInv -> ConfRead ->
ConfWrite -> MemRMult -> MemReadLine ->

ConfRead -> IORRead -> MemRead -> IOWrite -> MemWrite -> MemWrInv ->
MemRMult -> ConfWrite -> MemReadLine ->

MemWrite -> ConfRead -> IORRead -> IOWrite -> MemWrInv -> MemRead ->
ConfWrite -> MemRMult -> MemReadLine ->

MemRead -> MemWrite -> MemRMult -> MemWrInv -> MemReadLine ->
ConfRead -> ConfWrite -> IORRead -> IOWrite ->

IORRead -> MemWrite -> MemRead -> ConfRead -> IOWrite -> ConfWrite ->
MemRMult -> MemWrInv -> MemReadLine ->

IORRead -> ConfRead -> MemWrite -> IOWrite -> MemWrInv -> ConfWrite ->
MemRead -> MemRMult -> MemReadLine ->

V C S  S i m u l a t i o n  R e p o r t

```

### 13.14.4 *break and return*

*break* and *return* can be used to terminate a production prematurely. They are different in that when a *break* statement is executed from within a production block, it forces a jump out of the “randsequence” block, while a *return* statement aborts the generation of the current production. With *return*, sequence generation continues with the next production following the aborted production.

Let us first look at “break” with an example:

```
module TOP();
    int cdo1, cdo2, cdo3;
    int rnum;

initial begin
repeat(10)begin
    rnum = $urandom_range(4,0);
    $display ("rnum=%0d",rnum);

    randsequence( prodM )

prodM : do1;

//break from productions A, B and randsequence
do1 : { if (rnum >= 3) break; } A B;

A : {$write ("A");}; //terminal productions
B : {$write ("B");};

endsequence
$display("\n");
end
end
```

In this example, we use a “break” statement in production “do1”:

```
do1 : { if (rnum >= 3) break; } A B;
```

This means that whenever random number “rnum” is  $\geq 3$ , abort productions A and B and quit randsequence. When “do1” is hit and if the condition is true, the entire sequence will be aborted. This is evident from the simulation log.

*Simulation log:*

rnum=3

rnum=0

AB

rnum=3

rnum=0

AB

rnum=2

AB

rnum=1

AB

rnum=1

AB

rnum=3

rnum=2

AB

rnum=2

AB

### V C S   S i m u l a t i o n   R e p o r t

As you see in the simulation log, whenever rnum is  $\geq 3$ , none of the productions get executed. Only when rnum is less than 3, the productions A and B get executed.

Now let us look at how *return* works. The example is similar to the one above, but we are using “*return*” instead of “*break*.“ Recall that “*return*” will only abort the production where it is applied, the rest of the productions will continue to be generated. Here is an example:

```
module TOP( );
    int cdo1, cdo2, cdo3;
    int rnum;

    initial begin
        repeat(10)begin
            rnum = $urandom_range (4,0);
            $display ("rnum=%0d", rnum);

            randsequence( prodM )
            prodM : do1 do2;

//abort only productions A, B
```

```
do1 : { if (rnum >= 3) return; } A B;
do2 : C D;
A : {$write ("A");}; //terminal productions
B : {$write ("B");};
C : {$write ("C");};
D : {$write ("D");};
endsequence
$display("\n");
end
end
endmodule
```

*Simulation log:*

rnum=3

CD

rnum=0

ABCD

rnum=3

CD

rnum=0

ABCD

rnum=2

ABCD

rnum=1

ABCD

rnum=1

ABCD

rnum=3

CD

rnum=2

ABCD

rnum=2

ABCD

V C S   S i m u l a t i o n   R e p o r t

In this example, we use a “return” with production “do1” to abort it. Compared to “break,” “return” will only abort the production where it is applied – not – the entire sequence. In the simulation log, we see that whenever rnum is  $\geq 3$ , only “do2” (C and D) productions get generated, since “do1” was aborted. When rnum is

less than 3, all productions get generated. To reiterate, when “do1” gets aborted, the “do2” continues to execute. The entire sequence is not aborted as with “break.”

### 13.14.5 Passing Values Between Productions

Just as in a SystemVerilog task, you can pass data values down to a production about to be generated. It uses the same syntax as a task. Productions that accept data include a formal argument list, and the syntax for declaring arguments is also the same as a task. Here is a simple example on how to pass values to a production:

```
module TOP();
    initial begin

        repeat(4)begin
            randsequence( prodM )
            prodM : rand join do1 do2;

            do1 : A B;
            do2 : C D;

            A : disp ("A"); //pass string "A" to production 'disp'
            B : disp ("B"); //pass string "B" to production 'disp'
            C : disp ("C"); //pass string "C" to production 'disp'
            D : disp ("D"); //pass string "D" to production 'disp'
            disp (string sDisp) : { $write (sDisp); }; //'disp' accepts
            string type

        endsequence
        $display("\n");
    end
end
endmodule
```

This example is very similar to preceding examples, except that each of the terminal production A, B, C, and D generates the production “disp” and passes it a string argument. “disp” accepts a string-type “sDisp” and displays it. Note the absence of { } brackets when production “disp” is generated from A, B, C, and D. Production takes in the string argument and displays it. This is a simple example that shows how to pass data values to a production. Note that we are using “rand join” in production “prodM,” and hence you see random streams being displayed by production “disp”. Here is the simulation log.

*Simulation log:*

ACBD

CADB

ACBD

ABCD

## V C S   S i m u l a t i o n   R e p o r t

### 13.14.5.1 Passing Return Value

In addition to passing values to a production, you can also return values *from* a production. Productions that return data require a type declaration – else you will get a compile error. The return type must precede the production. The return value is read in the code blocks of the production that triggered the generation of the production returning the value. Let us look at a simple example to put this in perspective:

```
module TOP();
initial begin
repeat(4)begin
    randsequence( prodM )
    prodM : do1 ;
    do1 : A B {$write("A=%h B=%h", A,B);};

    bit [7:0] A : { return $urandom; };
    //A : { return $urandom; };
    //Compile Error, no return type

    bit [15:0] B : { return $urandom_range(15,0); };
endsequence
$display("\n");
end
endmodule
```

In this example, let us focus on productions A and B. Both return a value to the production “do1” which triggered A and B. Do not confuse this “return” with the one that aborted a production. Here you have a “return” with an expression. The “return” statement assigns the given expression to the corresponding production. In our case, production A returns a \$urandom value, while production B returns a \$urandom\_range value. The return values from both A and B are now available to “do1”

(the production that triggered A and B). We then display the values returned by A and B in production “do1.” Here is the simulation log.

*Simulation log:*

A=36 B=000c

A=7d B=0002

A=0b B=000f

A=40 B=0007

### V C S   S i m u l a t i o n   R e p o r t

As you notice, production A returned an 8-bit value generated by \$urandom. Production B returned a 16-bit value in the range 15 to 0, generated by \$urandom\_range.

If you did not specify a return type as in the following case:

```
A : { return $urandom; }; //Compile Error - no return type
```

you will get the following compile error (Synopsys – VCS):

Error-[SV-IVPC] Illegal void production call  
testbench.sv, 10

Illegal call to void production 'A' in statement '\$write("A=%h B=%h", A,  
B);'.

Here is another example:

```
module TOP ( );
initial begin
repeat (2) begin
randsequence( vop ) //top production

vop : value operator value
    //note 'value' is an implicit array: value[1:2]
{$display("value[1]=%b value[2]=%b \n",
value[1],value[2]);};

bit [7:0] value : { return $urandom; } ;

operator : add dec mult {$display("add=%s dec=%s mult=%s",
add, dec, mult);};

string add : { return "addition" ; };
string dec : { return "decrement" ; };

```

```
    string mult : { return "multiplication" ; };
endsequence
end
end
endmodule
```

In this example, the key statement to note is the following:

```
vop : value operator value
//note: 'value' is an implicit array value[1:2]
```

Here production “value” appears twice in the production “vop.” If a production appears only once in a rule, the type of the implicit variable is the return type of the production. If a production appears multiple times, the type is an array where the element type is the return type of the production. So, there is an implicit declaration of “value” as follows:

```
bit [7:0] value [1:2];
```

The array is indexed from 1 to the number of times the production appears within the rule. The elements of the array are assigned the values returned by the instances of the production according to the syntactic order of appearance.

*Simulation log:*

```
add=addition dec=decrement mult=multiplication
value[1]=00110110 value[2]=00111100
```

```
add=addition dec=decrement mult=multiplication
value[1]=01111101 value[2]=11100010
```

As you notice from the simulation log, there are two array elements of “value,” and each one gets the return value of \$random. The production “operator” is further divided into its component productions “add,” “dec,” and “mult,” each of which gets a return value of string type.

# Chapter 14

## SystemVerilog Assertions



**Introduction** This chapter explores SystemVerilog Assertions (SVA). It discusses SVA methodology, immediate/deferred assertions, concurrent assertions and its operators, property, sequence, multi-threading, “bind” properties, sampled value functions, global clocking past/future functions, abort properties, multiclock assertions, etc.

As is well known in the industry, the design complexity at 5nm node and below is exploding. Small form factor requirements and conflicting demands of high performance and low power and small area result in ever so complex design architecture. Multi-core, multi-threading, and power, performance, and area (PPA) demands exacerbate the design complexity and functional verification thereof.

The burden lies on functional and temporal domain verification to make sure that the design adheres to the specification. Not only is RTL (and Virtual Platform) functional verification important but so is silicon validation. Days when engineering teams would take months to validate the silicon in the lab are over. What can you do during pre-silicon verification to guarantee post-silicon validation a first pass success?

Note that the verification complexity applies to both ASIC designs and FPGA designs. Specifically, FPGA designs are essentially SoC designs with multiple well-placed and routed cores in the design. The days of burn and learn strategy employed by FPGA design and verification engineers are over. In burn, the FPGA design and debug in the lab require that the FPGA design is ready (to some extent) *before* you burn the FPGA. If the FPGA design was not well verified, then the debug time in lab increases exponentially. This is the reason a robust verification methodology is essential for FPGA designs as well.

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_14](https://doi.org/10.1007/978-3-030-71319-5_14)) contains supplementary material, which is available to authorized users.

## 14.1 SystemVerilog Assertions Evolution

In earlier days of SystemVerilog, there was no temporal (sequential) domain checking support in the language. One had to link an “external” language with SystemVerilog (think PSL), creating a mixed-language environment. One had to maintain two simulators tied to specific EDA vendors. So, the powers- to-be in the SystemVerilog standards committee added a distinct subset to the language and called it SystemVerilog Assertions (SVA). SystemVerilog Assertions added much-needed support for a dedicated language to support comprehensive temporal domain checking. Note that SystemVerilog Assertions is orthogonal to SystemVerilog language. In other words, the syntax of SVA is totally different from that of SystemVerilog and SystemVerilog Functional Coverage sub-languages. But all these languages simulate unified in a simulation time tick. We will see how SVA executes in the simulation kernel in a simulation time tick in the upcoming sections.

SVA is derived and influenced by many different disparate languages. Figure 14.1 shows the contributing languages to the evolution of SVA.

SystemVerilog Assertions language is derived from many different languages. Features from these languages either influenced the language or were directly used as part of the language syntax/semantic.

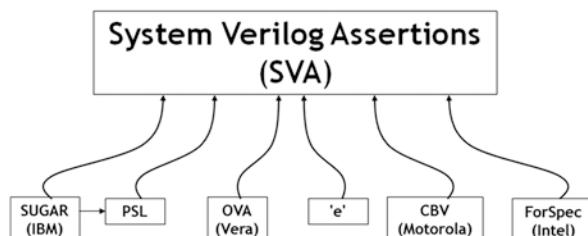
Sugar from IBM led to PSL. Both contributed to SVA. The other languages that contributed are Vera, “e,” CBV from Motorola, and ForSpec from Intel.

In short, when we use SystemVerilog Assertions language, we have the benefit of using the latest evolution of an assertion language that benefited from many other robust assertion languages.

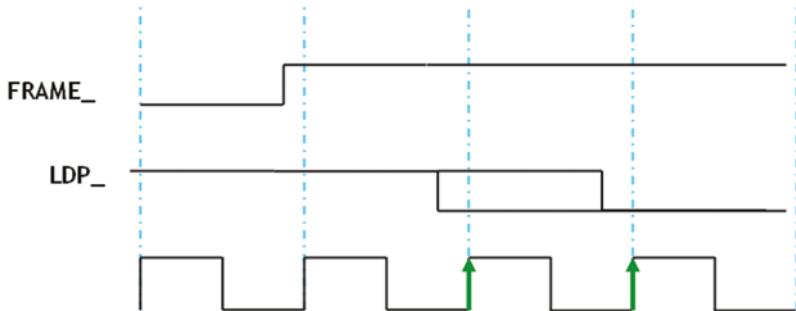
## 14.2 What Is an Assertion?

An assertion is simply a check against the specification of your design that you want to make sure never violates. If the specs are violated, you want to see a failure.

A simple example is given in Fig. 14.2. Whenever FRAME\_ is de-asserted (i.e., goes high), the Last Data Phase (LDP\_) must be asserted (i.e., goes low) within the next two clocks. Such checks are imperative to correct functioning of a given interface.



**Fig. 14.1** SystemVerilog Assertions evolution



*When **FRAME\_** is de-asserted (high), **LDP\_** (last data phase) (low) must be asserted within the next 2 clocks*

```
property ldpcheck;
  @ (posedge clk) $rose (FRAME_) |-> ## [1:2] $fell (LDP_);
endproperty
ap: assert property (ldpcheck) else $display ("ldpcheck FAIL");
cp: cover property (ldpcheck) $display ("ldpcheck PASS");
```

**Fig. 14.2** A simple bus protocol design and its SVA property

There is the property “ldpcheck” that says “at posedge clock, if FRAME<sub>\_</sub> rises, it implies that within the next 2 clocks LDP<sub>\_</sub> falls.” SVA language is precisely designed to tackle such sequential/temporal domain scenarios. As we will see in Sect. 14.3.1, modeling such a check is far easier in SVA than in SystemVerilog. Note also that assertions work in temporal domain (and we will cover a lot more on this later) and are concurrent as well as multi-threaded. These attributes are what makes SVA language so suitable for writing temporal domain checks.

Figure 14.2 shows the assertion for this simple bus protocol. We will discuss how to read this code and how this code compares with Verilog in the immediately following Sect. 14.3.1.

## 14.3 Why Assertions? What Are the Advantages?

As we discussed in the introductory section, we need to increase the productivity of the design/simulate/debug/cover loop. Assertions help exactly in these areas. As we will see, they are easier to write than standard Verilog or SystemVerilog (thereby increasing design productivity), easier to debug (thereby increasing debug productivity), provide functional coverage (thereby supporting coverage), and simulate faster compared to the same assertion written in Verilog or SystemVerilog. Let us see these advantages one by one.

### 14.3.1 Assertions Shorten Time to Develop

Referring to the timing diagram in Fig. 14.2, let us see how SVA shortens time to develop. The SVA property that we just discussed, “ldpcheck,” is written saying “at posedge clock, if FRAME\_ rises, it *implies* that within the next 2 clocks LDP\_ falls.” This is almost like writing the checker in English. We then “assert” this property, which will check for the required condition to meet at every posedge clk. We also “cover” this property to see that we have indeed exercised the required condition. But we are getting ahead of ourselves. All this will be explained in detail in the coming sections. For now, simply understand that the SV assertion is easy to write, easy to read, and easy to debug.

Now examine the Verilog code for the same check (Fig. 14.3). There are many ways to write this code. One of the ways at the behavioral level is shown. Here you “fork” off two procedural blocks: one that monitors LDP\_ and another that waits for two clocks. You then disable the entire block (“ldpcheck”) when either of the two procedural blocks complete. As you can see that not only is the checker hard to read/interpret but also very error prone. You may end up spending more time writing/debugging your checker than the logic under verification.

#### Verilog Code

```
always @(posedge FRAME_)
begin:ldpcheck
  @(posedge clk);
  fork
    begin
      @(negedge LDP_) disable ldpcheck;
    end
    begin
      repeat (2) @(posedge clk); $display("ldpcheck FAIL");
      disable ldpcheck;
    end
  join
end
```

**Fig. 14.3** Verilog code for the simple bus protocol

### 14.3.2 Assertions Improve Observability

One of the most important advantages of assertions is that they fire at the source of the problem (Fig. 14.4). As we will see in the coming sections, assertions are located local to logic in your design. In other words, you do not have to back trace a bug all the way from primary output to somewhere internal to the design where the bug originated. Assertions are written such that they are close to logic (e.g., @ (posedge clk) state0 |-> Read); . Such an assertion is sitting close to the state machine, and if the assertion fails, we know that when the state machine was in state0 that read did not take place. Some of the most useful places to place assertions are FIFOs, counters, block-to-block interface, block to/from System IO interface, state machines, system-level functional paths, etc. These constructs are where many of the bugs originate. Placing an assertion that check for local condition will fire when that local condition fails, thereby directly pointing to the source of the bug. This can be called black box verification with white box observability.

Traditional verification can be called black box verification with black box observability, meaning you apply vectors/transactions at the primary input of the “block” without caring for what is in the block (black box verification) and you observe the behavior of the block only at the primary outputs (black box observability). Since you do not have observability in the design under test, you basically start debugging from primary output to internal logic and with lengthy waveform-based debug you find the bug. Assertions on the other hand allow you to do black box verification with white box (internal to the block) observability.

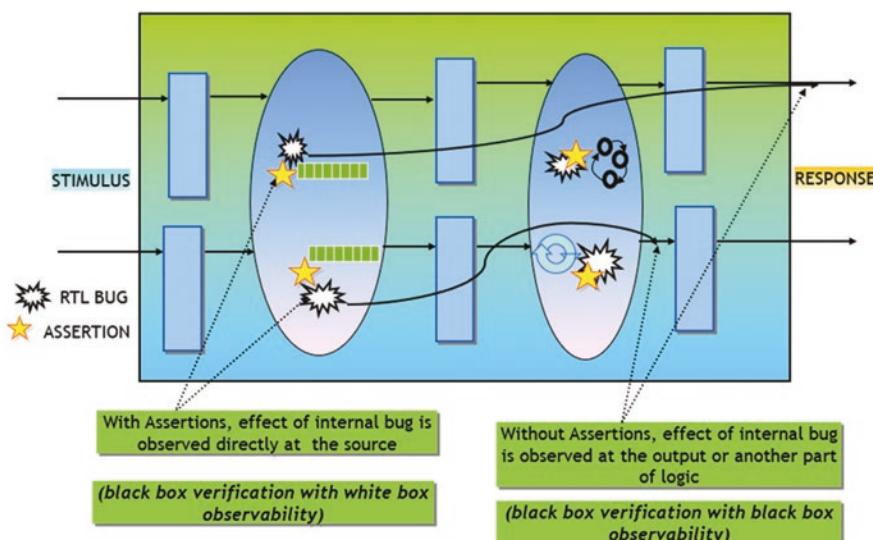


Fig. 14.4 Assertions improve observability

### 14.3.3 Other Major Benefits

- SVA language supports multiclock domain crossing (CDC) logic
  - SVA properties can be written that cross from one clock domain to another. Great for data integrity checks while crossing clock domains (think asynchronous FIFO). There is not a single ASIC/FPGA design that I know of that does not have multiple clock domains. So, this feature is of utmost importance.
- Assertions are readable: great for documenting and communicating design intent
  - Great for creating executable specs. Instead of lengthy and verbose description to convey the specs, you can write assertions to convey the same specs.
  - Process of writing assertions to specify design requirements and conducting cross design reviews identify:
    - Errors, inconsistencies, omissions, vagueness
    - Use it for design verification (test plan) review. Create an assertion plan and check against the specification that you want to verify.
- Reusability for future designs
  - Parameterized assertions (e.g., for a 16-bit bus interface) are easier to deploy with the future designs (with a 32-bit bus interface).
  - Assertions can be modeled outside of RTL and easily bound (using “bind”) to RTL keeping design and DV logic separate, easy to maintain, and reusable for the next design project.
- Assertions are always on
  - Assertions never go to sleep (until you specifically turn them off).
  - In other words, active assertions take full advantage of every new test/stimulus configuration added by monitoring design behavior against the new stimulus.
- Acceleration/emulation with assertions
  - Long latency and massive random tests need acceleration/emulation tools. These tools are beginning to support synthesizable assertions. Assertions are of great help in quick debug of long/random tests. We will discuss this further in the coming sections.
- Global severity levels (\$Error, \$Fatal, etc.)
  - Helps maintain a uniform error reporting structure in simulation
- Global turning on/off of assertions (as in \$dumpon/\$dumpoff)
  - Easier code management (no need to wrap each assertion with an on/off condition).

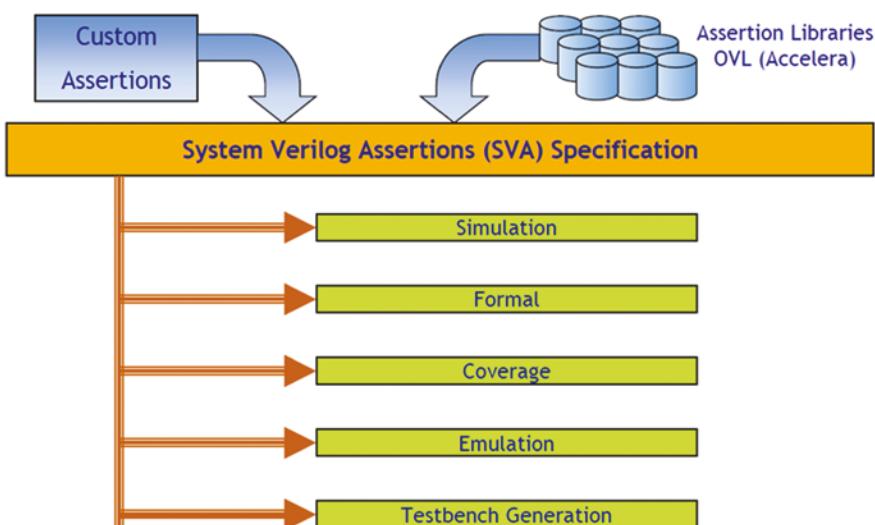
- Formal verification depends on assertions
  - The same assertions used for design simulation are also used directly by formal verification tools (with some exceptions). Static formal applies its algorithms to make sure that the assertion never fails.
  - “assume” and “restrict” allow for correct design constraint important to formal.
- One language, multiple usage
  - “assert” for design check and for formal verification
  - “cover” for temporal domain coverage check
  - “assume” and “restrict” for specifying design constraints for formal verification

#### 14.3.4 One-Time Effort, Many Benefits

Figure 14.5 shows the advantage of assertions. Write them once and use them with many tools.

We have discussed at the high level the use of assertions in simulation, formal, coverage, and emulation. But how do you use them for testbench generation/checker and what is OVL assertion library?

**Test-Bench Generation/Checker** With ever-increasing complexity of logic design, the testbenches are getting ever so complex as well. How can assertions help in



**Fig. 14.5** Assertions and OVL for different uses

designing testbench logic? Let us assume that you need to drive certain traffic to a DUT input under certain condition. You can design an assertion to check for that condition, and upon its detection, the *fail* or *pass* action block triggers, which can be used to drive traffic to the DUT. Checking for a condition is far easier with assertion language than with SystemVerilog alone. The second benefit is to place assertions on verification logic itself. Since verification logic (in some cases) is even more complex than the design logic, it makes sense to use assertions to check testbench logic also (test the testbench).

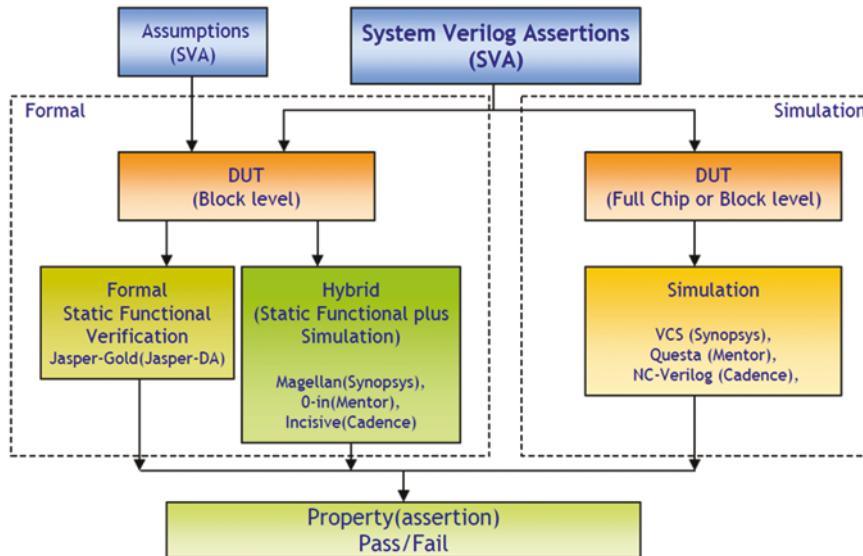
**OVL** Open Verification Library. This library of predefined checkers was written in Verilog before PSL and SVA became mainstream. Currently the library includes SVA (and PSL)-based assertions as well. The OVL of assertion checkers is intended for use by design, integration, and verification engineers to check for good/bad behavior in simulation, emulation, and formal verification. OVL contains popular assertions such as FIFO assertions, among others. OVL is still in use, and you can download the entire standard library from Accellera website: <http://www.accellera.org/downloads/standards/ovl>.

We will not go into the detail of OVL since there is plenty of information available on the net. OVL code itself is quite clear to understand. It is also a good place to see how assertions can be written for “popular” checks (e.g., FIFO) once you have better understanding of assertion semantics.

## 14.4 Assertions in Static Formal

The same assertions (with some exceptions) that you write for design verification can be used with formal/static functional verification or the so-called hybrid static functional plus simulation verification. Figure 14.6 shows (on LHS) SVA *assumptions* and (on RHS/center) SVA *assertions*. As you see, the assumptions are most useful to *static functional verification (a.k.a. formal)* (even though assumptions can indeed be used in simulation as well, as we will see in the later sections), while SVA assertions are useful in both formal and simulation.

So, what is static functional verification (also called static formal functional or simply formal)? In plain English, static formal is a method whereby the static formal algorithm applies all possible combinational and temporal domain stimulus possibilities to exercise all possible “logic cones” of a given logic block and see that the assertions are not violated. This eliminates the need for a testbench and also makes sure that the logic never fails under *any* circumstance. This provides 100% comprehensiveness to the logic under verification. So, why do we ever need to write a testbench? The static formal (as of this writing) is limited by the size of the logic block (i.e., gate equivalent RTL) especially if the temporal domain of inputs to exercise is large. The reason for this limitation is that the algorithm has to create different logic cones to try and prove that the property holds. With larger logic blocks, the



**Fig. 14.6** Assertions in formal and simulation

number of these so-called logic cones explodes. This is also known as “state space explosion” problem. To counter this problem, simulation experts came up with the *hybrid simulation* technique. In this technique, simulation is deployed to reach “closer” to the assertion logic and then employ the static functional verification algorithms to the logic under test. This reduces the scope of the number of logic cones and their size, and you may be successful in seeing that the property holds. Since static functional or hybrid is beyond the scope of this book, we will leave it at that.

## 14.5 Methodology Components

Let us discuss SVA methodology components. Then we will see an example of what type of assertions to add in a real-life bus protocol.

### 14.5.1 Types of Assertions to Add

It is important to understand and plan for the types of assertions one needs to add. Make this part of your verification plan. It will also help you partition work among your team members:

- RTL assertions (design intent)
  - Intra-module
    - Illegal state transitions, deadlocks, livelocks
    - FIFOs, onehot, etc.
- Module/block interface assertions (design interface intent)
  - Inter-module protocol verification, illegal combinations (ack cannot be “1” if req is “0”); steady-state requirements (when slave asserts write\_queue\_full, master cannot assert write\_req).
  - Good rule of thumb – *Every design assumption is an assertion.*
- Chip functionality assertions (chip/SoC functional intent)
  - A PCI transaction that results in target retry will indeed end up in the retry queue.
- Chip/system interface assertions (chip/system interface intent)
  - Commercially available standard bus assertion VIPs can be useful in comprehensive check of your design’s adherence to std. protocol such as PCIe, AXI, etc.
  - Bus interface will be in “wait” state whenever the master and slave are not ready.
- Performance implication assertions (performance intent)
  - Cache latency for read, packet processing latency, etc. to catch performance issues before it is too late. This assertion works like any other. For example, if the “read cache latency” is greater than two clocks, fire the assertion. This is an easy-to-write assertion with especially useful return.
  - I would like to put special emphasis on the “performance implication” assertions. Many miss on this point. Coming from processor background, I have seen that these assertions turn out to be some of the most useful assertions. These assertions would let us know of the, e.g., cache latency upfront and would allow us enough time to make architectural changes to fix the latency requirements.

#### **14.5.2 How to Add Assertions? What Is the Protocol?**

- Do not duplicate RTL
  - White box observability does not mean adding an assertion for each line of RTL code. This is a very important point, in that if RTL says @next clock,  $a=b+c$  does not write an assertion that says the same thing!!

- Capture the intent
  - For example, a write that follows a read to the same address in the request pipe will always be allowed to finish before the read. This is the intent of the design. How the designer implements reordering logic is not of much interest. So, from verification point of view, you need to write assertions that verify the chip design intent.
  - A note here says that the above does not mean you do not add low-level assertions. Classic example here is FIFO assertions. Write FIFO assertions for all FIFOs in your design. FIFO is low-level logic, but many of the critical bugs hang around FIFO logic, and adding these assertions will provide maximum bang for your buck.
- Add assertions throughout the RTL design process
  - They are hard to add as an after-thought.
  - Will help you catch bugs even with your simple block-level testbench.
- If an assertion did not catch a failure...
  - If the test failed and none of the assertions fired, see if there are assertions that need to be added which would fire for the failing test.
  - The newly added assertion is now active for any other test that may trigger it.

Note: This point is very important when deciding if you have added enough assertions. In other words, if the test failed and none of the assertions fired, there is a good chance you still have more assertions to add.

- Reuse
  - Create libraries of common “generic” properties with formal arguments that can be instantiated (reused) with actual arguments. We will cover this further in the chapter.
  - Reuse for the next project.

### 14.5.3 How Do I Know I Have Enough Assertions?

- It is the “Test plan, test plan, test plan...”
  - Review and re-review your test plan against the design specs.
  - Make sure you have added assertions for every “critical” function that you must guarantee works.
- If tests keep failing but assertions do not fire, you do not have enough assertions.
  - In other words, if you had to trace a bug from primary outputs (of a block or SoC) without any assertions firing, that means that you did not put enough assertions to cover that path.

- “formal” (a.k.a. static formal a.k.a. static functional verification) tool’s ability to handle assertions.
- What this means is that if you do not have enough “assertion density” (meaning if a register value does not propagate to an assertion within three to five clocks – resulting in assertions sparsely populated within design), the formal analysis tool may give up due to the state/space explosion problem. In other words, a static functional formal tool may not be able to handle a large temporal domain. If the assertion density is high, the tool has to deal with smaller cones of logic. If the assertion density is sparse, the tool has to deal with larger cones of logic in both temporal and combinatorial spaces, and it may run into trouble.

#### 14.5.4 A Simple PCI Read Example: Creating an Assertion Test Plan

Let us consider a simple example of PCI read. Given the specification in Fig. 14.7, what type of assertions would the design team add and what type would the verification team add? The tables below describe the difference. I have only given few of the assertions that could be written. There are many more assertions that need to be written by verification and design engineers. However, this example will act as a basis for differentiation.

Designers add assertions at the micro-architecture level (Table 14.2), while verification engineers concentrate at the system level (Table 14.1) specifically the interface level in this example.

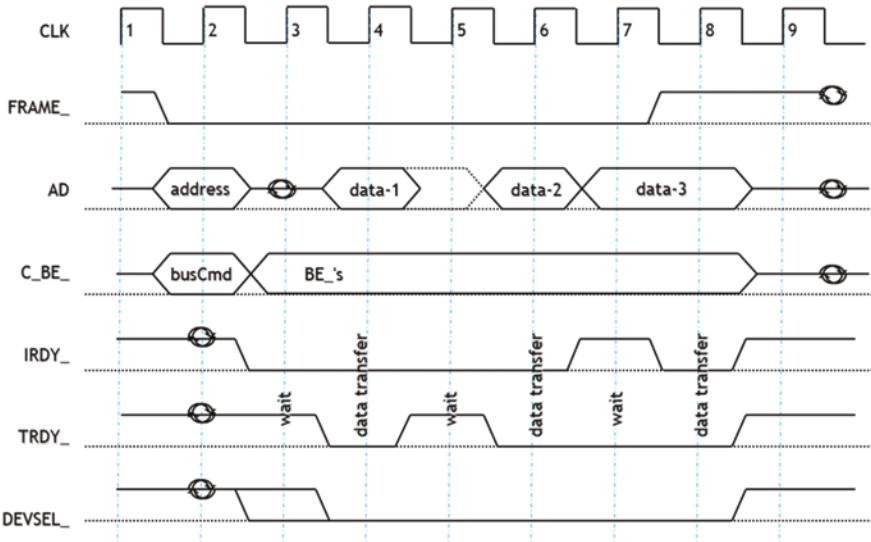


Fig. 14.7 A simple PCI read protocol

**Table 14.1** PCI read protocol test plan by functional verification team

PCI: Basic Read Protocol Test Plan – Verification Team			
Property name	Description	Property fail?	Property covered?
Protocol interface assertions			
Check PCI_AD_CBE (check1)	On falling edge of FRAME_, AD and C_BE_ bus cannot be unknown		
checkPCI_DataPhase (check2)	When both IRDY_ and TRDY_ are asserted, AD or C_BE_ bus cannot be unknown		
checkPCI_Frame_Irdy (check3)	FRAME can be de-asserted only if IRDY_ is asserted		
checkPCI_trdyDevsel (check4)	TRDY_ can be asserted only if DEVSEL_ is asserted		
checkPCI_CBE_during_trx (check5)	Once the cycle starts (i.e., at FRAME_ assertion), C_BE_ cannot float until FRAME_ is de-asserted		

**Table 14.2** PCI read protocol test plan by design team

PCI: Basic Read Protocol Test Plan – Design Team			
Property name	Description	Property fail?	Property covered?
Micro-architectural assertions			
check_pci_adrcbe_St	PCI state machine is in the “adr_cbe” state the first clock edge when FRAME_ is found asserted		
check_pci_data_St	PCI state machine is in the “data_transfer” state when both IRDY_ and TRDY_ are asserted		
check_pci_idle_St	PCI state machine is in the “idle” state when both FRAME_ and IRDY_ are de-asserted		
check_pci_wait_St	PCI state machine is in “wait” state if either IRDY_ or TRDY_ is de-asserted		

We will model the assertions for this PCI protocol later in the book under LAB6 exercise. It is too early to jump into writing assertions without knowing the basics.

The PCI protocol shown here is for a simple *read*. With FRAME\_ assertion, AD address and C\_BE\_ have valid values. IRDY\_ is asserted to indicate that the master is ready to receive data. Target transfers data with intermittent wait states. Last data transfer takes place a clock after FRAME\_ is de-asserted.

Let us see what type of assertions needs to be written by design and verification engineers.

Note that in Table 14.2, there are two columns: (1) Did the property *fail*? (2) Did the property get covered? There is no column for the property *pass*. That is because “cover” in an assertion triggers only when a property is exercised but does not fail; in other words, it passes. Hence, there is no need for a *pass* column. This “cover” column tells you that you indeed covered (exercised) the assertion and that it did not fail. When the assertion *fails*, it tells you that the assertion was exercised and that it failed.

## 14.6 Assertion Types

There are three types of assertions supported by SVA. In brief, here is their description. We will discuss them in plenty of details throughout this chapter:

- Immediate assertion
- Concurrent assertion
- Deferred immediate assertion

### Immediate Assertions

- Simple *non-temporal domain assertions* that are executed like statements in a procedural block
- Interpreted the same way as an expression in the conditional of a procedural “if” statement
- Can be specified only where a procedural statement is specified

### Concurrent Assertions

- These are *temporal domain assertions* that allow creation of complex sequences using clock (sampling edge)-based semantics.
- They are edge sensitive and not level sensitive. In other words, they must have a “sampling edge” on which it can sample the values of variables used in a sequence or a property. The sampling edge can be synchronous or asynchronous.
- A concurrent assertion is “assert,” “cover,” “ssume,” or “restrict.” We will discuss each type in the coming sections.

### Deferred Immediate Assertions

- Deferred assertions are a type of immediate assertions. Note that immediate assertions evaluate *immediately* without waiting for variables in its combinatorial expression to *settle* down. This also means that the immediate assertions are very prone to glitches as the combinatorial expression settles down and may fire multiple times. On the other hand, deferred assertions do not evaluate their sequence expression until the end of time stamp when all values have settled down (or in the reactive region of the time stamp).

If some of this does not quite make sense, that is ok. That is what the rest of the chapter will explain. Let us start with immediate assertions and understand its semantics. We then move on to concurrent assertions and lastly deferred assertions. The chapter’s main focus is on concurrent assertion because that is really the main gist of SystemVerilog Assertions language.

## 14.7 Conventions Used in This Chapter

This chapter describes the signal level-sensitive and edge-sensitive figure annotation conventions that are applied throughout the chapter.

Refer to Table 14.3. *The level-sensitive attribute of a signal is shown as a “fat” high and low symbol.* I could have drawn regular timing diagrams but saw that they look very cumbersome and does not easily convey the point. Hence, I chose the fat arrow to convey that *when the fat arrow is high, the signal was high before the clock, at the clock, and after the clock. The same applies for the fat low arrow.*

For edge-sensitive assertions, I chose the regular timing diagram to distinguish them from the level-sensitive symbol.

A high green arrow is for *pass* and a low red arrow is for *fail*.

## 14.8 Immediate Assertions

This chapter will introduce the “immediate” assertions (immediate “assert,” “cover,” “assume”) starting with a simple definition and leading to detailed nuances of its semantics and syntax. There are two types of immediate assertions, namely, immediate assertion and deferred immediate assertion. We will cover both in this section.

Immediate assertions are simple non-temporal domain assertions that are executed like statements in a procedural block. Interpret them as an expression in the condition of a procedural “if” statement. Immediate assertions can be specified only

**Table 14.3** Conventions used in this chapter

	LEVEL SENSITIVE HIGH: This symbol means that the signal is detected HIGH (level sensitive) at the clock edge noted in a timing diagram. It could have been high before the clock edge and may remain high after the clock edge. <u><i>It does NOT however mean that a ‘posedge’ is expected on this signal at the noted clock edge.</i></u>
	LEVEL SENSITIVE LOW: This symbol means that the signal is detected LOW (level sensitive) at the clock edge noted in a timing diagram. It could have been low before the clock edge and may remain low after the clock edge. <u><i>It does NOT however mean that a ‘negedge’ is expected on this signal at the noted clock edge.</i></u>
	EDGE SENSITIVE HIGH: This symbol means that a posedge is expected on this signal.
	EDGE SENSITIVE LOW: This symbol means that a negedge is expected on this signal.
	PROPERTY PASSes: This symbol means that a sequence/property match is detected here (i.e. the sequence/property <b>PASSes</b> ).
	PROPERTY FAILs: This symbol means that a sequence/property did not match here (i.e. the sequence/property <b>FAILs</b> ).

- Immediate assertion statement is a test of an expression performed when the statement is executed in a procedural code.
- The expression is non-temporal.

The 'else' clause applies to the 'assert' statement. If the 'assert' fails, the action specified with 'else' will be taken

**Immediate assertion.** Combinational only; no temporal domain sequence. If the 'assert' evaluates to true, the action specified with it is taken.

```
always @(posedge clk)
begin
  if (a)
    begin
      @(posedge d);
      → bORc : assert (b || c) $display("\n",$stime,,,"%m assert
passed\n");
      → else //This 'else' is for the 'assert'; not for the 'if (a)'
           $fatal("\n",$stime,,,"%m assert failed \n");
    end
  end
```

An optional statement label can be provided (very useful with %m display format).

For example, assuming the module name containing the assertion is 'test\_immediate', the \$display will print the following, if the assertion passes ::

40 test\_immediate.bORc assert passed.

Can use one of assertion severity level system tasks in the assertion action block. These levels are Sfatal, Serror, Swarning, Sinfo (discussed in detail later...)

Fig. 14.8 Immediate assertion: basics

where a procedural statement is specified. The evaluation is performed immediately with the values taken at that moment for the assertion condition variables. The assertion condition is non-temporal, which means its execution computes and reports the assertion results at the *same* time.

Figure 14.8 describes the basics of an immediate assertion. It is so called because it executes immediately at the time it is encountered in the procedural code. It does not wait for any temporal time (e.g., “next clock edge”) to fire itself. The assertion can be preceded by a level-sensitive or an edge-sensitive statement. In contrast, concurrent assertions only work on “sampling/clock” edge-sensitive logic and not on level-sensitive logic.

We see in Fig. 14.8 that there is an immediate assertion embedded in the procedural block. The procedural block is triggered by @ (posedge clk). The immediate assertion is triggered after @ (posedge d) and checks to see that (b || c) is true.

We need to note a couple of points here. First, the very preceding statement in this example is @ (posedge d), an edge-sensitive statement. However, it does not have to be. It can be a level-sensitive statement also or any other procedural statement. The reason I am pointing this out is that concurrent assertions can work only off a sampling “edge” and not off a level-sensitive control. Keep this in your back pocket because it will be very useful to distinguish immediate assertions from

concurrent assertions. Second, the assertion itself cannot have temporal domain sequences. In other words, an immediate assertion cannot consume “time.” It can only be combinatorial which can be executed in zero time. In other words, the assertion will be computed, and results will be available at the *same* time that the assertion was fired. If the “assert” statement evaluates to 0, X, and Z, then the assertion will be considered to *fail*, else it will be considered to *pass*.

We also see in the figure that there is (what is known as) an action block associated with *fail* or *pass* of the assertion. This is no different than the *pass/fail* logic we design for an “if...else” statement.

From syntax point of view, an immediate assertion uses only “assert” as the keyword in contrast to a concurrent assertion that requires “assert property.”

One key difference between immediate and concurrent assertions is that concurrent assertions always work off the sampled value in *prepended* region (see Sect. 14.10.2) of a simulation tick, while immediate assertions work *immediately* when they are executed (as any combinatorial expression in a procedural block) and do *not* evaluate its expression in the prepended region. Keep this thought in your back pocket for now since we have not yet discussed concurrent assertions and how assertions get evaluated in a simulation time tick. But this key difference will become important to note as you learn more about concurrent assertions.

Finally, as we discussed above, the immediate assertion works on a combinatorial expression whose variables are evaluated “immediately” at the time the expression is evaluated. These variables may transition from one logic value to another (e.g., 1 to 0 to 1) within a given simulation time tick, and the immediate assertion may get evaluated multiple times before the expression variable values “settle” down. This is why immediate assertions are also known to be “glitch” prone. This is where the “deferred immediate” assertions come into picture. We will discuss those in Sect. 14.8.1.

To complete the story, there are three types of immediate assertions:

**Immediate assert**

**Immediate assume**

**Immediate cover**

Note also that the chapter contains a lot more information on other types of assertions that can be called from a procedural block (just as you call immediate assertions). For example, you can call a “property” or a “sequence” (or “restrict” for formal verification) from a procedural block. “assume” and “cover” are a bit early to discuss. And, I have not discussed “property” and “sequence” yet. We will discuss these features in the upcoming sections.

Moving on...

Figure 14.9 points out a couple of finer points. First, do not put anything in the so-called action block (*pass* or *fail*) of the immediate assertion. Most synthesis tools simply ignore the entire immediate assertion with its action blocks (which makes sense) and with it will go your logic that (if) you were planning on putting in your design. This is rather obvious but easy to miss.

```

always @(posedge clk)
begin
  if (busAck)
    begin
      checkbusReq: assert (busReq && !reset) $display("\n",$stime,,,"%m passed\n");
    end
  else
    begin
      $fatal("\n",$stime,,,"%m failed \n");
      machineCheck = 1'b1; //DON'T PUT EXECUTABLE RTL HERE..
    end
end

```



ENTIRE 'assert' block is ignored by synthesis. So,  
DO NOT PLACE ANY EXECUTABLE RTL CODE IN THE  
ASSERT 'pass' OR 'fail' ACTION BLOCK

#### Immediate Assertion :: Illegal in non-procedural statement

```
assign arb = assert (a || b); //ILLEGAL
```



Immediate assertion cannot be used in continuous assign because that's a  
non-procedural statement. This will result in a compile time Error.

**Fig. 14.9** Immediate assertions: finer points

Note that an immediate assertion cannot be used in a continuous assignment statement because continuous assign is not a procedural block.

### 14.8.1 Deferred Immediate Assertions

Deferred immediate assertions are a type of “immediate” assertions. Recall that “immediate” assertions evaluate *immediately* without waiting for variables in its combinatorial expression to settle down. This also means that the immediate assertions are very prone to simulation glitches as the combinatorial expression settles down, and the immediate assertion may fire multiple times at the same time. On the other hand, deferred assertions do not evaluate their sequence expression until the *end of time tick* when all values have settled down (or in the reactive region of the time tick).

The syntax for deferred immediate assertion is “`assert #0`” or “`assert final`.” It is the `#0` (or “final”) that distinguishes deferred immediate assertion from the immediate assertion.

Note that there are two types of deferred immediate assertions. The difference between the two is identified by the keywords “`assert #0`” for observed deferred assertions and “`assert final`” for final deferred assertions.

For all practical purpose, I use “assert final” for deferred immediate assertion. That is because the “observed immediate,” in certain circumstances, may still be glitch prone. The difference between observed and final deferred assertions is in the extent of glitch filtering. Observed assertions filter glitches that occur in a single scheduling region set, active or reactive. In contrast, final assertions filter glitches that are created by the interaction between active and reactive regions. In that case the glitch spans both regions and would not be filtered by the observed deferred assertion.

I will leave the discussion of the differences between the two at this and focus on “assert final.” I will be using deferred immediate assertion terminology to mean “assert final,” unless I specifically talk about the observed immediate assertion.

Let us examine the following example:

```
assign not_a = !a;
always_comb begin: b1
    a1: assert (not_a != a) //immediate
    a2: assert #0 (not_a != a); //Observed Deferred immediate
    a3: assert final (not_a != a) //Final Deferred immediate
end
```

Let us examine the difference between immediate and deferred immediate assertions in this example. As soon as “a” changes, always\_comb wakes up, and both the immediate and deferred assertions fire right away. When the immediate assertion fires, the continuous assignment “not\_a = !a” may not have completed its assignment. In other words, “a” has not been inverted yet. But the immediate assertion expects an inverted “a” on “not\_a” immediately. The assertion will fail. This is why immediate assertions are known to be glitch prone.

On the other hand, the deferred assertion will wait until all expressions in the given time stamp have settled down (in other words, it was put in the deferred assertion report queue). In our case, the continuous assign would have completed its evaluation by the end of time stamp (meaning when the deferred assertion queue will be flushed), and “not\_a” would indeed be = !a. This is the value the deferred assertion will take into account when evaluating its expression. The deferred assertion will pass.

To reiterate, in a simple immediate assertion, pass and fail actions take place immediately upon assertion evaluation. In a deferred immediate assertion, the actions are delayed until later in the time step, providing some level of protection against unintended multiple executions on transient or “glitch” values.

Note that there is a limitation on the action block that a deferred immediate assertion has. The action block can only be a single subroutine (a task or a function). The requirement of a single subroutine call also implies that no begin-end block can surround the pass or fail statements, as begin is itself a statement that is a subroutine call. A subroutine argument may be passed by value as an input or passed by

reference as a ref or const ref. Actual argument expressions that are passed by value, including function calls, will be fully evaluated at the instant the deferred assertion expression is evaluated. It is also an error to pass automatic or dynamic variables as actuals to a ref or const ref formal.

For example, the following action blocks are illegal with deferred assertions because either they contain more than one statement or are not subroutine calls:

```
frameirdy: assert final (!frame_ == irdy) else begin interrupt=1; $error ("FAILure"); end //ILLEGAL
```

```
frameirdy: assert final (!frame == irdy) else begin $error("FAILure"); end //ILLEGAL
```

Following are legal.

```
frameirdy: assert final (!frame == irdy) else $error("FAILure"); //LEGAL (no begin-end)
```

```
frameirdy: assert final (!frame == irdy) $info("PASS"); else $error("FAILure"); //LEGAL
```

```
frameirdy: assert final (!frame == irdy);  
//LEGAL - no action block
```

Another feature of deferred immediate assertion to note is that it can be declared both in the procedural block and *outside* of it (recall that immediate assertion can only be declared in procedural block). For example, the following is legal for deferred immediate assertion but not for immediate assertion:

```
module (x, y, z);
    ....
    z1: assert final (x == y || z);
endmodule
```

This is equivalent to

```
module (x, y, z);
    always_comb begin
        z1: assert final (x == y || z);
    end
endmodule
```

Analogous to “assert,” we also have deferred “cover” and “assume.” Again, the idea is the same as that for immediate “cover” and “assume” that you want to use the final values of the combinatorial logic before evaluating “cover” or “assume.”

A deferred “assume” will often be useful in cases where a combinational condition is checked in a function but needs to be used as an assumption rather than a proof target by formal tools. A deferred cover is useful to avoid crediting tests for covering a condition that is only met in passing by glitch values:

```
assign a = c || d;
assign b = e || f;

always_comb begin:b1
    a1: cover (b != a) //immediate cover
    a2: cover #0 (b != a); //Observed deferred cover
    a3: cover final (b != a) //Final deferred cover
end
```

and for ‘assume’

```
always_comb begin:b1
    a1: assume (b !=a) //immediate assume
    a2: assume #0 (b != a); //deferred assume
    a3: assume final (b !=a) //deferred assume
end
```

Finally, you can disable a deferred assertion. For example:

```
always @(Nogo or go) begin : b1
    a1: assert final (Nogo) else $fatal(1, "Nogo");
if (go) begin
    disable a1;
end
end
```

## 14.9 Concurrent Assertions: Basics

This section introduces basics of concurrent assertions, namely, “sequence,” “property,” “assert,” “cover,” and “assume.” It discusses fine-grained nuances of clocking of concurrent assertions and also implication operators, multi-threaded semantics, formal arguments, “bind”ing of assertions, formal arguments, severity levels, and disable iff, among other topics.

Concurrent assertions are temporal domain assertions that allow creation of complex sequences which are based on *clock (sampling) edge* semantics. This is in

**SPEC:** At posedge clk, if cStart is High, that 'req' is high the same clock and 'gnt' is high 2 clocks later.

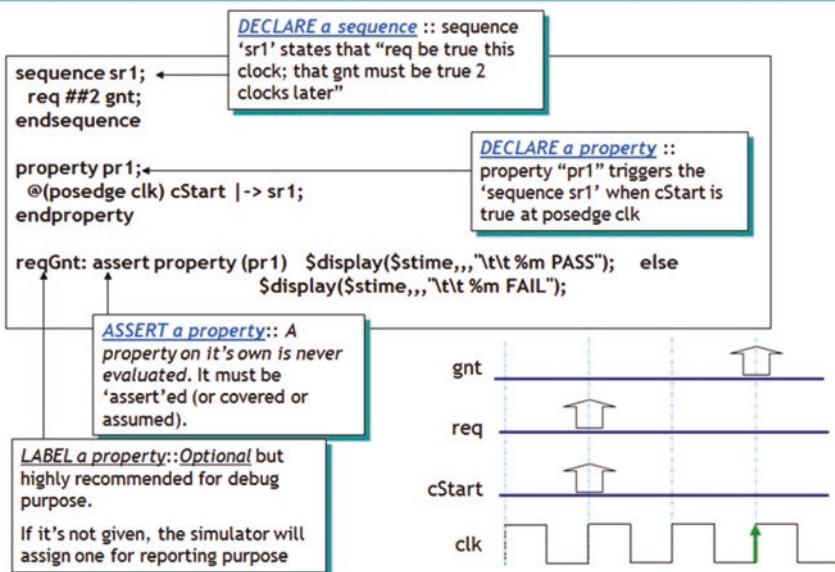


Fig. 14.10 Concurrent assertion: basics

contrast to the immediate assertions that are purely combinatorial and do not allow temporal domain sequences.

Concurrent assertions are the gist of SVA language. They are called concurrent because they execute in *parallel* with the rest of the design logic and are multi-threaded. Let us start with basics and move onto the complex concepts of concurrent assertions.

In Fig. 14.10, we have declared a property "pr1" and asserted it with a label "reqGnt" (label is optional but highly recommended). The figure explains various parts of a concurrent assertion including a property: a sequence and assertion of the property.

The "assert property (pr1)" statement triggers property "pr1." "pr1" in turn waits for the antecedent "cStart" to be true at posedge clk and on it being true *implies* (fires) a sequence called "sr1." "sr1" checks to see that "req" is high when it is fired and that two clocks later "gnt" is true. If this temporal domain condition is satisfied, then the sequence "sr1" will *pass*, and so will property "pr1" and the "assert property" will be a *pass* as well. Let us continue with this example and study other key semantics.

As explained in Fig. 14.11, the following are the basic and mandatory parts of an assertion. Each of these features will be further explored as we move along:

1. "assert" – you have to assert a property, i.e., invoke or trigger it.
2. There is an action block associated with either the pass or fail of the assertion.

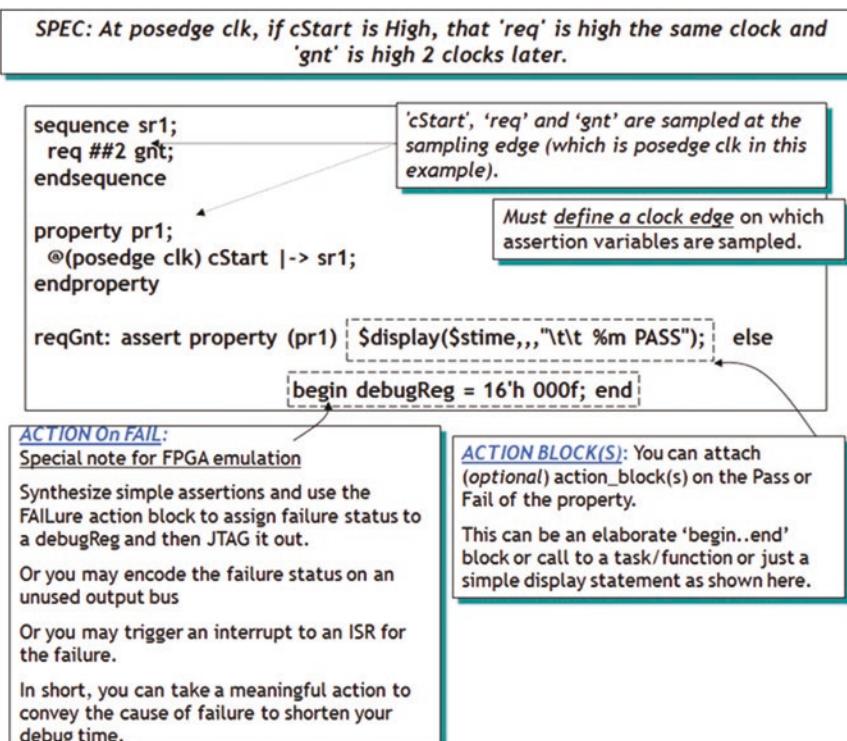


Fig. 14.11 Concurrent assertion: – sampling edge and action blocks

3. “property pr1” is edge triggered on posedge of clk (more on the fact that you *must* have a sampling edge to sample values is explained further on).
4. “property pr1” has an *antecedent* which is the signal “cStart,” which if sampled high (in the preponed region) on the posedge clk, will imply that the *consequent* (sequence sr1) be executed. More on preponed region coming up shortly.
5. Sequence sr1 samples “req” to see if it is sampled high the same posedge of clk when the sequence was triggered because of the *overlapping implication* operator and then waits for two clocks and sees if “gnt” is high.
6. Note that each of “cStart,” “req,” and “gnt” are *sampling* at the edge specified in the property which is the posedge of “clk.” In other words, even though there is no edge specified in the sequence, the edge is inherited from property pr1.

Note also that we are using the notion of sampling the values at posedge clk which means that the “posedge clk” is the *sampling edge*. In other words, the sampling edge can be anything (as long as it’s an edge and not a level), meaning it does not necessarily have to be a synchronous edge such as a clock. It can be an asynchronous edge as well. However, *be very careful about using an asynchronous edge* unless you are sure what you want to achieve. It is too soon to get into that. This is a very important concept in concurrent assertions and should be well understood.

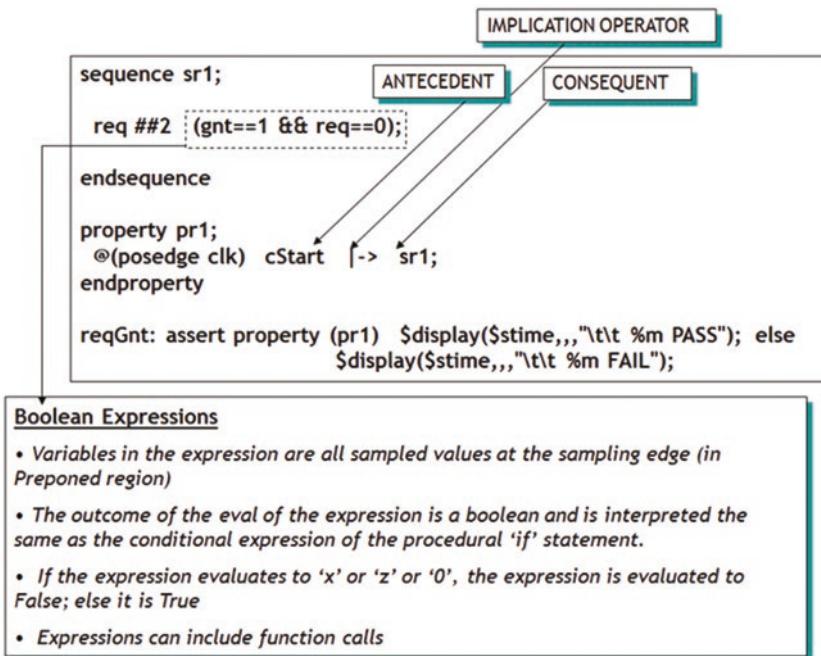


Fig. 14.12 Concurrent assertion: implication, antecedent, and consequent

Now, let us slightly modify the sequence “sr1” to highlight Boolean expression in a sequence or a property and study some more key elements of a concurrent assertion.

As shown in Fig. 14.12, there are three main parts of the expression that determines when an assertion will fire, what it will do once fired, and time duration between the firing event and execution event.

The condition, based on which an assertion will fire, is called an *antecedent*. This is the LHS of the implication operator.

RHS of the assertion that executes once the antecedent fires/matches is called the *consequent*.

The way to “read” the implication operator is “if there is a match on the antecedent that the consequent will be executed.” If there is no match, consequent will not fire, and the assertion will continue to wait for a match on the antecedent. The *implication* operator also determines the time duration that will lapse between the antecedent match and the consequent execution. In other words, the implication operator ties the antecedent and the consequent in one of two ways. It ties them with an *overlapping* implication operator or a *non-overlapping* implication operator. More on this coming up...

There’s one additional note on Boolean expressions before we move on. The following types are not allowed for the variables used in a Boolean expression.

- Dynamic arrays
- Class
- String
- Event
- Real, shortreal, realtime
- Associative arrays
- Chandle

Figure 14.13 further explains the antecedent and consequent. As shown, you do not have to have a sequence in order to model a property. If the logic to execute in consequent is simple enough, then it can be declared directly in consequent as shown. But please note that *it is always best to break down a property into smaller sequences to model complex properties/sequences*. Hence, consider this example only as describing the semantics of the language. Practice should be to divide and conquer. You will see many examples, which seem very complex to start with, but once you break them down into smaller chunks of logic and model them with smaller sequences, tying all those together will be much easier than writing one long complex assertion sequence.

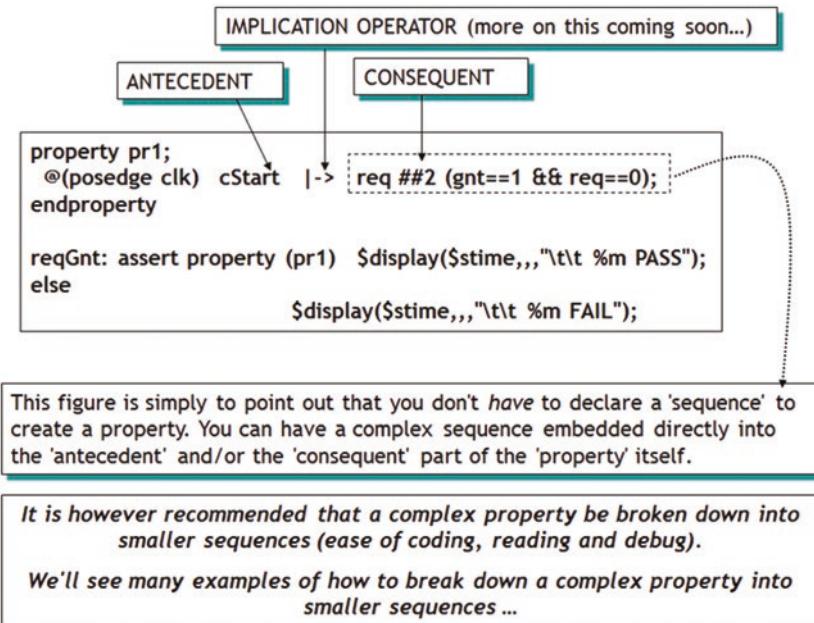


Fig. 14.13 Property with an embedded sequence

### 14.9.1 Implication Operator

Implication operator ties the antecedent and consequent. If antecedent holds true, it implies that the consequent should hold true.

There are two types of implication operators as shown in Fig. 14.14:

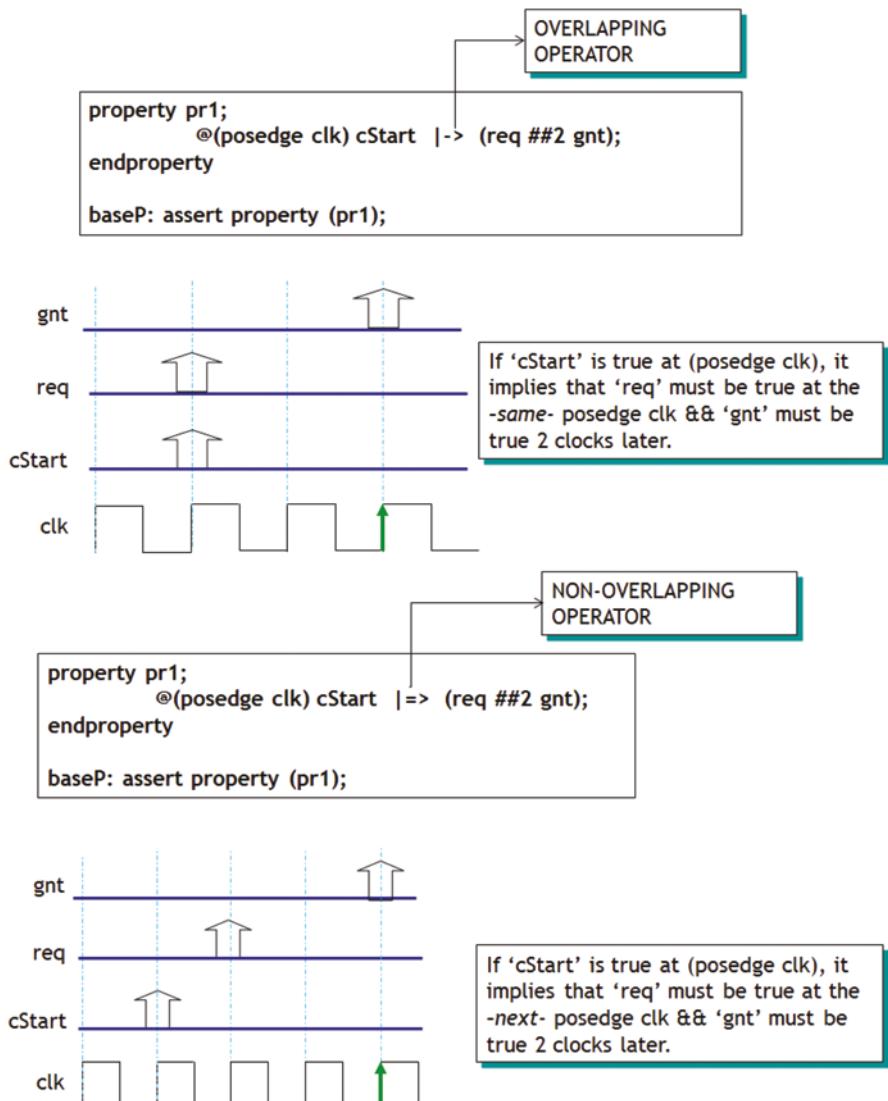


Fig. 14.14 Implication operator: overlapping and non-overlapping

**1. Overlapping implication operator:** Referring to Fig. 14.14, the topmost property shows the use of an overlapping operator. Note its symbol ( $| ->$ ), which differs from that of the non-overlapping operator ( $| =>$ ). Overlapping means that when the antecedent is found to be true, the consequent will start its execution (evaluation) at the *same* clk. As shown in the figure, when cStart is sampled high at posedge of clk, the req is required to be high at the *same* posedge clk. This is shown in the timing diagram associated with the property:

- (a) So, what happens if “req” is sampled true at the next posedge clk after the antecedent (and false before that)? Will the overlapping property pass?

**2. Non-overlapping implication operator:** In contrast, non-overlapping means that when the antecedent is found to be true, the consequent should start its execution, *one clk later*. This is shown in the timing diagram associated with the property:

- (a) So, what happens if “req” is sampled true at the same posedge clk as the antecedent (and false after that)? Will the non-overlapping property pass?

The answer to both 1.a and 2.a is *no*.

In 1.a, the property strictly looks for “req” to be true the *same* clock when “cStart” is true. It does not care if “req” is true the next clock. So, if “req” is not true when “cStart” is true, the property will fail.

In 2.a, the property strictly looks for “req” to be true the *next* clock after “cStart” is true. It does not care if “req” is true the same clock. So, if “req” is not true one clock after “cStart” is true, the property will fail.

Figure 14.15 further shows the equivalence between overlapping and non-overlapping operators. “ $|=>$ ” is equivalent to “ $|-> \#1$ .” Note that  $\#1$  is not the same as Verilog’s  $\#1$  delay.  $\#1$  means delay of one clock edge (sampling edge). Hence “ $|-> \#1$ ” means the same as “ $|=>$ .”

**Suggestion** To make debugging easier and have project-wide uniformity, use the overlapping operator in your assertions. Reason? Overlapping is the common denominator of the two types of operator. You can always model non-overlapping from overlapping, but you cannot do vice versa. What this means is that during debug everyone would know that all the properties are modeled using overlapping and that the number (#) of clocks are exactly the same as specified in the property. You do not have to add or subtract from the number (#) of clocks specified in the chip specification. More important, if everyone uses his or her favorite operator, debugging would be very messy not knowing which property uses which operator.

Finally, do note that concurrent assertions can be placed:

1. In “always” procedural block
2. In “initial” procedural block
3. Standalone (static) – outside of the procedural block – which is what we have seen so far

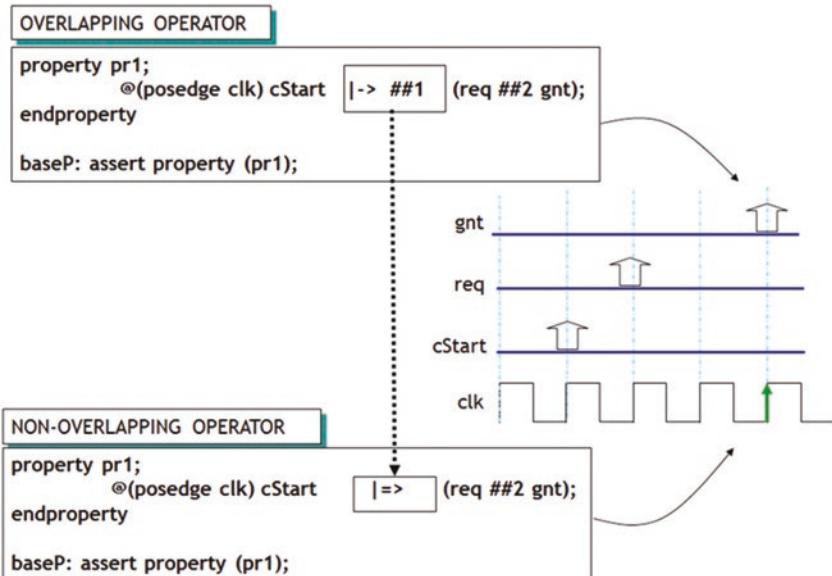


Fig. 14.15 Equivalence between overlapping and non-overlapping implication operators

## 14.10 Clocking Basics

As mentioned before, a concurrent assertion is evaluated only on the occurrence of an “edge,” known as the “sampling edge.” Most often it is a synchronous edge that you may be using. But you can indeed have an asynchronous edge as well. *But be very careful on asynchronous sampling edge, especially when the antecedent and the consequent use the same variable.* In Fig. 14.16, we are using a non-overlapping implication operator, which means that at a posedge of clk if cStart is high, then one clock later sr1 should be executed.

Let us revisit “sampling” of variables. The expression variables cStart, req, and gnt are all sampled in the *prepended region* (see Sect. 14.10.2) of posedge clk. In other words, *if cStart=1 and posedge clk change at the same time, the sampled value of cStart in the “prepended region” will be equal to “0” and not “1.”* We will soon discuss what “prepended region” really means and how it affects the evaluation of an assertion, especially when the sampling edge and the sampled variable change at the same time.

Note again that “sequence sr1” does not have a clock in its expression. The clock for “sequence sr1” is inherited from the “property pr1.” This is explained next using Fig. 14.17.

As explained in Fig. 14.17, the “clk” as an edge can be specified either directly in the assert statement or in the property or in the sequence. *Regardless of where it is declared, it will be inherited by the entire assertion* (i.e., the assert, property, and sequence blocks).

```

sequence sr1;
  req ##2 gnt;
endsequence

property pr1;
  @(posedge clk) cStart |=> sr1;
endproperty

reqGnt: assert property (pr1) $display($stime,,,"t\%m PASS"); else
  $display($stime,,,"t\%m FAIL");

```

'cStart' 'req' and 'gnt' are sampled at the sampling edge (which is posedge clk in this example).

Must define a clock edge on which assertion variables are sampled.

#### :: CLOCKING BASICS ::

- A concurrent assertion is evaluated only at the occurrence of a clock tick.
- The definition of a clock is explicitly specified by the user.
- Assertion without a clock (or a sampling edge) will result in a compile Error.
- The clock expression can be more complex than just a single signal name. E.g., you can have (CLK && Gating\_signal).

**Fig. 14.16** Clocking basics

#### Suggestion

As noted in Fig. 14.17, my recommendation is to specify the “clk” in a property. Reason being you can keep sequences void of sampling edge (i.e., “clk”) and thus make them reusable. The sampling edge can change in the property, but sequence (or cascaded sequences) remain untouched and can change their logic without worrying about the sampling edge. Note that it is also more readable when the sampling edge “clk” is declared in a property.

Note that a clock can be contextually inferred from a procedural block for a concurrent assertion. For example:

```
always @(posedge clk) assert property (not (FRAME_ ##2 IRDY));
```

Here the concurrent assertion is fired from a procedural block. The clock “@ (posedge clk)” is inferred from the “**always @(posedge clk)**” statement.

Finally, note that a named event can also act as a clock. For example:

```

module eventtrig;
  event e;
  always @(posedge clk) -> e;
  a1: assert property (@e a |=> b);
endmodule

```

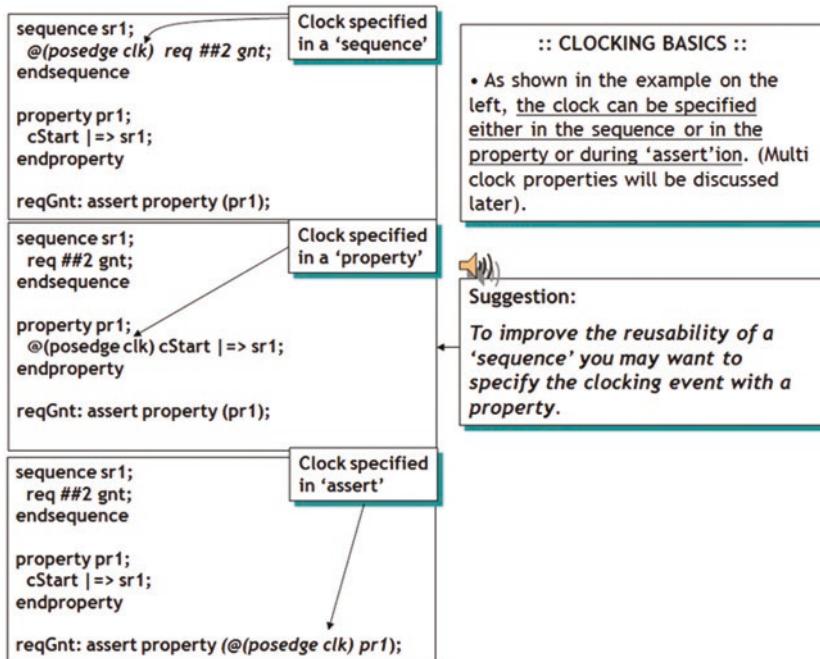


Fig. 14.17 Clocking basics: clock in “assert,” “property,” and “sequence”

For most part, concurrent assertions are triggered from outside of a procedural block.

### 14.10.1 Default Clocking Block

For a long chain of properties and sequences in the file, you can also use the default clocking block as explained in Fig. 14.18. The figure explains the different ways in which clocking block can be declared and the scope in which it is effective.

The scope of a default clocking declaration is the entire module, interface, program, or checker in which it appears, including nested declarations of modules, interfaces, or checkers. A nested module, interface, or checker may, however, have its own default clocking declaration, which overrides a default from outside.

The scope of a default clocking declaration does not descend into instances of modules, interfaces, or checkers.

“Clocking” is a keyword . The top block of the figure shows declaration of “*default clocking cb1*” which is then inherited by the properties “checkReqGnt” and “checkBusGrant” that follow. This default clocking block will be in effect until another default clocking block is defined. The bottom part of the figure is

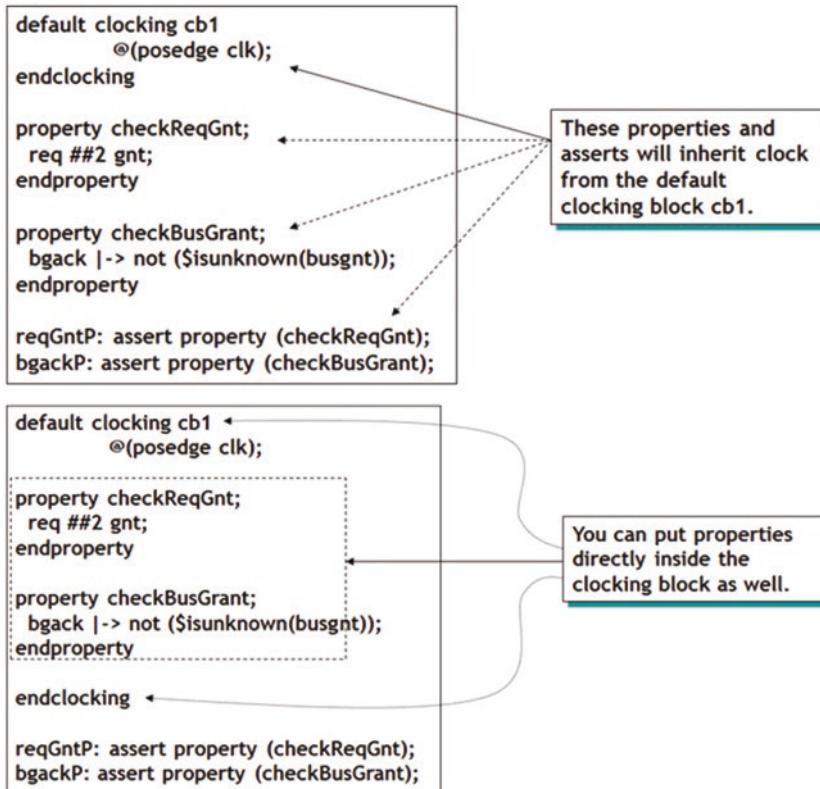


Fig. 14.18 Default clocking block

interesting. Here the properties are directly embedded in the default clocking block. I do not recommend doing that though. The clocking block should only contain clock specifications, which will keep it modular and reusable. Use your judgment wisely on such issues.

Figure 14.19 declares two clocking blocks, namely, “cb1” and “cb2,” in a stand-alone Verilog module called “design\_clocks.” This is a great way to organize your clocking strategy in one module. Once defined, you can use any of the clocking block that is required simply by referring to it by its hierarchical instance name as shown in the figure.

Here is some food for thought. I have outlined a couple of pros and cons of using a default clocking block. It is mostly advantageous but there are some caveats.

**Pros:** The argument toward default block is reusability. You may change the clocking relation in the default block, and it will be applicable to all the following blocks. You do not have to individually change clocking scheme in each property. This is indeed a true advantage, and if you plan to change the clocking scheme in the default block without affecting the properties that follow, do use the default block by all means.

```

module top;
  design_clocks design_clocks();
endmodule

module design_clocks;
bit clk;
clocking cb1
  @(posedge PCIe_clk);
endclocking

clocking cb2
  @(posedge AXI_clk);
endclocking

endmodule

module busModule (input logic req, gnt, bgack, busgnt, clk);
default clocking top.design_clocks.cb1; // Declare a 'clocking' block and
                                         // use it as 'default'

property checkReqGnt;
  req ##2 gnt;
endproperty

property checkBusGrant;
  bgack |> not ($isunknown(busgnt));
endproperty

reqGntP: assert property (checkReqGnt);
bgackP: assert property (checkBusGrant);

endmodule

```

Declare a 'clocking' block and  
use it as 'default'

Fig. 14.19 “Clocking” and “default clocking”

**Cons:** Readability/debuggability: When you see a property without any sampling edge, you have to scroll back to “someplace” to see what sampling edge is being used. You have to find the very preceding clocking block (in case of default clocking block in the same file as the property) and can’t just go to the top of the file. I like properties that are mostly self-contained with the sampling edge. Sure, it is a bit more typing but a lot more readable.

Here is an example of how you can use the “default” clocking block but also override it with explicit (non-default) clocking:

```

module default_explicit_clocking;

  default clocking negedgeClock @ (negedge clk1); endclocking
  clocking posedgeClock @ (posedge clk2); endclocking

  //will inherit default clock - negedgeClock
  d2: assert property (x |=> y);

  //will inherit default clock - negedgeClock
  d3: assert property (z [=2] |-> a);

  //will use non-default clocking posedgeClock

```

```

nd1: assert property (@posedgeClock b |=> c);

endmodule

```

Obviously, you do not *have* to declare the “clocking” block as shown above. You can simply use @ (posedge clk2) directly in the property assertion, as shown below:

```

module default_explicit_clocking;

default clocking negedgeClock @ (negedge clk1); endclocking

//will inherit default clock - negedge clk1
d2: assert property (x |=> y);

//will inherit default clock - negedge clk1
d3: assert property (z [=2] |-> a);

//explicit declaration of clock - posedge clk2
nd1: assert property (@(posedge clk2) b |=> c);

endmodule

```

Or you can model the same clocking structure as follows, using a property with its own explicit clock:

```

module default_explicit_clocking;

default clocking negedgeClock @ (negedge clk1); endclocking

property nClk; @ (posedge clk2) b |=> c; endproperty

//will inherit default clock - negedge clk1
d2: assert property (x |=> y);

//will inherit default clock - negedge clk1
d3: assert property (z [=2] |-> a);

//explicit declaration of clock - posedge clk2
nd1: assert property (nClk);

endmodule

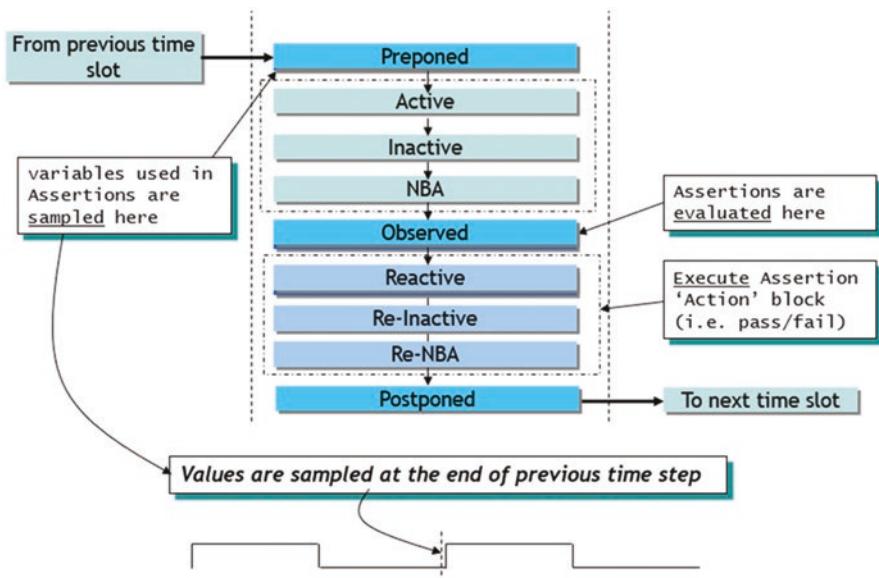
```

Note the following rules that apply to a clocking block:

1. Multiclocked sequences and properties (Sect. 14.20) are not allowed within the clocking block.
2. If a named sequence or property that is declared outside the clocking block is instantiated within the clocking block, the instance is singly clocked, and its clocking event is identical to that of the clocking block.
3. An explicitly specified leading clocking event in a concurrent assertion statement supersedes a default clocking event.

### 14.10.2 Sampling Edge (Clock Edge)

Let me first give a brief description of the active, observed, and reactive region. As far as the sampling edge is concerned, the prepended region comes into picture and you need to understand it very carefully. Please refer to Fig. 14.20 for the following discussion.



**Fig. 14.20** Assertion variable sampling and evaluation/execution in a simulation time tick

### 14.10.3 Active Region

In this region, the assertion “clock ticks” are detected, and assertions are scheduled in the observed region. Events originating in observed region get scheduled into the active region (or reactive region) for execution. The design code statements are executed in the active region.

### 14.10.4 Observed Region

The observed region is meant for the evaluation of sequences, properties, and concurrent assertions. The principal function of this region is to evaluate the concurrent assertions using the values sampled in the prepended region. Signal values remain constant during the observed region. Events originated in this region get scheduled into the active and reactive regions. In general, here is what the observed region does:

- Determine match of sequences.
- Start new attempts for sequences.
- Start new attempts for assertions.
- Resume evaluation of previous attempts.
- Schedule action blocks (*not* execute them – that is done in reactive region).

### 14.10.5 Reactive Region

The reactive region executes statements from programs and checkers and assertion action blocks. Programs are intended for writing testbenches, as external environments for designs, feeding stimuli, observing design evaluation results, and building tests to exercise the design.

### 14.10.6 Preponed Region

This region is of importance in terms of understanding how the so-called sampling semantics of assertions work.

How does the so-called sampling edge sample the variables in a property or a sequence is one of the most important concepts you need to understand when designing assertions? As shown in Fig. 14.20, the important thing to note is that the variables used in assertions (property/sequence/expression) are sampled in the

*prepended* region. What does that mean? It means, for example, if a sampled variable changes the same time as the sampling edge (e.g., clk), the value of the variable will be the value it held – *before* – the clock edge:

```
@ (posedge clk) a |=> !b;
```

In the above sequence, let us say that variable “a” changes to “1” the same time that the sampling edge clock goes posedge clk (and assume “a” was “0” before it went to a “1”). Will there be a match of the antecedent “a”? No! Since “a” went from “0” to “1” the same time that clock went posedge, the sampled value of “a” at posedge clk will be “0” (from the prepended region) and not “1.” This will not cause the property to trigger because the antecedent is not evaluated to be a “1.” This will confuse you during debug. You would expect “1” to be sampled and the property triggered thereof. However, you will get just the opposite result.

This is a very important point to understand because in a simulation waveform (or for that matter with Verilog \$monitor or \$strobe), you will see a “1” on “a” with posedge clk and would not understand why the property did not fire or why it failed (or passed for that matter). *Always remember that at the sampling edge, the “previous” value (i.e., a delta before the sampling edge in the prepended region) of the sampled variable is used.* To reiterate, prepended region is a precursor to the time slot, where only sampling of the data values takes place. No value changes or events occur in this region. Effectively, sampled values of signals do not change through the time slot.

The following is to establish what is *not* sampled. As you read the book further, you will better understand what this means. For now, keep it in your back pocket.

The following is *not* sampled in the prepended region of the time tick:

- Assertion *local* variables.
- Assertion action blocks (pass and fail).
- Clocking event expression, e.g., @ (posedge clk); here clk is not sampled – but the *current* value of clock is used).
- Disable condition (variables of conditional expression) of “disable iff.”
- Actual arguments passed to “ref” or “const ref” arguments of subroutines attached to sequences.
- The sampled value of a const cast expression is defined as the current value of its argument. For example, if “a” is a variable, then the sampled value of const(a) is the current value of a. When a past or a future value of a const cast expression is referenced by a sampled value function, the current value of this expression is taken instead.
- The default sampled value of a static variable is the value assigned in its declaration, or in the absence of such an assignment, it is the default (or uninitialized) value of the corresponding type.
- The default sampled value of any other variable or net is the default value of the corresponding type. For example, the default sampled value of variable “y” of type *logic* is 1'bx.
- The default sampled value comes into picture at time “0.” After that, it will always be the value from the prepended region.

Here is a complete example including the testbench and comments that explain how sampling of variables in the prepended region affect assertion results:

```

module assert1;
  reg A, B, C, D, clk;

  property ab;
    @ (posedge clk) !A |-> B;
  endproperty

  aba: assert property (ab) else $display ($stime,,, "ab FAIL");
  abc: cover property (ab) $display($stime,,, "ab PASS");

  initial begin
    clk=0; A=0; B=0; //Note: A and B are equal to '0' at time 0.
    forever #10 clk=~clk;
  end

  initial begin
    `ifdef PASS

      /* Following sequence of events will cause property 'ab'
       * to PASS because even though A=0 and B=1 change simultaneously they
       * had settled down because of #1 before posedge clk. Hence when @
       * (posedge clk) samples A, B; A=0 and B=1 are sampled. The property
       * antecedent '!A' is evaluated to be true and at that same time
       * (overlapping operator) B==1. Hence the property passes */
      A=0;
      B=1;
      #1;
      @ (posedge clk)

      `else
        /* Following sequence of events will cause property 'ab'
         * to FAIL. Here is the story. A=0 and B=1 change at the same time as
         * posedge clk. This causes the sampled value of B to be equal to '0'
         * and not '1' because the sampling edge (posedge clk) samples the
         * variable values in the prepended region and B was equal to '0' in
         * the prepended region. Note that A was equal to '0' in the prepended
         * region because of its initialization in the 'initial' block. So,
         * now you have both 'A' and 'B' = 0. Since A is 0, !A is true, and
         * the property evaluation takes place. Property expects B==1 the
         * same time (overlapping operator) that !A is true. However, 'B's
         * sampled value is '0' and the property fails. */
    `endif
  end

```

```

@ (posedge clk)
  A=0;
  B=1;
`endif
@ (negedge clk)
$finish(2);
end
endmodule

```

Here are some detailed rules on how variables and expressions are sampled (at a clock edge). At this time, you may not understand them well. But once you better familiarize yourself with how sampled values of variables (local, automatic, normal) take place, this will be a good reference.

The definition of a sampled value of an expression is based on the definition of sampled value of a variable. The general rule for variable sampling is as follows:

- The sampled value of a variable in a time slot corresponding to time greater than 0 is the value of this variable in the *prepended* region of this time slot.
- The sampled value of a variable in a time slot corresponding to time 0 is its default sampled value.

This rule has the following exceptions:

- Sampled values of automatic variables, local variables, and active-free checker variables (Sect. 20.4 for checker variables) are their current values. However,
- When a past or a future value of an active-free checker variable is referenced by a *sampled value function* (e.g., \$past), this value is sampled in the *postponed* region of the corresponding past or future clock tick.
- When a past or a future value of an automatic variable (Sect. 2.8.2 for automatic variables) is referenced by a sampled value function, the current value of the automatic variable is taken instead.

The sampled value of an expression is defined as follows:

- The sampled value of an expression consisting of a single variable is the sampled value of this variable.
- The sampled value of the triggered event property and the sequence methods .triggered and .matched (see Sect. 14.22) is defined as the current value returned by the event property or sequence method.
- The sampled value of any other expression is defined recursively using the values of its arguments. For example, the sampled value of an expression e1 & e2, where e1 & e2 are expressions, is the bitwise AND of the *sampled* values of e1 and e2. In particular, if an expression contains a function call, to evaluate the sampled value of this expression, the function is called on the sampled values of its arguments at the time of the expression evaluation.

## 14.11 Concurrent Assertions Are Multi-threaded

This is about the most important concept you need to grasp when it comes to concurrent assertions. We all know SystemVerilog is a concurrent language, but is it multi-threaded? SVA by default is concurrent and multi-threaded.

In Fig. 14.21, we have declared the same assertion that you have seen before, namely, at posedge clk, if cStart is sampled high that sr1 will be triggered at the same posedge clk which will then look for “req” to be high at that clock and “gnt” to be sampled high two clocks later.

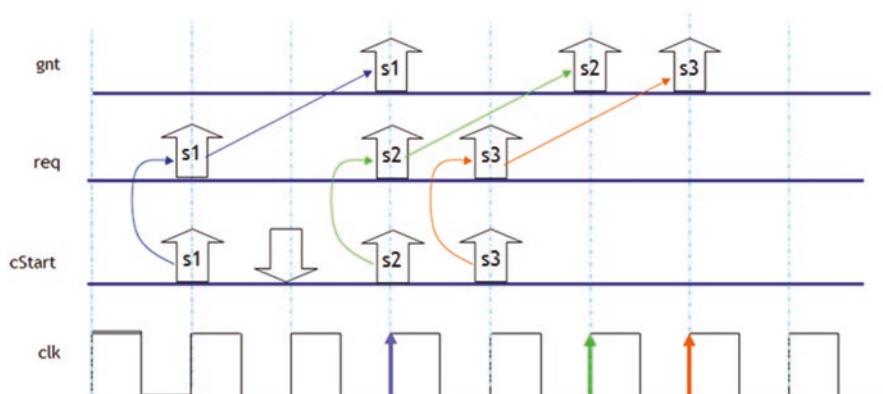
Now, let us say that cStart is *sampled* high (see S1 in waveform) at a posedge of clk and that “req” is also sampled high at that same edge. After this posedge clk, the sequence will wait for two clocks to see if “gnt” is high.

But before the two clocks are over, cStart goes low and then goes high (S2) exactly two clocks after it was sampled high. This is also the same edge when our first trigger of assertion will look for “gnt” to be high (S1). So, what will the assertion do? Will it re-fire itself because it meets its antecedent condition (S2) and ignore “gnt” that it’s been waiting for from the first trigger (S1)? Yes, it will re-fire,

```
sequence sr1;
  req #2 gnt;
endsequence

property pr1;
  @(posedge clk) cStart |-> sr1;
endproperty

reqGnt: assert property (pr1) $display($stime,,,"t\t %m PASS");
else $display($stime,,,"t\t %m FAIL");
```



**Fig. 14.21** Multi-threaded concurrent assertions

but no, it will *not* ignore “gnt.” It will sample “gnt” to be high (S1) and consider the first trigger (cStart (S1)) to *pass*. So, what happens to the second trigger (cStart (S2))? It will start *another* thread. It will again wait for two clocks to check for “gnt.” So far so good. We see one instance of SVA being threaded.

But life just got more interesting.

After S2, the very next clock cStart is sampled high again (S3). And “req” is high as well (req(S3)). Now what will the assertion do? Well, S3 will thread itself with S2. In other words, there are now two distinct threads of the same assertions waiting to sample “gnt” two clocks after their trigger. The figure perfectly (!) lines up “gnt” to be high two clocks both after S2 and after S3, and all three triggers of the same assertions will *pass*.

## 14.12 Formal Arguments

One of the key features of assertions is that they can be parameterized. In other words, assertions can be designed with formal arguments to keep them generic enough for use with different actual arguments.

Figure 14.22 explains how formal and actual arguments work. Notice that the formal arguments can be specified in a sequence and in a property.

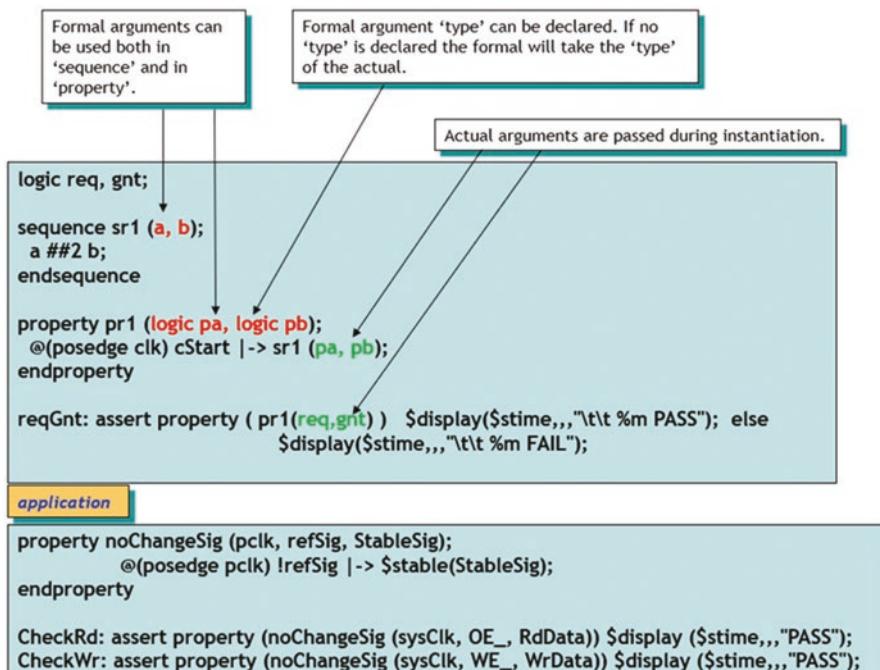
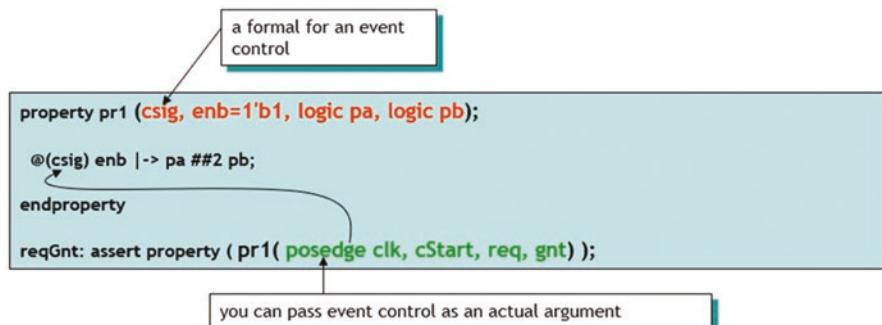


Fig. 14.22 Formal and actual arguments



**Fig. 14.23** Event control as formal argument

The application shows the advantage of formal arguments in reusability. Property “noChangeSig” has three formal arguments, namely, `pclk`, `refSig`, and `Sig`. The property checks to see that if `refSig` is sampled low at `posedge pclk`, the `Sig` is “stable.” Once such a generic property is written, you can invoke it with different `clk`, different `refSig`, and different `StableSig`. `CheckRd` is a property that uses `sys-Clk`, `OE_`, and `RdData` to check for `RdData` to be stable, while `CheckWr` uses `sys-Clk`, `WE_`, and `WrData` to check for `WrData` to be stable.

One of the powerful ways of using a formal is as an event control. You can then pass an event as an actual, thus keeping the property generic. Figure 14.23 shows such an application.

In any project, there are generic properties that can be reused multiple times by passing different actual arguments. This is reusable not only in the same project but also among projects.

Companies have created libraries of such properties that projects look up and reuse according to their needs.

## 14.13 Disable (Property) Operator: disable iff

Of course, you need a way to disable a property under conditions when the circuit is not stable (think reset). That is exactly what “disable iff” operator does. It allows you to explicitly disable the property under a given condition. Note that “disable iff” reads as “disable if and only if.” The example in Fig. 14.24 shows how you can disable an assertion during an active reset. There is a good chance you will use this reset-based disable method in all your properties throughout the project.

So, what happens if a property has started executing and the “disable iff” condition occurs in the middle of its execution?

The property in Fig. 14.24 checks to see that `sdack_falls` (i.e., contained) within `soe_` (do not worry; we will see how such properties work in later chapters). It also has the “`disable iff (! reset)`” condition. Disable this property if and only if `reset` is asserted (active low).

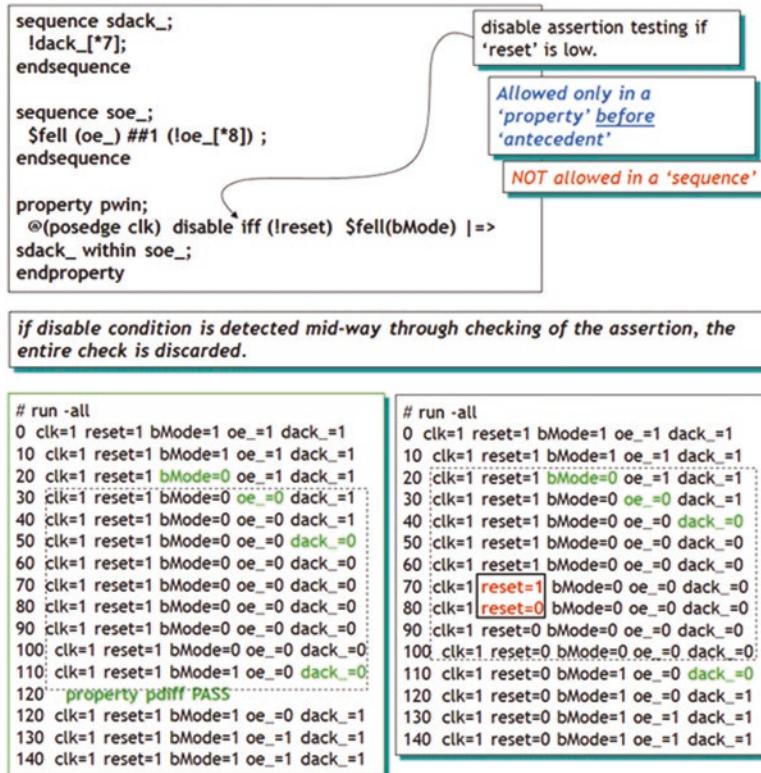


Fig. 14.24 “disable iff” operator

Let us examine the simulations logs.

In the LHS simulation log, reset is never asserted, and the assertion completes (and passes in this case).

In the RHS simulation block, reset is asserted in the middle of the check “sdack\_within soe,” and the entire assertion is discarded. You will not see pass/fail for this assertion because it has been discarded. The entire assertion is disabled if the “disable iff” condition occurs in the middle of an executing assertion. Some folks mistake such discard as a failure, which is incorrect.

Once an assertion has been disabled with “disable iff” construct, it will restart only after the “disable iff” condition is not true anymore.

Here are the rules that govern “disable iff”:

1. “disable iff” can be used only in a property – not in a sequence.
2. “disable iff” can only be used before the declaration of the antecedent condition.
3. “*disable iff* expression is not sampled at a clock edge as with other expressions in the concurrent assertion. The expressions in a disable condition are evaluated using the *current* values of variables (not sampled in prepended region). “*disable iff*” expression can be thought of as asynchronous; it can trigger in between

clock events or at clock event. This is an important point because we will be discussing a lot about sampled values, and this is an exception. In our example, we have disable iff (!reset). Here the “reset” signal is not sampled. The disable iff condition will trigger as soon as “reset” goes low.

4. Nesting of “disable iff” clauses, explicitly or through property instantiations, is not allowed.
5. The operator “disable iff” cannot be used in the declaration of a recursive property.
6. We have not discussed .triggered or .matched methods, but here is the rule for your reference. “disable iff” may contain the sequence Boolean method .triggered. But it cannot contain any reference to local variables or to the sequence method .matched.

## 14.14 Severity Levels

Assertions also allow error reporting with different severity levels: \$fatal, \$error (default), \$warning, and \$info. Figure 14.25 explains the meaning of each.

\$error is default, meaning if no failure clause is specified in the assert statement, \$error will kick in and provide a simulator-generated error message. If you have specified a label (and you should have) to the assertion, that will be (most likely) displayed in the \$error message. I say most likely because the SystemVerilog LRM

```
sequence sr1;
  req ##2 gnt;
endsequence

property pr1;
  @(posedge clk) cStart |>> sr1;
endproperty

reqGnt: assert property (pr1) else ;
```

You can also use one of the following SV system tasks in the fail statement.

**\$fatal** ← run time fatal (quit simulation)

**\$error** ← run time Error. *Default* according to SV 3.1a LRM. Vendor specific command line options may change this behavior.

**\$warning** ← run time Warning.

**\$info** ← means this assertion failure carries no specific severity.

```
reqGnt: assert property (pr1) else $fatal($stime,,, "%m Assert Fail");
```

Fig. 14.25 Severity levels for concurrent and immediate assertions

does not specify exact format of \$error. It is simulator vendor specific. \$warning and \$info are described in Fig. 14.25.

## 14.15 Binding Properties

“bind” allows us to keep design logic separate from the assertion logic. Design managers do not like to see anything in RTL that is not going to be synthesized. “bind” helps in that direction.

There are three modules in Fig. 14.26. The “designModule” contains the design. The “propertyModule” contains the assertions/properties that operate on the logic in “designModule.” And the “test\_bindProperty” module binds the propertyModule to the designModule. By doing so, we have kept the properties of the “propertyModule” separate from the “designModule.” You do not have to place properties in the same module as the design module. As mentioned before, you should keep your design void of all constructs that are non-synthesizable. In addition, keeping assertions and design in separate modules allows both the design and the DV engineers

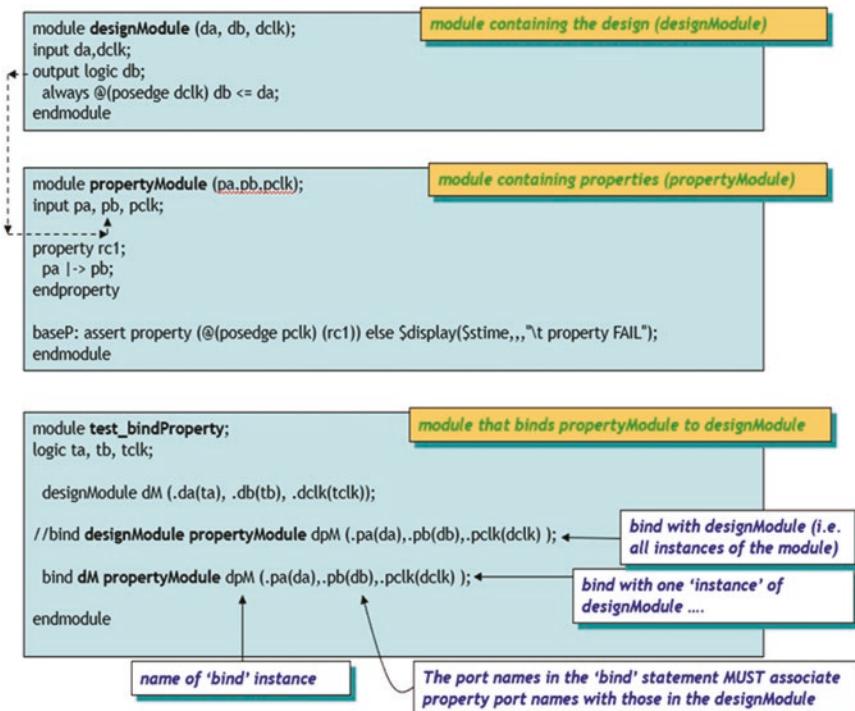


Fig. 14.26 Binding properties

work in parallel without restrictions of a database management system where a file cannot be modified by two engineers at the same time.

In order for “bind” to work, you have to declare either the instance name or the module name of the designModule in the “bind” statement. You need the design module/instance name, property module name, and the “bind” instance name for “bind” to work. In our case the design module name is designModule, its instance name is “dM,” and the property module name is propertyModule.

The (uncommented) “bind” statement uses the module instance “dM,” binds it to the property module “propertyModule,” and gives this “bind” an instance name “dpM.” It connects the ports of propertyModule with those of the designModule. With this the “property rc1” in propertyModule will act on designModule ports as connected.

The commented “bind” statement uses the module name “designModule” to bind to the “propertyModule,” whereby all instances of the “designModule” will be bound to the “propertyModule.”

In essence, we have kept the properties of the design and the logic of the design separate. This is the recommended methodology. You could achieve the same results by putting properties in the same module as the design module, but that is a highly non-modular and intrusive methodology. In addition, as noted above, keeping them separate allows both the DV and the design engineer to work in parallel.

### 14.15.1 *Binding Properties (Scope Visibility)*

But what if you want to bind the assertions of the propertyModule to internal signals of the designModule? That is quite doable.

As shown in Fig. 14.27, “rda” and “rdb” are signals internal to designModule. These are the signals that you want to use in your assertions in the “propertyModule.” Hence, you need to make “rda” and “rdb” visible to the “propertyModule.” However, you do not want to bring “designModule” internal variables to external ports in order to make them visible to the “propertyModule.” You want to keep the “designModule” completely untouched. To do that, you need to add input ports to the “propertyModule” and bind those to the internal signals of the “designModule” as shown in Fig. 14.27. Note that in our example we bind the propertyModule ports “pa” and “pb” to the designModule internal registers “rda” and “rdb.” In other words, you can directly refer to the internal signals of designModule during “bind.” “bind” has complete *scope visibility* into the bound module “designModule.” Note that with this method you do not have to provide the entire hierarchical instance name when binding to “propertyModule” input ports.

Note also that you do not have to have designModule ports “da” and “db.” “rda” and “rdb” are directly visible when you instantiate the designModule. You just need the propertyModule ports (“pa” and “pb” to bind to the designModule internal variables). In other words, the following will work too:

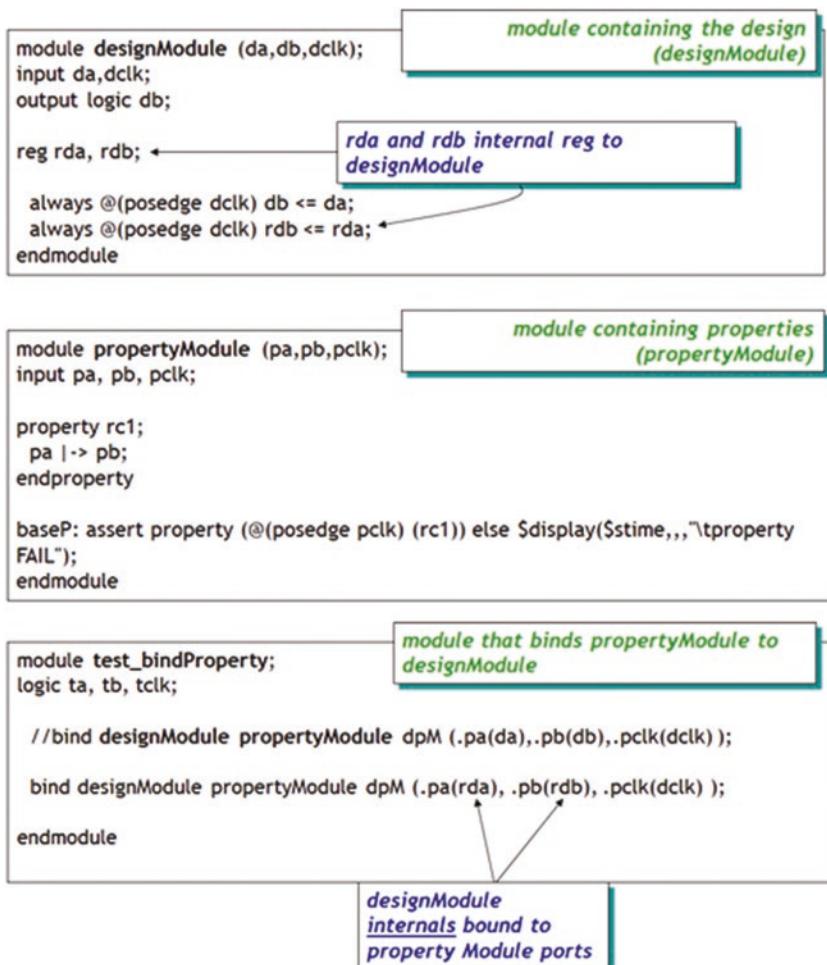


Fig. 14.27 : Binding properties to design “module” internal signals (scope visibility)

```

module designModule (dclk);
  input dclk;

  reg rda, rdb;

  always @ (posedge dclk) rdb <= rda;

endmodule

```

```

module propertyModule (pa, pb, pclk);
  input pa, pb, pclk;

  property rcl;
    pa |-> pb;
  endproperty

  baseP: assert property (@(posedge pclk) (rcl)) else
$display($stime,,,"tproperty FAIL");
endmodule

module test_bindProperty;
  logic tclk;

  designModule dM (.dclk(tclk));

  //BIND to designModule internal variables
bind dM propertyModule dpM (.pa(rda),.pb(rdb),.pclk(dclk) );
endmodule

```

### **14.15.2 VHDL DUT Binding with SystemVerilog Assertions**

Yes, binding is independent of the language. The IEEE standard does not specify exactly how to accomplish it, but EDA vendors have implemented it to suit their tools/methodology.

Here is an example showing how binding a VHDL entity to SystemVerilog Assertions module works. This is based on Mentor's Questa simulator.

The example in Fig. 14.28 shows a simple VHDL entity called “cpu” and a SystemVerilog Assertions module called “cpu\_props.” The binding takes place in SystemVerilog module called “sva\_wrapper.” The binding mechanism is identical to that we saw in previous sections. There is no difference in binding a Verilog module to a Verilog module and binding a VHDL entity to a Verilog module.

When it comes to simulation, each EDA vendor has its own way of compiling VHDL and Verilog together to accomplish the “bind.” The example shows the way Mentor’s Questa simulator accomplishes this.

<p><b>VHDL DUT</b></p> <pre> entity cpu is port ( a : in std_logic; b : in std_logic; ...);  architecture rtl of cpu is ... signal c : std_logic; begin ... end rtl; </pre>	<p><b>SVA</b></p> <pre> module cpu_props(input d,e,f); ... assert property (@(posedge d) e  -&gt; ##[1:2] f ); ... endmodule </pre>
<b>‘bind’ SVA to VHDL DUT</b>	
<pre> module sva_wrapper;  bind cpu cpu_props cpu_sva_bind (.d(a),.e(b),.f(c));     // Connect SystemVerilog ports to VHDL ports (a and b)     // and to the internal signals (c) endmodule </pre>	
<pre> vlib work vlog *.sv vcom *.vhd vsim top sva_wrapper </pre>	

Fig. 14.28 Binding VHDL DUT to SystemVerilog Assertions module

## 14.16 Difference Between “sequence” and “property”

We will see a lot more on assertions using sequences and properties, but it is good to clearly understand the differences between the two:

- **“sequence”**
  - A sequence is a building block. Think of it as a macro or a subroutine where you can define a specific relationship for a given set of signals.
  - A sequence on its own does not trigger. It must be “assert”ed or “cover”ed.
  - A named sequence may be instantiated by referencing its name. The reference may be a hierarchical name.
  - *A sequence does not allow implication operator.* It simply allows temporal (or combinatorial) domain relationship between signals.
  - A sequence can have optional formal arguments.
  - A clocking event can be used in a sequence.
  - A sequence can be declared in a module, an interface, a program, a clocking block, a package, a compilation unit scope, a checker, and a generate block (*but – not – in a “class”*).
- **“property”**
  - A property also does not trigger by itself until “assert”ed (or “cover”ed or “assume”d).

- Properties have implication operator that imply the relationship between an antecedent and a consequent.
- Sequences can be used as building blocks of complex properties.
- Clocking event can be specified in a property, in a sequence, or in both.
- The formal and actual arguments can also be “property expressions” – meaning you can pass a “property” as an actual to a formal type of “property.”
- Local variable arguments (see Sect. 14.21) can only be “local input.”
- A property can be declared in a module, an interface, a program, a clocking block, a package, a compilation unit scope, a checker, and a generate block (*but – not – in a “class”*).

## 14.17 Sampled Value Functions

This section introduces and provides applications for sampled value functions \$rose, \$fell, \$past, and \$stable. Note that there are also quite a few new sampled value functions introduced in 2009–2012 LRM (e.g., \$changed, \$rose\_gclk, \$sampled, etc.). These are covered in the later sections.

These sampled value functions (Fig. 14.29) allow for antecedent and/or the consequent to be edge triggered. \$rose means that the *least significant bit* of the

<code>\$rose (expression [, clocking event]);</code>	Returns True if the <i>least significant bit</i> of the expression changed to ‘1’ from the previous tick of the clocking event. Otherwise, it returns False.
<code>\$fell (expression [, clocking event]);</code>	Returns True if the <i>least significant bit</i> of the expression changed to ‘0’ from the previous tick of the clocking event. Otherwise, it returns False.
<p><b>Notes:</b></p> <ul style="list-style-type: none"> <li>• The <code>[, clocking event]</code> is optional and usually derived from the clocking event of the assertion or from the inferred clock of the procedural block where the function is used</li> <li>• When these functions are called at or before the simulation time step in which the first clocking event occurs, the results are computed by comparing the sampled value of the expression with its default sampled value.</li> <li>• These functions can be used in property/sequence as well as in procedural code as expressions</li> </ul>	

Fig. 14.29 Sampled value functions \$rose and \$fell: basics

expression (in \$rose(expression)) was sampled to be “0” at the previous clk edge (previous meaning the immediately preceding clk from current clk) and is sampled “1” at the current clock edge. For \$fell, just the opposite need to take place. Preceding value should be sampled “1,” and current sampled value should be “0.” As explained with examples below, one needs to understand the difference between level-sensitive sample and edge-sensitive sample.

But why do we call these functions “sampled value”? That is because they are triggered only when the sampled value of the expression in the preponed region differs at two successive clock edges as described above. In other words, \$rose(abc) does *not* mean “posedge abc” as in Verilog. \$rose(abc) does not evaluate to true as soon as abc goes from 0 to 1. \$rose(abc) simply means that abc was sampled “1” at the current clock edge (in preponed region) and that it was *not* sampled a “1” at the immediately preceding clock edge.

Note also that both \$rose and \$fell work only on the least significant bit of the expression. You will soon see what happens if you use a bus (vector) in these two sampled value functions.

#### 14.17.1 \$rose: Edge Detection in Property/Sequence

Property “checkiack” in the top logic/timing diagram will (Fig. 14.30) *pass* because both the “intr” and “iack” signals meet the required behavior of \$rose (values at two successive clks are different and are “0” followed by “1”). However, the logic in the

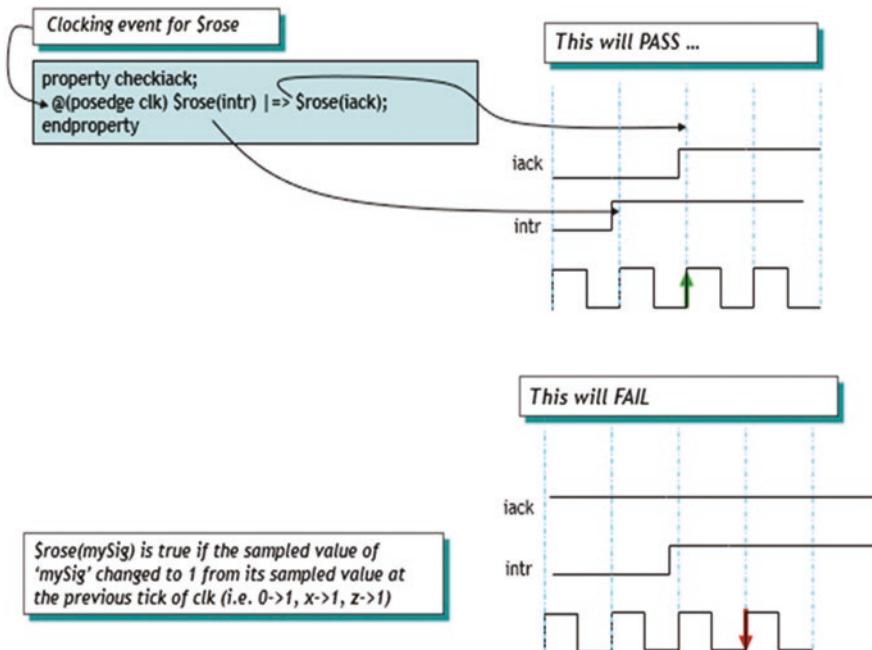


Fig. 14.30 \$rose: basics

bottom diagram fails because while \$rose(intr) meets the requirement of \$rose, “iack” does not. “iack” does not change from “0” to “1” between the two clk edges.

\$fell is explained in Fig. 14.31.

Important Note: To reiterate the points made above. \$rose does *not* mean posedge, and \$fell does *not* mean negedge (as in Verilog). In other words, the assertion will not consider \$rose(intr) to be true as soon as a posedge on “intr” is detected. The \$rose ()/\$fell () behavior is derived by “sampling” the expression at two successive clk edges and see if the values are opposite and in compliance with \$rose () or \$fell ()�

In other words, the fundamentals of concurrent assertions specify that everything must be sampled at the sampling edge in the prepended region. Behavior is based on sampled value and not the current value.

### 14.17.2 \$fell: Edge Detection in Property/Sequence

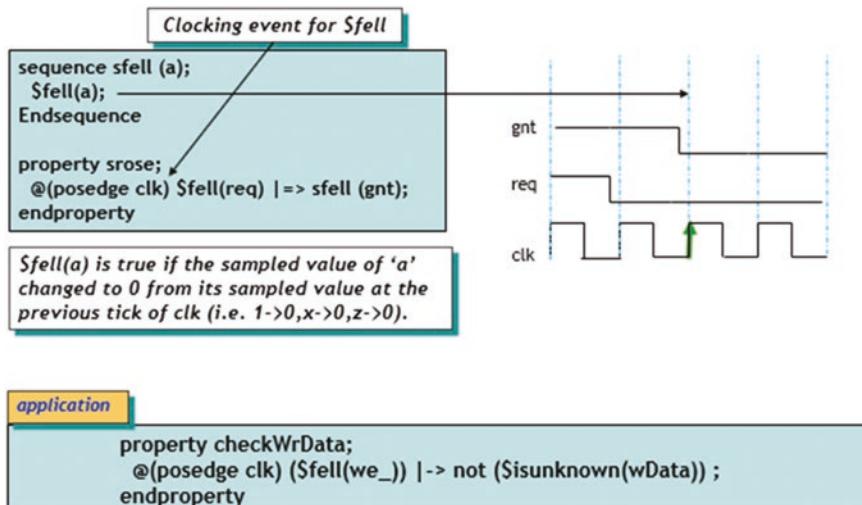


Fig. 14.31 \$fel: basics

### 14.17.3 Edge Detection Is Useful Because...

Consider the following example. It has severe performance implications that you should be aware of. Figure 14.32 shows a property where both the antecedent and consequent are level sensitive (@(posedge clk) intr |=> iack;). This means that every time “intr” is high, the consequent will fire. As long as “intr” is high, a new thread

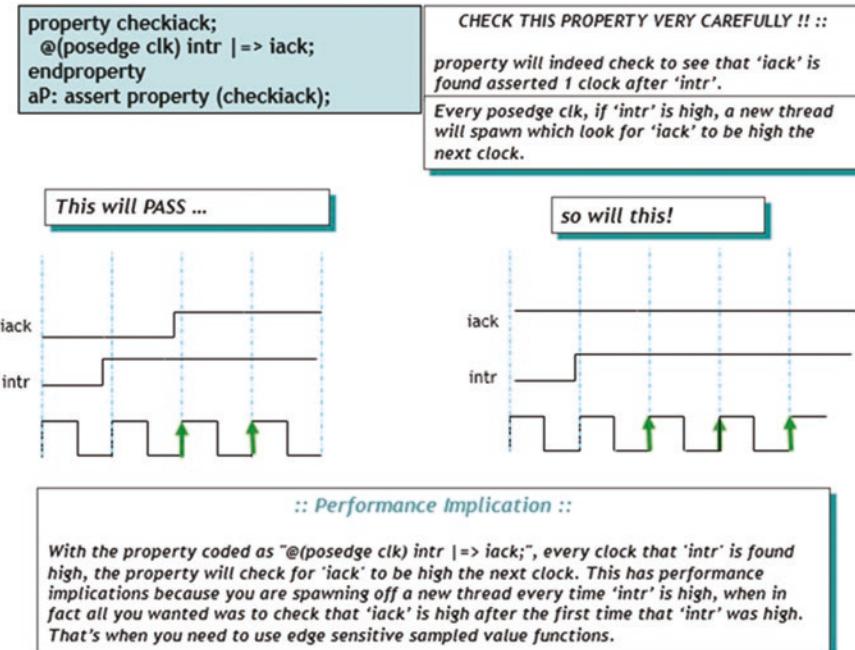


Fig. 14.32 Edge sensitive and performance implication

will spawn. If you simply wanted to check that *when “intr” went high*, the next clock “iack” should go high, then this way of coding has severe performance penalty. Too many threads will spawn, one every clock as long as “intr” is high. Instead, you may want to use edge-sensitive functions such as \$rose and \$fell.

The following would be more appropriate if all you wanted to do was to check for “iack” to go from inactive “0” state to active state “1” once edge-sensitive “intr” was asserted. After that, the state of “intr” does not matter. The following is a better way to write the property if your intention was to check for rising edge of iack one clock after rising edge of intr:

```
property checkiack;
  @ (posedge clk) $rose(intr) |=> $rose(iack);
endproperty
```

#### 14.17.4 \$stable

\$stable ( ), as the name implies, looks for its expression to be stable between two clock edges (i.e., two sampling edges) (Fig. 14.33). It evaluates the expression at current clock edge and compares it with the value sampled at the immediately preceding clock edge. If both values are same, the check passes.

```
$stable(StableSig [,clocking_event]);
```

*returns true if the value of the expression (StableSig) did not change from its sampled value at the previous clock tick.*

*[,clocking event] is optional and usually derived from the clocking event of the assertion or from the inferred clock of the procedural block where the function is used*

```
property noChangeSig (pclk, refSig, StableSig);
    @(posedge pclk) refSig |> $stable(StableSig);
endproperty

assert noChangeSig (sysClk, ConfigRd, ConfigRdParm); else failmsg;
```

#### \$stable in continuous assign

#### Explicit Clocking event for \$stable

```
assign stableVal = ($stable(ConfigSig), @(posedge clk)) ? sigVal : errorVal;
```

*If ConfigSig has been stable since last clock; assign sigVal to stableVal.*

*If ConfigSig did change since the last clock; assign errorVal to stableVal.*

Fig. 14.33 \$stable: basics

Note the use of \$stable in continuous assignment statement. You have to explicitly embed a clocking event with \$stable. This is the same rule that applies to other sampled value functions.

But what if you want to check if the signal has been stable for more than one clock? Read on... \$past ( ) will solve that problem.

### 14.17.5 \$past

\$past( ) is an interesting function. It allows you to go into past as many clocks as you wish to. You can check for an “expression1” to have a certain value, the number of clocks (strictly prior time ticks) in the past. Note that the number of ticks is optional. If you do not specify it, the default will be to look for “expression1” one clock in the past.

Another caveat that you need to be aware of is when you call \$past in the initial time ticks of simulation and there are not enough clocks to go in the “past.” For example, you specify “a l-> \$past (b)” and the antecedent “a” is a “1” at time 0. There is not a clock tick to go in the past. In that case, the assertion will use the “initial” value of “b.” What is the initial value of “b”? It’s not the one in the “initial” block; it’s the value with which the variable “b” was declared (as in “logic b = 1’b1;”). In our case, if “b” was not initialized in its declaration, the assertion will fail. If it was declared with an initial value of 1’b1, the assertion will pass.

```
$past (expression1, [, number_of_ticks] [,expression2] [,clocking_event]);
```

**\$past** returns the sampled value of the *expression1* that was present *number\_of\_ticks* prior to the time of evaluation of \$past

[,number\_of\_ticks] specifies the number of clock ticks in the past (default = 1)

[,expression2] is used as a gating expression for the clocking event of expression1

[,clocking event] is optional and usually derived from the clocking event of the assertion or from the inferred clock of the procedural block where the function is used

\$past function returns value (and NOT a boolean pass/fail as returned by \$rose,\$fell,\$stable)

```
bit [3:0] a,b,c;
always @(posedge clk)
begin
  if ($past(a) == 4'h5 ) $display ($stime,,,"t 'Past a' = %h",$past(a));
  if ($past(b) == 4'ha ) $display ($stime,,,"t 'Past b' = %h",$past(b));
  c = ($past(a) & $past(b));
end
```

'c' is assigned the bit wise '&' of the past values of a and b

```
# run -all
#
#      15  clk=1 a=5 b=a
#
#      25  clk=1 a=0 b=0
#
#      25  'Past a' = 5
#
#      25  'Past b' = a
```

Fig. 14.34 \$past: basics

You can also “gate” this check with expression2. The example in Fig. 14.35 shows how \$past works. We are using a gating expression in this example. It is not a requirement as noted in Fig. 14.34, but it will most likely be required in your application. If expression2 is not specified, no clock gating is assumed.

If you understand the use of \$past with a gating expression, then its use without one will be straightforward to understand.

Figure 14.35 asserts the following property:

**lastV == \$past(Sig, numClocks, enb);**  
*checks for the 'lastV' on Sig, numClocks in the past, gated by 'enb'*

```
property IV(Sig,numClocks,enb,lastV);
  (lastV == $past(Sig, numClocks, enb) );
endproperty

assert property (@(posedge clk) done |-> IV(mySig, 2, enb, lastVal)) else
  $display($stime,,,"FAIL Expected lastVal=%h\n",lastVal);

cover property (@(posedge clk) done |-> IV(mySig, 2, enb, lastVal))
  $display($stime,,,"PASS Expected lastVal=%h\n",lastVal);

always @(posedge clk)
  $display($stime,,,"clk=%b mySig=%h past=%h enb=%h done=%b", clk, mySig,
    $past(mySig, 2, enb), enb, done);
```

*'enb' in (lastV == \$past(Sig,numClocks,enb) ); means ::  
sampling of 'Sig' is performed based on it's clock gated by 'enb'.*

*In other words, \$past evaluates 'Sig' iff 'enb' is true.*

Fig. 14.35 \$past: gating expression

```
assert property (@ posedge clk) done |-> lV (mySig,2, enb,
lastVal) else $display(....);
```

And property IV (IV stands for the last value) models the following:

```
property lV(Sig, numClocks, enb, lastV);
  (lastV == $past (Sig, numClocks, enb));
endproperty
```

The property says check Sig and numClocks in the past and see if it has the value "lastV;" and do this check *if and only if* "enb" (the gating expression) is high when you *start* the check (i.e., when antecedent done=1 in the assert statement). *Let me re-emphasize that the gating expression is checked when you start the check.* The gating expression needs to be true when \$past is called. Many seem to miss this point. Without a gating signal, the property will always evaluate whenever the antecedent is true and look for the required expression value N number of clocks in the past.

**Useful Note on Debug** Note the use of \$past in the \$display statement. This way of using an expression in a \$display is an excellent way to debug your design. You can always display what happened in the past to debug the current state of design.

**Simulation Efficiency Tip** \$past(..., n, ...) is not efficient for big delays “n.” It is also recommended that you minimize the number of calls of \$past in an assertion and avoid calling it whenever it is not essential.

#### 14.17.5.1 Application: \$past()

In Fig. 14.36, note the use of \$past both in the consequent and in the antecedent. The top application uses it in consequent, while the bottom application uses it in antecedent.

*Application:* One more application uses \$past with consecutive range operator.

Please refer to the consecutive repetition operator described in Sect. 14.18.3.

*Specification:*

For a burst cycle, when the “read” signal is high until burst read is complete that during this period, the rd\_addr is incremented by one in every clock cycle.

*Solution:*

<b>application</b>	<b>Specification:</b> If current 'state' is cacheRead, the past state cannot be cachelInv (you can never Read from an invalid line)
	<pre>property rdPtrlncr;   @(posedge clk) (state == cacheRead)  -&gt; (\$past(state) != cachelInv); endproperty</pre>
<b>application</b>	<b>Specification:</b> If pipe stall is asserted and data was ready to be sent 2 clocks prior, that the current state must be data hold.
	<pre>property dHoldCheck;   @(posedge clk) (     (pipeStall)     &amp;&amp;     (\$past(State,2)==dataSend)   )    -&gt; (State == dataHold); endproperty</pre>

Fig. 14.36 \$past application

```

sequence check_rd_addr;
    ((rd_addr == $past (rd_addr+1)) && read) [*0:$] #\$fell (read);
endsequence

sequence read_cycle;
    ($rose(read) && reset_);
endsequence

property burst_check;
    @ (posedge clk) read_cycle |-> check_rd_addr;
endproperty

```

This property reads as follows.

Sequence “check\_rd\_addr” checks to see that current rd\_addr is equal to past rd\_addr+1. That is done using the \$past sampled value function. We then continue to check this behavior “consecutively” until \$fell(read) occurs (which is the end of the read cycle). So, every clock, the entire expression “((rd\_addr == \$past (rd\_addr+1)) && read)” is consecutively checked until “read” goes low. Please refer to the consecutive repetition operator described in Sect. 14.18.3.

Sequence read\_cycle checks for the start of the read cycle.

And property “burst\_check” connects the two sequences to check that the consecutive rd\_addr is in increment of 1 until the read cycle ends.

**Table 14.4** Concurrent assertion operators

Operator	Description
<b>##m</b>	Clock delay
<b>##[m:n]</b>	
<b>[*m]</b>	Repetition – consecutive
<b>[*m:n]</b>	
<b>[=m]</b>	Repetition – non-consecutive
<b>[=m:n]</b>	
<b>[&gt;m]</b>	GoTo repetition – non-consecutive
<b>[&gt; m:n]</b>	
<b>sig1 throughout seq1</b>	Signal sig1 must be true <i>throughout</i> sequence seq1
<b>seq1 within seq2</b>	Sequence seq1 must be contained <i>within</i> sequence s2
<b>seq1 and seq2</b>	“and” of two sequences. Both sequences must start at the same time but may end at different times
<b>seq1 intersect seq2</b>	“intersect” of two sequences; same as “and” but both sequences must start and end at the same time
<b>seq1 or seq2</b>	“or” of two sequences. It succeeds if either sequence succeeds
<b>first_match (sequence)</b>	Matches only the first of possibly multiple matches

**Table 14.5** Concurrent assertion operators – contd.

Operator	Description
<b>not &lt;property_expr&gt;</b>	If <property_expr> evaluates to true, then <i>not</i> <property_expr> evaluates to false and vice versa
<b>if (expression) property_expr1 else property_expr2</b>	If...else within a property
<b> &gt;</b>	Overlapping implication operator
<b> =&gt;</b>	Non-overlapping implication operator
<b>always, s_always, eventually, s_eventually, until, s_until, until_with, s_until_with, nexttime, s_nexttime</b>	Temporal operators
<b>iff, implies</b>	Equivalence operators
<b>#-#, #=#</b>	Followed by operators

## 14.18 Operators

This section describes all the operators offered by the language (both for a “sequence” and a “property”) including clock delay with and without range, consecutive repetition with and without range, non-consecutive repetition with and without range, “throughout,” “within,” “and,” “or,” “intersect,” “first\_match,” “if... else,” etc.

The following lists (Table 14.4 and Table 14.5) all the operators offered by the language (IEEE-1800, 2005). We will discuss operators of IEEE-1800-2009–2012 LRM in a separate section (see Sect. 14.24).

### 14.18.1 ##m: Clock Delay

Clock delay is about the most basic of all the operators and probably the one you will use most often! ##m means a delay of “m” number of sampling edges (or clocks). In this example (Fig. 14.37), the sampling edge is a “posedge clk”: hence ##m means “m” number of posedge clks.

The property evaluates antecedent “z” to be true at posedge clk and implies the sequence “Sab.” “Sab” looks for “a” to be true at that same clock edge (because of the overlapping operator) and, if that is true, waits for two posedge clks and then looks for “b” to be true.

In the simulation log, we see that at time 10, posedge of clk, z=1 and a=1. Hence, the sequence evaluation continues. Two clks later (at time 30), it checks to see if b=1, which it finds to be true and the property passes.

Similar scenario unfolds starting time 40. But this time, b is not equal to 1 at time 60 (two clks after time 40) and the property fails.

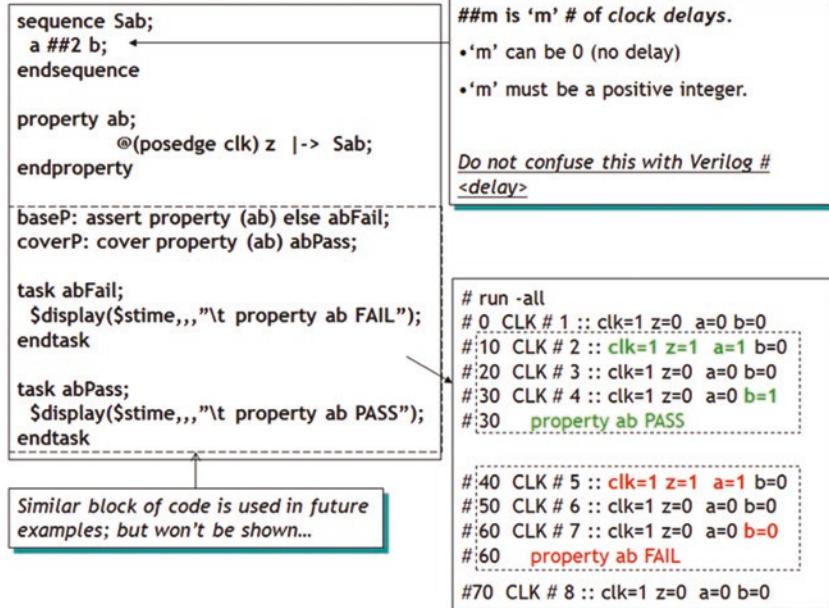


Fig. 14.37 ##m clock delay: basics

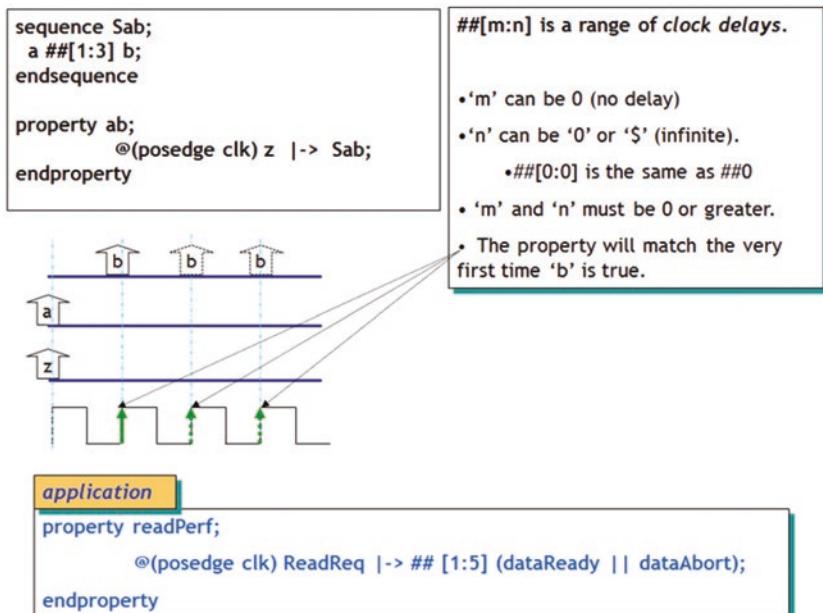


Fig. 14.38 ##[m:n] clock delay range

### 14.18.2 $\#\![m:n]$ : Clock Delay Range

In many applications, it may be necessary for a signal or expression to be true in a given *range* of clocks (as opposed to the fixed number of clocks). We need an operator that does just that.

$\#\![m:n]$  allows a range of sampling edges (clock edges) in which to check for the expression that follows it. Figure 14.38 explains the rules governing the operator. Note that here also *m* and *n* need to be constants. They cannot be variables.

The property “ab” in the figure says that if at the first posedge of clk, “z” is true that sequence “Sab” be triggered. Sequence “Sab” evaluates “a” to be true the same clock that “z” is true and then looks for “b” to be true delayed by either one clk or two clks or three clks. The very – first – instance that “b” is found to be true within the three clocks, the property will pass. If “b” is not asserted within three clks, the property will fail.

Note that in the figure you see three passes. That simply means that whenever “b” is true the first time within three clks, the property will pass. It does not mean that the property will be evaluated and pass three times.

#### 14.18.2.1 Clock Delay Range Operator: $\#\![m:n]$ – False Positive

Back to multiple threads! But this is a very interesting behavior of multi-threaded assertions. This is something you really need to understand.

Referring to Fig. 14.39, at s1, “rdy” is high and the antecedent is true. That implies that “rdyAck” be true within the next five clks. s1 thread starts looking for

```
property rdyProtocol;
  @(posedge clk) rdy |-> ##[1:5] rdyAck;
endproperty
```

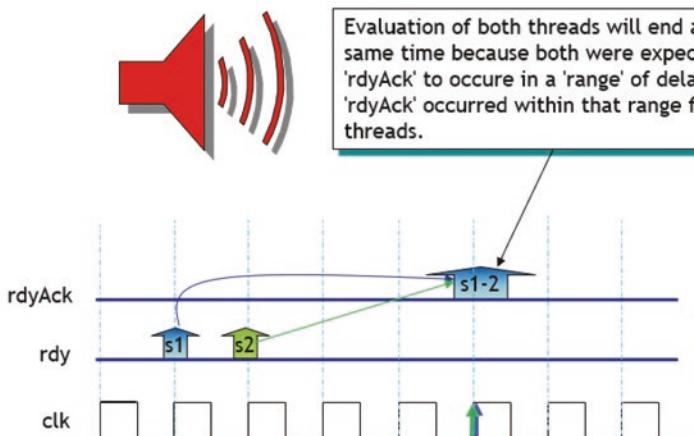


Fig. 14.39  $\#\![m:n]$ : multiple threads

“rdyAck” to be true. The very next clock, rdyAck is not yet true, but luck has it that “rdy” is true again at this next clk (s2). This will fork off another thread that will also wait for “rdyAck” to be true within the next five clks. The “rdyAck” comes along within five clks from s1 and that thread will pass.

But the second thread will also pass at the same time, because it also got its “rdyAck” within the five clks that it was waiting for.

This is a – very – important point to understand. *The range operator can cause multiple threads to complete at the same time.* This is in contrast to what we saw earlier with ##m constant delay where each thread will always complete only after the fixed ##m clock delays. There is a separate end to each separate thread. With the range delay operator, multiple threads can end at the same time.

**Important** Let us further explore this concept since *it can indeed lead to false positive*. How would you know which “rdyAck” is for which “rdy”? If you did not receive “rdyAck” for the second “rdy,” the property will *not* fail, and you will get a false positive.

One hint is to keep the antecedent an edge-sensitive function. For example, in the above example, instead of “@ (posedge clk) rdy,” we could have used “@ (posedge clk) \$rose(rdy)” which would have triggered the antecedent only once, and there will not be any confusion of multiple threads ending at the same time. This is a performance hint as well. Use edge-sensitive sampled value functions whenever possible. Level-sensitive antecedent can fork off unintended multiple threads affecting simulation performance. But this is not really a solution, just a possible workaround.

The real solution is to use local variables to ID each “rdy” and “rdyAck.” This will indeed make sure that you received a “rdyAck” for each “rdy” and also that each “rdyAck” is associated with the correct “rdy.”

Now, I have not explained local variables yet! Please refer to the section on local variables (see Sect. 14.21) to get familiar with it. That section is in-depth study of local variables. But you do not need to understand that entire section to understand the following example. Just scan through the initial pages of the section, and you will understand that local variables are *dynamic* variables with each instance of the sequence forking off an independent thread of local variables. You do not need to keep track of the pipelined behavior. The local variable does it for you. Having understood that, you should be able to follow the following example.

### Problem Statement

Please refer to Fig. 14.39. Two “rdy” signals are asserted on consecutive clocks with a range of clocks during which a “rdyack” must arrive. “rdyack” arrives that satisfies the time range requirements for *both* “rdy”s and the property passes. We have no idea whether a “rdyack” arrived for each “rdy.” The *pass* of the assertion does not guarantee that. In the following example, I am also using the concept of attaching subroutines (subroutine being \$display in this example), which is not covered so far but it is quite intuitive, and you should be able to understand attachment of \$display statements in the example.

The following example simulates the property and shows that both instances of “rdy” will *pass* with a single “rdyAck.”

**Problem:**

```

module range_problem;
  logic clk, rdy, rdyAck;

  initial
  begin
    clk=1'b0; rdy=0; rdyAck=0;
    #500 $finish(2);
  end

  always begin
    #10 clk=!clk;
  end

  initial
  begin
    repeat (5) begin @ (posedge clk) rdy=~rdy; end
  end

  initial $monitor($stime,,,"clk=",clk,,,"rdy=",rdy,,,"rdyAc
k=",rdyAck);

  initial
  begin
    repeat (4) begin @ (posedge clk); end
    rdyAck=1;
  end

  sequence rdyAckCheck;
    (1'b1,$display($stime ,,"ENTER SEQUENCE rdy ARRIVES"))
    ##[1:5]
    ((rdyAck),$display($stime ,,"rdyAck ARRIVES"));
  endsequence

  gcheck: assert property (@(posedge clk) rdy |-> rdyAckCheck)
  begin $display($stime,,,"PASS"); end
  else begin $display($stime,,,"FAIL"); end

endmodule

/* Simulation log:
0 clk=0 rdy=0 rdyAck=0

```

```

#      10 clk=1 rdy=1 rdyAck=0
#      20 clk=0 rdy=1 rdyAck=0
#      30 ENTER SEQUENCE rdy ARRIVES ← First ‘rdy’ is detected
#      30 clk=1 rdy=0 rdyAck=0

#      40 clk=0 rdy=0 rdyAck=0
#      50 clk=1 rdy=1 rdyAck=0
#      60 clk=0 rdy=1 rdyAck=0
#      70 ENTER SEQUENCE rdy ARRIVES ← Second ‘rdy’ is detected
#      70 clk=1 rdy=0 rdyAck=1
#      80 clk=0 rdy=0 rdyAck=1
#      90 clk=1 rdy=0 rdyAck=1
#      90 rdyAck ARRIVES ← ‘rdyack’ detected for both ‘rdy’s and the prop-
erty PASSes.
#      90 PASS
*/

```

**Solution:**

```

module range_solution;
  logic clk, rdy, rdyAck;
  byte rdyNum, rdyAckNum;

  initial
  begin
    clk=1'b0; rdy=0; rdyNum=0; rdyAck=0; rdyAckNum=0;
    #500 $finish(2);
  end

  always
  begin
    #10 clk=!clk;
  end

  initial
  begin
    repeat (4)
    begin
      @ (posedge clk) rdy=0;
      @ (posedge clk) rdy=1; rdyNum=rdyNum+1;
    end
  end

  initial $monitor($stime,,, "clk=",clk,,, "rdy=",rdy,,, "rdyNum=
",rdyNum,,, "rdyAckNum",rdyAckNum,,, "rdyAck=",rdyAck);

```

```

always
begin
    repeat (4)
        begin
            @ (posedge clk); @ (posedge clk); @ (posedge clk);
                rdyAck=1; rdyAckNum=rdyAckNum+1;
            @ (posedge clk) rdyAck=0;
        end
    end

    sequence rdyAckCheck;
    byte localData;
    /* local variable 'localData' declaration. Note this is a
    dynamic variable. For every entry into the sequence, it will create
    a new instance of localData and trigger an independent thread. */

    /* Store rdyNum in the local variable 'localData' */
    (1'b1,localData=rdyNum, $display ($stime ,,"rdy ARRIVES:
",,"LOCAL rdyNum=",localData))

    ##[1:5]

    /* Compare rdyAckNum with the rdyNum stored in 'localData' */
    ((rdyAck && rdyAckNum==localData),
     $display($stime ,,"rdyAck ARRIVES ",,"LOCAL" ,,
"rdyNum=",localData,, "rdyAck=",rdyAckNum));

```

**endsequence**

```

gcheck: assert property (@(posedge clk) (rdy) |-> rdyAck-
Check) begin $display($stime,,,"PASS"); end
    else begin $display($stime,,,"FAIL",,, "rdyNum=",rdyNum,,,
"rdyAckNum=",rdyAckNum); end

```

**endmodule**

```

/*
# 0 clk=0 rdy=0 rdyNum= 0 rdyAckNum 0 rdyAck=0
# 10 clk=1 rdy=0 rdyNum= 0 rdyAckNum 0 rdyAck=0
# 20 clk=0 rdy=0 rdyNum= 0 rdyAckNum 0 rdyAck=0
# 30 clk=1 rdy=1 rdyNum= 1 rdyAckNum 0 rdyAck=0
# 40 clk=0 rdy=1 rdyNum= 1 rdyAckNum 0 rdyAck=0
# 50 clk=1 rdy=1 rdyNum= 1 rdyAckNum 1 rdyAck=1

```

```

#      50 rdy ARRIVES: LOCAL rdyNum= 1 ← First 'rdy' arrives. A 'rdyNum'
(generated in your testbench as shown above) is assigned to 'localData'. This
'rdyNum' is a unique number for each invocation of the sequence and arrival
of 'rdy'.
#      clk=0 rdy=0 rdyNum= 1 rdyAckNum 1 rdyAck=1
#      70 clk=1 rdy=1 rdyNum= 2 rdyAckNum 1 rdyAck=0
#      70 rdyAck ARRIVES LOCAL rdyNum= 1 rdyAck= 1 //First 'rdy-
Ack' arrives
#      70 PASS ← When 'rdyAck' arrives, the sequence checks to see that its 'rdy-
AckNum' (again, assigned in the testbench) corresponds to the first rdyAck. If
the numbers do not match the property fails. Here they are indeed the same and
the property PASSes.

#      80 clk=0 rdy=1 rdyNum= 2 rdyAckNum 1 rdyAck=0
#      90 clk=1 rdy=0 rdyNum= 2 rdyAckNum 1 rdyAck=0
#      90 rdy ARRIVES: LOCAL rdyNum = 2 ← Second 'rdy' arrives. localData
is assigned the second 'rdyNum'. This redNum will not overwrite the first rdy-
Num. Instead, a second thread is forked off and 'localData' will maintain (store)
the second 'rdyNum'.
#      100 clk=0 rdy=0 rdyNum= 2 rdyAckNum 1 rdyAck=0
#      110 clk=1 rdy=1 rdyNum= 3 rdyAckNum 1 rdyAck=0
#      120 clk=0 rdy=1 rdyNum= 3 rdyAckNum 1 rdyAck=0
#      130 rdy ARRIVES: LOCAL rdyNum = 3 //Third 'rdy' arrives
#      130 clk=1 rdy=0 rdyNum= 3 rdyAckNum 2 rdyAck=1
#      140 clk=0 rdy=0 rdyNum= 3 rdyAckNum 2 rdyAck=1
#      150 clk=1 rdy=1 rdyNum= 4 rdyAckNum 2 rdyAck=0
#      150 rdyAck ARRIVES LOCAL rdyNum= 2 rdyAck= 2 //Second 'rdy-
Ack' arrives
#      150 PASS ← When 'rdyAck' arrives, the sequence checks to see that its 'rdy-
AckNum' corresponds to the second rdyAck. If the numbers do not match the prop-
erty fails. Here they are indeed the same and the property PASSes. This is what we
mean by pipelined behavior, in that, the second invocation of the sequence main-
tains its own copy of 'localData' and compares with the second 'rdyAck'. This way
there is no question of which 'rdy' was followed by which 'rdyAck'. No false posi-
tive. Rest of the simulation log follows the same chain of thought.

```

Can you figure out why the property fails at #270? Hint: Start counting clocks at time #170 when fourth 'rdy' arrives. Did a 'rdyAck' arrive for that 'rdy'?

```

#      160 clk=0 rdy=1 rdyNum= 4 rdyAckNum 2 rdyAck=0
#      170 rdy ARRIVES: LOCAL rdyNum = 4 //Fourth 'rdy' arrives
#      170 clk=1 rdy=1 rdyNum= 4 rdyAckNum 2 rdyAck=0
#      180 clk=0 rdy=1 rdyNum= 4 rdyAckNum 2 rdyAck=0
#      190 clk=1 rdy=1 rdyNum= 4 rdyAckNum 2 rdyAck=0
#      200 clk=0 rdy=1 rdyNum= 4 rdyAckNum 2 rdyAck=0
#      210 clk=1 rdy=1 rdyNum= 4 rdyAckNum 3 rdyAck=1
#      220 clk=0 rdy=1 rdyNum= 4 rdyAckNum 3 rdyAck=1

```

```

#      230 clk=1 rdy=1 rdyNum= 4 rdyAckNum 3 rdyAck=0
#      230 rdyAck ARRIVES LOCAL rdyNum= 3 rdyAck= 3 //Third 'rdy-
Ack' arrives
#      230 PASS

#      240 clk=0 rdy=1 rdyNum= 4 rdyAckNum 3 rdyAck=0
#      250 clk=1 rdy=1 rdyNum= 4 rdyAckNum 3 rdyAck=0
#      260 clk=0 rdy=1 rdyNum= 4 rdyAckNum 3 rdyAck=0
#      270 clk=1 rdy=1 rdyNum= 4 rdyAckNum 3 rdyAck=0
#      270 FAIL rdyNum= 4 rdyAckNum= 3 //Why does the assertion fail here?
  'rdyNum' and 'rdyAckNum' are not the same...
#      280 clk=0 rdy=1 rdyNum= 4 rdyAckNum 3 rdyAck=0
#      290 clk=1 rdy=1 rdyNum= 4 rdyAckNum 4 rdyAck=1
*/

```

Sequence “rdyAckCheck” is explained as follows.

Upon entry in the sequence, a copy of localData is created and a rdyNum is stored into it. While the sequence is waiting for #[1:5] for the rdyAck to arrive, another “rdy” comes in and sequence “rdyAckCheck” is invoked again. And again, the localData is assigned the next rdyNum and stored. This is where dynamic variable concept comes into picture. The second store of rdyNum into localData does not clobber the first store. A second copy of the localData is created, and its thread will also now wait for #[1:5]. Then, whenever “rdyAck” arrives, we compare its assigned number with the stored “rdy”Num. This way we make sure that for each “rdy” we will indeed get a unique “rdyAck.” Please carefully examine the simulation log to see how this works. I have placed comments in the simulation log to explain the operation.

### 14.18.3 *[\*m]: Consecutive Repetition Operator*

As depicted in Fig. 14.40, the consecutive repetition operator  $[*m]$  sees that the signal/expression associated with the operator stays true for “m” consecutive clocks. Note that “m” *cannot* be \$ (infinite # of consecutive repetition).

The important thing to note for this operator is that it will match at the *end* of the last iterative match of the signal or expression.

The example in Fig. 14.40 shows that when “z” is true, at the next clock, sequence “Sc1” should start its evaluation. “Sc1” looks for “a” to be true and then waits for one clock before looking for two consecutive matches on “b.” This is depicted in the simulation log. At time 10 “z” is high; at 20 “a” is high as expected (because of non-overlapping operator in property); and at times 30 and 40, “b” remains high matching the requirement b[\*2]. At the end of the second high on “b,” the property meets all its requirements and passes.

The very next part of the log shows that the property fails because “b” does not remain high for two consecutive clocks. Again, the comparison ends at the *last* clock where the consecutive repetition is supposed to end and then the property fails.

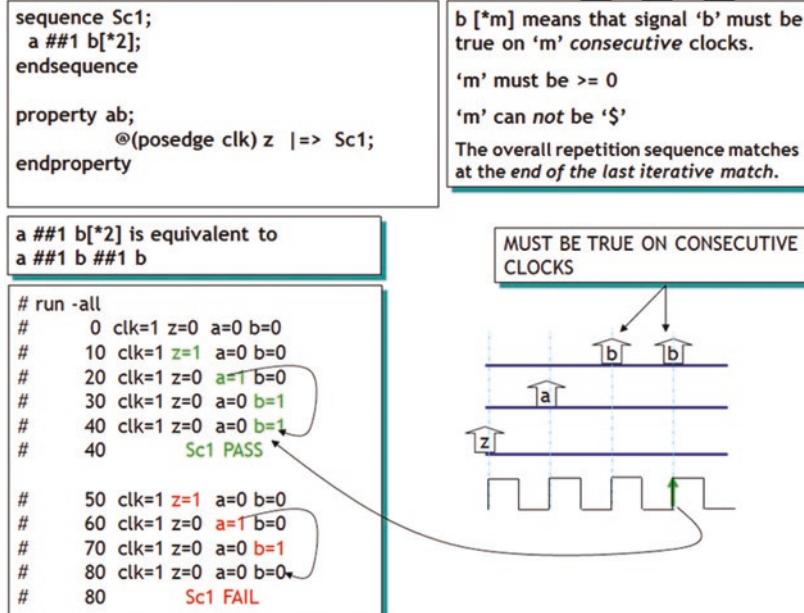
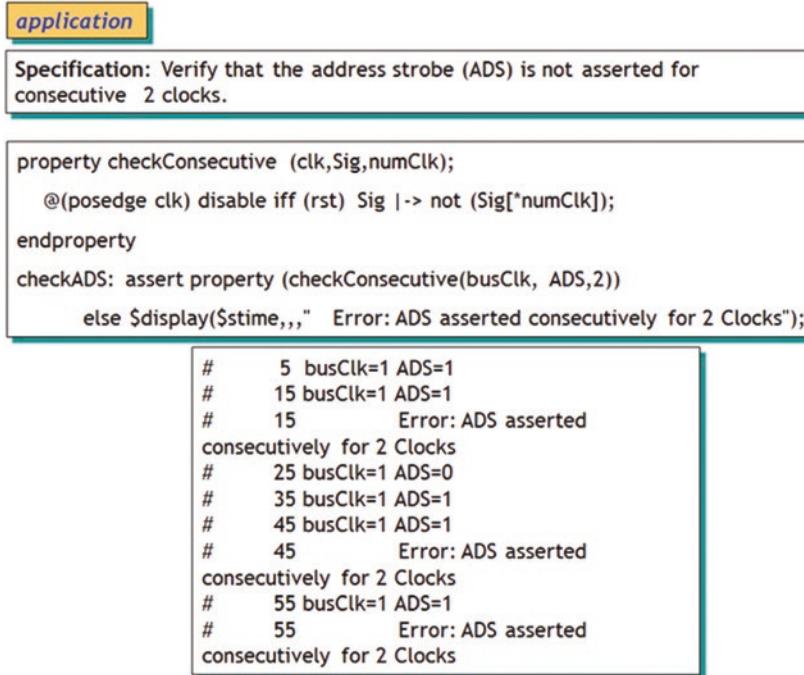
Fig. 14.40  $[*m]$ : consecutive repetition operator – basics

Fig. 14.41 Consecutive repetition operator: application

Figure 14.41 shows an interesting application where we effectively use the “not” of the repetition operator. At posedge busClk, if ADS is high then starting the same bus-Clk (overlapping operator), ADS is checked to see if it remains high consecutively for two busClk(s). If it does, then we take a “not” of it to declare that the property has failed. This is a very useful property, as simple as it looks. In many protocols one needs to make sure that certain signals follow very strict protocol. This application models just such a protocol. Note also the use of parameterized property.

Interesting to note is that if ADS is high for three consecutive clocks, the property will fail twice during those three clocks. Please see if you can figure out why. Hint, “Sig” is level sensitive.

#### **14.18.4 $[*m:n]$ : Consecutive Repetition Range Operator**

This is another important operator that you need to understand carefully, as benign as it appears.

Let us start with the basics.  $\text{sig}[*m:n]$  means that sig should remain true for minimum “m” number of consecutive clocks and maximum “n” number of consecutive clocks. That is simple enough. But here is the first thing that differs from the  $\text{sig}[*m]$  operator we just learned. The consecutive range operator match ends at the *first* match of the sequence that meets the required condition. Note this point carefully. It ends at the *first* match of the range operator (in contrast the non-range operator  $[*m]$  which ends at the *last* match of the “m”).

Figure 14.42 outlines the fact that  $b[*2:5]$  is essentially an OR of four different matches. When any one of these four sequences match, the property is considered to match and pass. In other words, the property first waits looking for two consecutive high on “b.” If it finds that sequence, the property ends and passes. If it does *not* find the second “b” to be true, the property will fail. It does *not* wait for the third consecutive high on “b” because there is not a third consecutive “b” if it was not consecutively high in the second clock. The chain was already broken. So, if “b” arrives in the second clock, the property will pass. If “b” was not true in the second clock, the property would fail. It will not wait for the max range.

So, why do we need the max range? When does the maximum range :5 come into picture? What does :5 really mean? See Fig. 14.43; it will explain what max range :5 means and how it gets used.

Note that we added “##1 c” in sequence Sc1. It means that there must be a “c” (high) one clock after the consecutive operator match is complete. Ok, simple enough.

Now let us look at the simulation log. Time 30 to 90 is straightforward. At time 30, z=1 and a=1, the next clock “b” = 1 and remains “1” for two consecutive clocks and then one clock later c=1 as required, and the property passes. But what if “c” was not equal to “1” at time 90? That is what the second set of events shows.

Z=1 and a=1 at time 110 and the sequence Sc1 continues. Ok. b=1 the next two clocks. Correct. But why doesn’t the property end here? Is it not supposed to end at

```

sequence Sc1;
  a ##1 b[*2:5];
endsequence

property ab;
  @(posedge clk) z |-> Sc1;
endproperty

```

a ##1 b[\*2:5] is equivalent to

```

a ##1 b ##1 b          ||
a ##1 b ##1 b ##1 b    ||
a ##1 b ##1 b ##1 b ##1 b ||
a ##1 b ##1 b ##1 b ##1 b

```

b [\*m:n] means that signal 'b' must be true on

*minimum 'm' consecutive clocks and maximum 'n' consecutive cycles.*

'm' must be  $\geq 0$ ;

'n' can be  $\geq 0$  or \$

The overall repetition sequence matches at the first match of the sequence that meets the required condition.



#### IMPORTANT POINT ::

The 'max' value (:5) in this example has meaning only if there is a qualifying event - *after*- b[\*2:5].

as in, a ##1 b[\*2:5] ##1 c;

In other words, if there isn't a "#1 c", the sequence will simply wait for the first 2 Consecutive 'b' and it will pass if it found them or fail if it didn't.

It would never wait for the max :5, because this is an OR.

So how does ":"5" work???

# run -all

```

#   90  clk=1 z=0 a=0 b=0
#  110  clk=1 z=1 a=1 b=0
#  130  clk=1 z=0 a=0 b=1
#  150  clk=1 z=0 a=0 b=1
#  150  Sc1 PASS
#  170  clk=1 z=0 a=0 b=1
#  190  clk=1 z=0 a=0 b=1
#  210  clk=1 z=0 a=0 b=1
#  230  clk=1 z=0 a=0 b=0
#  250  clk=1 z=1 a=1 b=0
#  270  clk=1 z=0 a=0 b=1
#  290  clk=1 z=0 a=0 b=0
#  290  Sc1 FAIL

```

Fig. 14.42 [\*m:n] consecutive repetition range: basics

the first match? Well, the reason the property does not end at 150 is because it needs to wait for c=1 the next clock. So, it waits for C=1 at 170. But it does not see a c=1. Shouldn't the property now fail? No. This is where the max range :5 comes into picture. Since there is a range [\*2:5], if the property does not see a c=1 after the first two consecutive repetitions of "b," it waits for the next consecutive "b" (total 3 now) and then looks for "c=1." If it does not see c=1, it waits for the next consecutive b=1 (total 4 now) and then looks for c=1. Still no "c"? It finally waits for max range :5, b=1, and then the next clock looks for c=1. If it finds one, the property ends and passes. If not, the property fails.

**Important** Here's how you need to read this assertion. Just as in clock delay range, here also (what I call) the qualifying event, namely, "#1 C," plays an important role. We are essentially saying that look for "#1 C" after two clocks or maximum of five clocks *but only until "#1 C" arrives*. We are looking for "b[\*2:5]" to occur but only until "C" arrives within those [\*2:5] clocks or at the max after five clocks. So, it is the end event a.k.a. qualifying event *after* the consecutive range is satisfied

```

sequence Sc1;
  a ##1 b[*2:5] ##1 c;
endsequence

property ab;
  @(posedge clk) z |-> Sc1;
endproperty

```

a ##1 b[\*2:5] is equivalent to

```

a ##1 b ##1 b      ##1 c    ||
a ##1 b ##1 b ##1 b ##1 c  ||
a ##1 b ##1 b ##1 b ##1 b ##1 c ||
a ##1 b ##1 b ##1 b ##1 b ##1 c ||

```

b [\*m:n] means that signal 'b' must be true on

*minimum 'm' consecutive clocks and maximum 'n' consecutive cycles.*

'm' must be  $\geq 0$ ;

'n' can be  $\geq 0$  or \$

*The overall repetition sequence matches at the first match of the sequence that meets the required condition.*

Requirement: After at least 2 consecutive High on 'b', if 'b' goes Low, that 'c' must go High the next clock.

But 'c' is low at #310 and the property fails.

```

# run -all
#
# 10 clk=1 z=0 a=0 b=0 c=0
#
# 30 clk=1 z=1 a=1 b=0 c=0
#
# 50 clk=1 z=0 a=0 b=1 c=0
#
# 70 clk=1 z=0 a=0 b=1 c=0
#
# 90 clk=1 z=0 a=0 b=0 c=1
#
# 90 Sc1 PASS
#
# 110 clk=1 z=1 a=1 b=0 c=0
#
# 130 clk=1 z=0 a=0 b=1 c=0
#
# 150 clk=1 z=0 a=0 b=1 c=0
#
# 170 clk=1 z=0 a=0 b=1 c=0
#
# 190 clk=1 z=0 a=0 b=1 c=0
#
# 210 clk=1 z=0 a=0 b=1 c=0
#
# 230 clk=1 z=0 a=0 b=0 c=1
#
# 230 Sc1 PASS
#
# 250 clk=1 z=1 a=1 b=0 c=0
#
# 270 clk=1 z=0 a=0 b=1 c=0
#
# 290 clk=1 z=0 a=0 b=1 c=1
#
# 310 clk=1 z=0 a=0 b=0 c=0
#
# 310 Sc1 FAIL

```

Fig. 14.43 [\*m:n] consecutive repetition range: example

(whether after two clocks or three clocks or four clocks or maximum of five clocks) that the qualifying event ends the assertion. Read this carefully until you understand what is going on.

Note also the following new shortcuts introduced in LRM 2009. I personally prefer explicit declaration and not the short cuts unless you want to impress your manager!!!

```

[*] is an equivalent representation of [*0:$]
[+] is an equivalent representation of [*1:$]
##[*] is used as an equivalent representation of ##[0:$].
##[+] is used as an equivalent representation of ##[1:$].

```

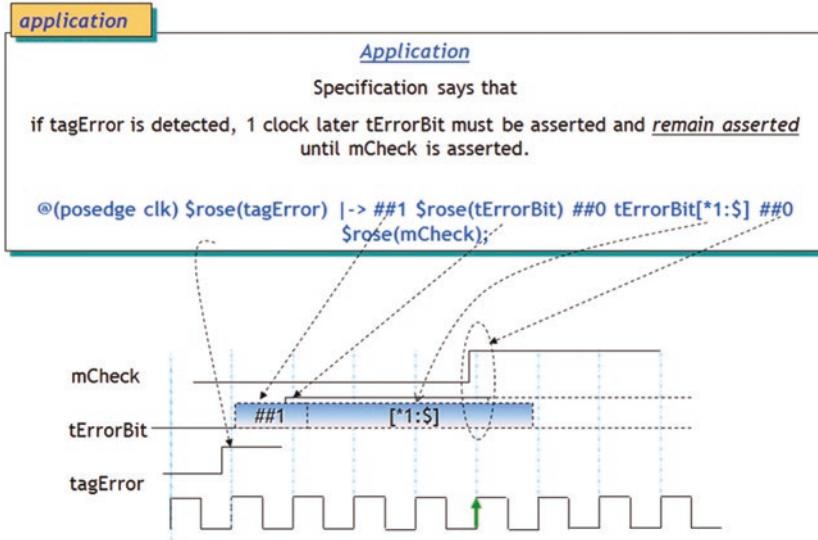


Fig. 14.44 [\*m:n] consecutive repetition range: application

#### 14.18.4.1 Applications: Consecutive Repetition Range Operator

This application is again on the same line that we have been following. The reason to carry on with the same example is to show how specification can change around seemingly similar logic.

Property in Fig. 14.44 says that at \$rose(tagError), check for tErrorBit to remain asserted until mCheck is asserted. If tErrorBit does not remain asserted until mCheck gets asserted, the property should fail.

So, at \$rose(tagError) and one clock later, we check to see that \$rose(tErrorBit) occurs. If it does, then we move forward at the same time (##0) with tErrorBit[\*1:\$]. This says that we check to see that tErrorBit remains asserted consecutively (i.e., at every posedge clk) forever but only *until the qualifying event \$rose(mCheck)*. In other words, the qualifying event is what makes consecutive range operator very meaningful as well as useful. *Think of the qualifying event as the one that ends the property.* This way, you can check for some expression to be true until the qualifying event occurs.

Another application.

Refer to Fig. 14.45. A PCI cycle starts when FRAME\\_is asserted (goes low) and the CMD is valid. A  $\text{CMD} == 4\text{'b}0001$  specifies the start of a PCI special cycle. On the start of such a cycle (i.e., the antecedent being true), the consequent looks for DEVSEL\\_ to be high forever consecutively at every posedge clk *until* FRAME\\_ is de-asserted (goes high). This is by far the easiest way to check for an event/expression to remain true (and we do not know for how long) until another condition/expression is true (i.e., *until* what I call the qualifying event or end event is true).

**application**

**Specification:**

PCI Special cycle requires that DEVSEL\_ should remain high (deasserted) during the special cycle.

```

property checkDevSelSpecialCycle;
  @(posedge clk)
    $fell(FRAME_) && (CMD == 4'b0001) |-> DEVSEL_*[1:$] ##0 $rose(FRAME_);
  endproperty

```

**HINT:** You can mix Edge Sensitive and Level Sensitive expressions in a logic condition.

Fig. 14.45 [\*m:n] consecutive repetition range: application

**application**

**Specification:**

Make sure that the state machine does not get stuck in current state except 'IDLE'.

```

property StuckState;
  @(posedge clk) disable iff (rst)
    ((currentState != IDLE) && $stable(currentState)[*32] |=> 1'b0);
  endproperty

```

**HINT:** You can simply declare your consequent as a failure.

Fig. 14.46 [\*m:n] consecutive repetition range: application

Note also that you can mix edge-sensitive and level-sensitive expressions in a single logic expression.

Another application (Fig. 14.46).

Property in Fig. 14.46 states that if the currentState of the state machine is not IDLE and if the currentState remains stable consecutively for 32 clocks, the property should fail.

There are a couple of points to observe.

Note that the entire expression `((currentState != IDLE) && $stable(currentState))` is checked for consecutive repetition of 32 times because we need to check at every clock for 32 clocks that the currentState is not IDLE and that it remains “!= IDLE” for 32 consecutive clocks. In other words, you have to make sure that within these 32 clocks, the current state does not go back to IDLE. If it does, then the antecedent does not match, and it will start all over again to check for the antecedent to be true.

Note that if the antecedent is indeed true it would mean that the state machine is indeed stuck into the same state for 32 clocks. In that case, we want the assertion to fail. That is taken care of by a *hard failure* in the consequent. We simply program consequent to fail without any pre-requisite.

As you notice, this property is unique in that the condition is checked for in the antecedent. The consequent is simply used to declare a failure.

One more application (Fig. 14.47)

Simulation log explains the behavior (Fig. 14.48).

#### Specification:

- When request is asserted that grant is asserted the very next clock.
  - grant must have been de-asserted prior to its assertion.*
- grant must remain asserted as long as request is asserted.
- grant must de-assert the very next clock after request is de-asserted.

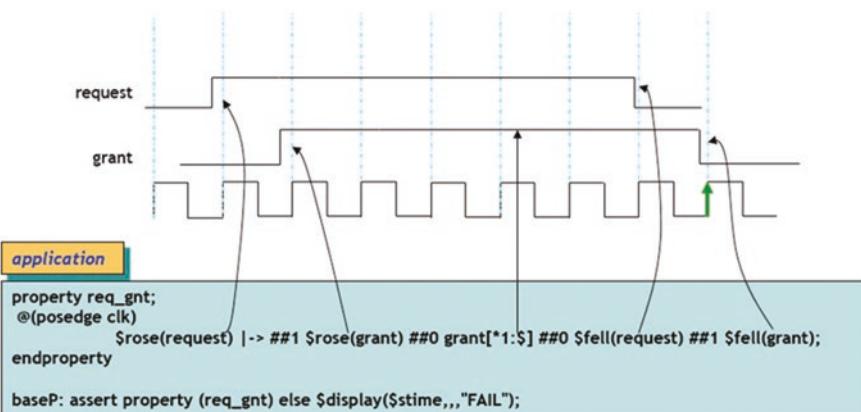


Fig. 14.47 Consecutive range application

```

# run -all
#      5 clk=1 request=0 grant=0
# 15 clk=1 request=0 grant=0
# 25 clk=1 request=1 grant=0
# 35 clk=1 request=1 grant=1
# 45 clk=1 request=1 grant=1
# 55 clk=1 request=1 grant=1
# 65 clk=1 request=0 grant=1
# 75 clk=1 request=0 grant=0
# 75 PASS
#
# 85 clk=1 request=0 grant=0
# 95 clk=1 request=1 grant=0
# 105 clk=1 request=1 grant=1
# 115 clk=1 request=1 grant=1
# 125 clk=1 request=1 grant=1
# 135 clk=1 request=1 grant=1
# 145 clk=1 request=1 grant=0
# 145 FAIL

```

grant falls before request. Hence the property fails.

**Fig. 14.48** Simulation log for consecutive range application

### 14.18.5 [=m]: Non-consecutive Repetition

Non-consecutive repetition is another useful operator (just as the consecutive operator) and used very frequently. In many applications, we want to check that a signal remains asserted or de-asserted a number of times and that we do *not* know when exactly these transitions take place. For example, if there is a non-burst *read* of length 8, you expect eight RDACK. These RDACK may come in a consecutive sequence *or not* (based on read latency). But you must have eight RDACK before read is done.

In Fig. 14.49, property “ab” says that if “a” is sampled high at the posedge of clock that starting next clock, “b” should occur twice not necessarily consecutively. They can occur any time after the assertion of “a.” The interesting (and important) thing to note here is that even though the property uses a non-overlapping implication operator (i.e., the first “b” should occur one clock after “a”=1), the first “b” can occur *any time after* one clock after “a” is found high. Not necessarily exactly one clock after “a”!!!

In the simulation log, the first part shows that “b” does occur one clock after “a” and then is asserted again a few clocks later. This meets the property requirements, and the assertion passes.

But note the second part of the log. “b” does – not – occur one clock after “a,” rather two clocks later. And then it occurs again a few clocks later. Even this behavior is considered to meet the property requirements and the assertion passes.

**Exercise** Based on the description above, do you think this property will ever fail? Please experiment and see if you can come up with the answer. It will also further confirm your understanding. Hint: There is no qualifying (i.e., end) event after “b[=2].”

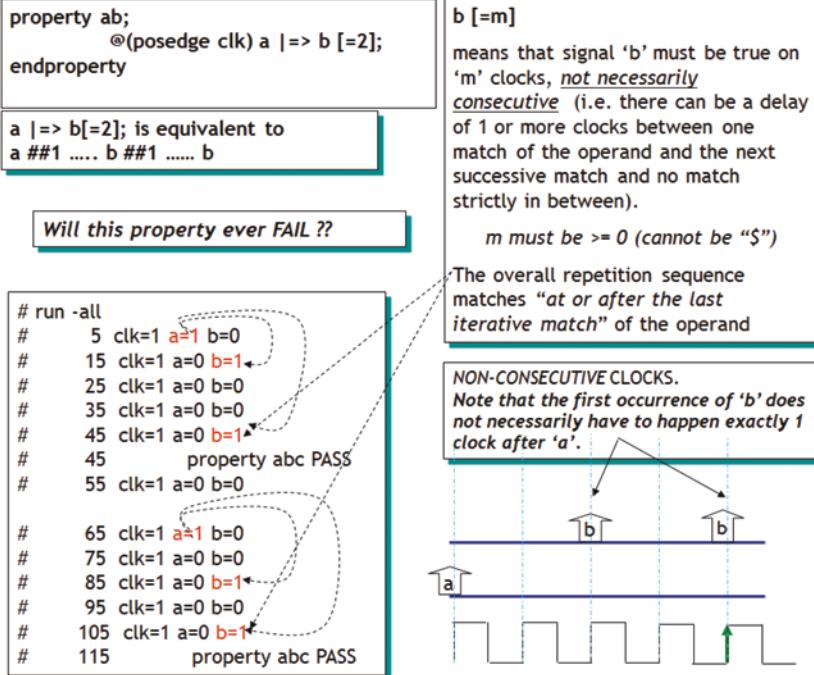


Fig. 14.49 Non-consecutive repetition operator

Continuing with the same analogy, refer to the example below (Fig. 14.50). Here again, just like in the consecutive operator, the qualifying event ("##1 C" in the example below) plays a significant role.

This is identical to the previous example except for the "##1 C" at the end of the sequence. The behavior of " $a \mid=> b[=2]$ " is identical to what we have seen above. "##1 c" tells the property that after the last "b," "c" must occur once and that it can occur *any time* after one clock after the last "b." Note again that even though we have "##1 c," "c" does *not* necessarily need to occur one clock after the last "b." It can occur after any number (#) of clks after one clock after the last "b" – as long as no other "b" occurs while we are waiting for "c." Confusing! Not really. Let us look at the simulation log in Fig. 14.50. That will clarify things.

In the log, a=1 at time 5, b=1 at time 25, and then at 45. So far so good. We are marching along just as the property expects. Then "c=1" occurs at time 75. That also meets the property requirement that "c" occurs any time after last b=1. *But* note that before c=1 arrived at time 75, "b" did not go to a "1" after its last occurrence at time 45. The property passes. Let us leave this at that for the moment. Now let us look at the second part of the log.

a=1 at time 95 and then b=1 at 105 and 125; we are doing great. Now we wait for c=1 to occur any time after last "b." C=1 occurs at time 175. But the property fails before that!! What is going on? Note b=1 at time 145. That is not allowed in this

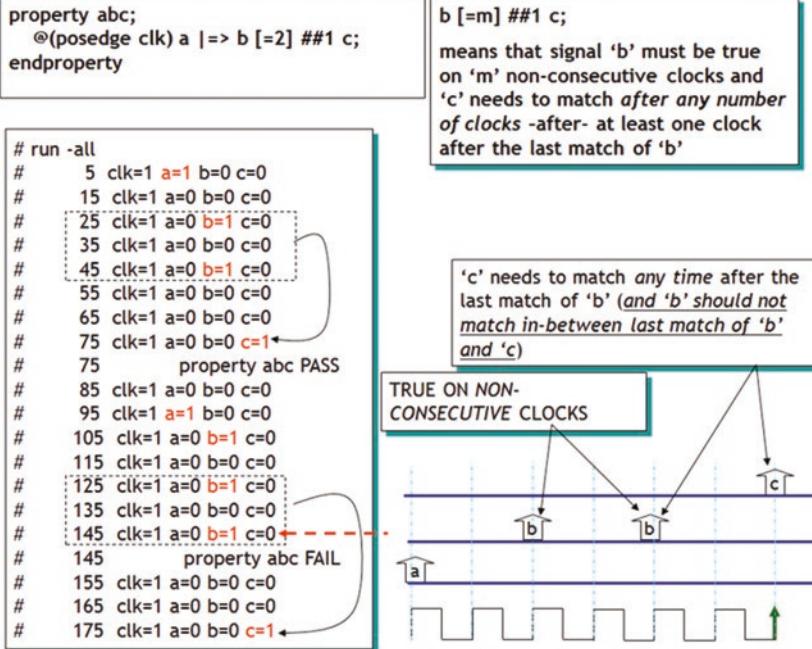


Fig. 14.50 Non-consecutive repetition operator: example

property. The property expects a  $c=1$  after the *last* occurrence of “b” but *before* any other  $b=1$  occurs. If another  $b=1$  occurs *before*  $c=1$  (as at time 145), then all bets are off. Property does not wait for the occurrence of  $c=1$  and fails as soon as it sees this extra  $b=1$ . In other words, (what I call) the qualifying event “##1 c” encapsulates the property and strictly checks that  $b[=2]$  allows only two occurrences of “b” before “c” arrives.

#### 14.18.6 $[=m:n]$ : Non-consecutive Repetition Range Operator

Property in Fig. 14.51 is analogous to the non-consecutive (non-range) property, except that this has a range. The range says that “b” must occur minimum two times or maximum five times after which “c” can occur any time after one clock and that no more than maximum of five occurrences of “b” occur between the last occurrence of  $b=1$  and  $c=1$ .

First simulation log (top left) shows that after  $a=1$  at time 5, b occurs twice (the minimum number (#) of times) at time 15 and 45 and then  $c=1$  at time 75. Why didn’t the property wait for five occurrences of  $b=1$ ? That is because after the second  $b=1$  at time 45,  $c=1$  arrives at time 75, and this  $c=1$  satisfies the property requirement of minimum of two  $b=1$  followed by a  $c=1$ . The property passes and does not

<pre> <b>property abc;</b>   @(posedge clk) a  =&gt; b [=2:5] ##1 c; <b>endproperty</b>  # #      5 clk=1 a=1 b=0 c=0 #      15 clk=1 a=0 b=1 c=0 #      25 clk=1 a=0 b=0 c=0 #      35 clk=1 a=0 b=0 c=0 #      45 .clk=1 a=0 b=1 c=0 #      55 clk=1 a=0 b=0 c=0 #      65 clk=1 a=0 b=0 c=0 #      75 clk=1 a=0 b=0 c=1 ← #      75 <b>property abc PASS</b> </pre>	<p><b>b [=m:n] ##1 c;</b></p> <p>means the property matches over an interval of clocks provided 'a' is true on the first clock tick, 'c' is true on the last clock tick and there are at least 'm' and at most 'n' not-necessarily consecutive clocks strictly in-between the first and the last on which 'b' is true (LRM :: SV 3.1a)</p> <p><i>m must be &gt;= 0 (cannot be "\$")</i></p> <p><i>n must be &gt;= 0 (can be "\$")</i></p>
<pre> <b># run -all</b> # #      5 clk=1 a=1 b=0 c=0 #      15 clk=1 a=0 b=0 c=0 #      25 clk=1 a=0 b=1 c=0 #      35 clk=1 a=0 b=1 c=0 #      45 clk=1 a=0 b=0 c=0 #      55 clk=1 a=0 b=1 c=0 #      65 clk=1 a=0 b=0 c=0 #      75 clk=1 a=0 b=1 c=0 #      85 clk=1 a=0 b=0 c=0 #      95 .clk=1 a=0 b=1 c=0 #      105 clk=1 a=0 b=0 c=0 #      115 clk=1 a=0 b=0 c=0 #      125 clk=1 a=0 b=0 c=1 ← #      125 <b>property abc PASS</b> </pre>	<pre> <b># run -all</b> # #      5 clk=1 a=1 b=0 c=0 #      15 clk=1 a=0 b=1 c=0 #      25 clk=1 a=0 b=0 c=0 #      35 clk=1 a=0 b=1 c=0 #      45 clk=1 a=0 b=0 c=0 #      55 clk=1 a=0 b=1 c=0 #      65 clk=1 a=0 b=0 c=0 #      75 clk=1 a=0 b=1 c=0 #      85 clk=1 a=0 b=0 c=0 #      95 .clk=1 a=0 b=1 c=0 #      105 clk=1 a=0 b=0 c=0 #      115 clk=1 a=0 b=1 c=0 ← ----- ··· #      115 <b>property abc FAIL:: # of</b> <b>posedge b' = 6</b> # #      125 clk=1 a=0 b=0 c=0 #      135 clk=1 a=0 b=0 c=0 #      145 clk=1 a=0 b=0 c=1 ← </pre>

Fig. 14.51 Non-consecutive range operator

need to wait for any further  $b=1$ . In other words, the property starts looking for the qualifying end event " $c=1$ " after the minimum required (2) " $b==1$ ." Since it did find a " $c=1$ " after two " $b=1$ ," the property ends there and passes.

Similarly, the simulation log on the bottom left shows that "b" occurs five (max) times and then "c" occurs without any occurrence of b in between the last occurrence of "b" and "c." The property passes. This is how that works. As explained above, after two " $b=1$ ," the property started looking for " $c==1$ ." But before the property detects " $c==1$ ," it sees another " $b==1$ ." That is ok because "b" can occur maximum of five times. So, after the third " $b==1$ ," the property continues to look for either " $c==1$ " or " $b==1$ " until it has reached maximum of five " $b==1$ ." This entire process continues until five "b"'s are encountered. Then the property simply waits for a "c."

**application**

**Specification:**

- If nonBurst Read of length 8 is asserted that the RdAck must be asserted 8 times and ReadDone must be asserted anytime after the last Read and that there are no more RdAck's between the last RdAck and ReadDone.

```

property RdAckCheck (int length);
    @(posedge clk) nBurstRead | => RdAck [=length] ##1 ReadDone;
endproperty
aP: assert property (RdAckCheck (8));

```

Fig. 14.52 Non-consecutive repetition range operator: application

So, what happens if you do not get a “c==1” after five “b”’s? While waiting for a “c” at this stage, if a sixth “b” occurs, the property fails. This failure behavior is shown in simulation log in the bottom right corner of Fig. 14.51.

#### 14.18.6.1 Application: Non-consecutive Repetition Operator

Here is a practical example of using non-consecutive operator (Fig. 14.52).

The property RdAckCheck will wait for nBurstRead to be high at the posedge clk. Once that happens, it will start looking for eight RdAck before ReadDone comes along. If ReadDone comes in after eight RdAck (and not nine), the property will pass. If ReadDone comes in before eight RdAck comes in, the property will fail. If a ninth RdAck arrives before ReadDone, the property will fail.

#### 14.18.7 [->] Non-consecutive GoTo Repetition Operator

This is the so-called non-consecutive GoTo operator! Very similar to [=m] non-consecutive operator. Note the symbol difference. The GoTo operator is [ -> ].

In Fig. 14.53, b[->2] acts exactly the same as b[=2]. So, why bother with another operator with the same behavior? It is the *qualifying event* that makes the difference. Recall that the qualifying event is the one that comes *after* the non-consecutive or the “GoTo” non-consecutive operator. I call it qualifying because it is *the end event* that qualifies the sequence that precedes for the final sequence matching.

In Fig. 14.54, we have the so-called qualifying event “##1 c.” The property says that on finding a=1 at posedge clk, b must be true two times (one clock after a=1) non-consecutively, and “c” must occur *exactly* one clock after the last occurrence of

```
property ab;
  @(posedge clk) a |=> b [-> 2];
endproperty
```

a |=> b[-> 2]; is equivalent to  
a ##1 ..... b ##1 ..... b

```
# run -all
#      5 clk=1 a=1 b=0
#     15 clk=1 a=0 b=1
#    25 clk=1 a=0 b=0
#   35 clk=1 a=0 b=0
#  45 clk=1 a=0 b=1
#  45 property abc PASS
#  55 clk=1 a=0 b=0
```

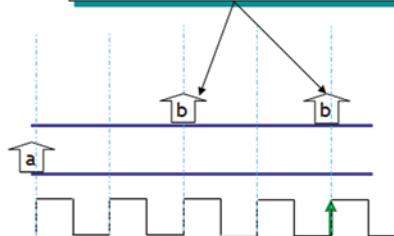
```
# 65 clk=1 a=1 b=0
# 75 clk=1 a=0 b=0
# 85 clk=1 a=0 b=1
# 95 clk=1 a=0 b=0
# 105 clk=1 a=0 b=1
# 115 property abc PASS
```

b [-> m] means that signal 'b' must be true on 'm' clocks, not necessarily consecutive (i.e. there can be a delay of 1 or more clocks between one match of the operand and the next successive match and no match strictly in between).

m must be >= 0 (cannot be "\$")

The overall repetition sequence matches "at the last iterative" match of the operand.

Matches ON NON-CONSECUTIVE CLOCKS



**NOTE::** Since there is no qualifying event -after- b[-> 2]; in this example, there is no difference between this example and the one with b[ =2] without a qualifying event. It's the qualifying event that differentiates between non-consecutive [= m] and the goto [-> m] constructs. Next slide...

Fig. 14.53 GoTo non-consecutive repetition operator

"b." In contrast, with "b[=2] ##1 c," "c" could occur *any time* after one clock after the last occurrence of "c." That is the difference between [=m] and [->m].

The simulation log in this example shows a *pass* and a *fail* scenario. *Pass* scenario is quite clear. At time 5, a==1, then two non-consecutive "b" occur, and then *exactly* one clock after the last "b=1," "c=1" occurs. Hence, the property passes. The *fail* scenario shows that after two occurrences of b==1, c==1 does *not* arrive *exactly* one clock after the last occurrence of b=1. That is the reason the b[->2] ##1 c check fails.

#### 14.18.8 Difference Between [=m:n] and [->m:n]

Refer to the simulation log in Fig. 14.55. The left-side and the right-side properties are identical except that the LHS uses [=2:5] and RHS uses [->2:5]. The LHS log, i.e., the one for b[=2:5] *passes*, while the one for b[->2:5] *fails* because according to

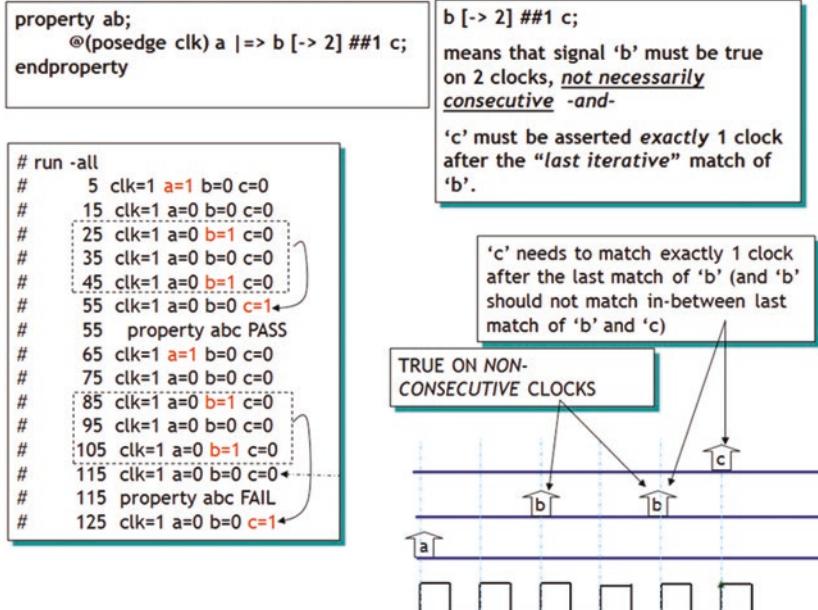


Fig. 14.54 Non-consecutive repetition: example

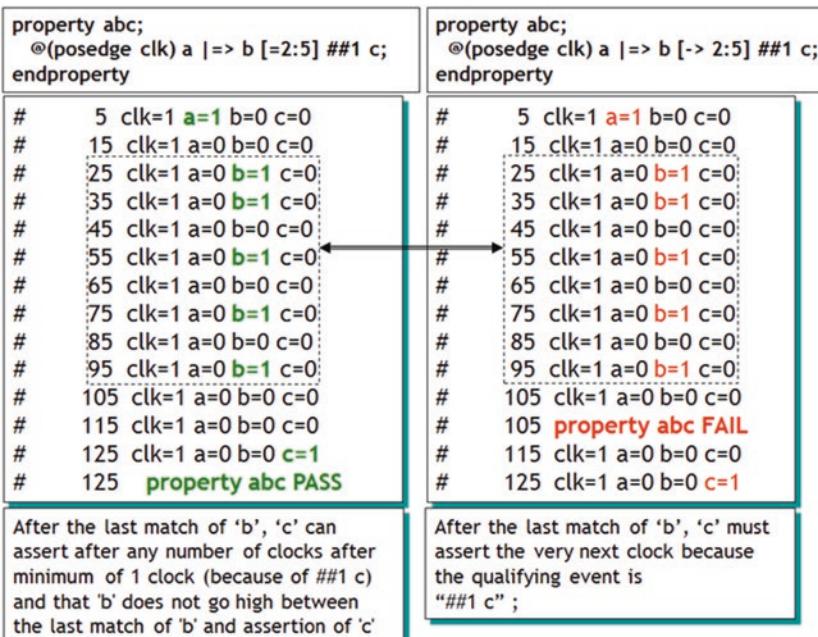


Fig. 14.55 Difference between [=m:n] and [-&gt; m:n]

**application****Specification:**

- For every 'req' you must get *at least* 1 'ack' and 'ack' must clear the next clock.

```
property ReqAckCheck;
```

```
    @(posedge clk) $rose(req) |=> ack[>-1] ##1 !ack;
```

```
endproperty
```

```
aP: assert property (reqAckCheck);
```

**Fig. 14.56** GoTo repetition non-consecutive operator: application

the semantics of “`b[>-2:5] ##1 c`,” “c” must arrive exactly one clock after the last occurrence of “b.”

Now here is a very important point. Note that “c” is expected to come in one clk after the last occurrence of b=1 because of “##1 c.” But what if you have “##2 c” in the property?

`b[=m] ##2 C`: This means that after “m” non-consecutive occurrence of “b,” “c” can occur any time *after* two clocks. The property does not start looking for “c” before two clocks, but any time after two clocks.

`b[>-m]##2 c`: This means that after “m” non-consecutive occurrence of “b,” “c” must occur *exactly* after two clocks. No more, no less.

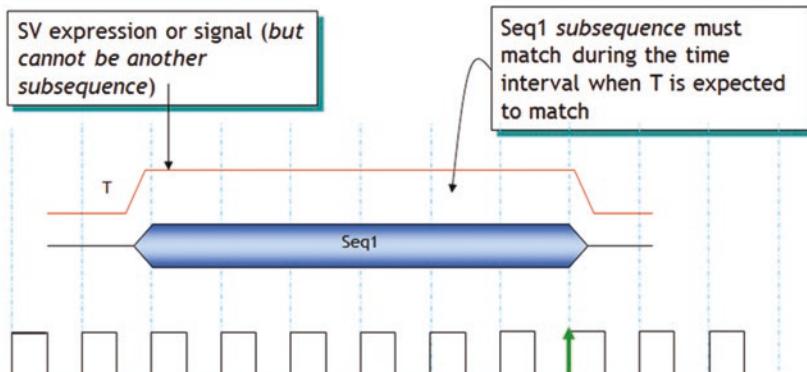
### 14.18.9 Application: GoTo Repetition – Non-consecutive Operator

This application (Fig. 14.56) says that at the rising edge of “req,” after one clock (because of non-overlapping operator), “ack” must occur once and that it must de-assert (go low) exactly one clock after its occurrence. If “ack” is not found de-asserted (low) exactly one clock after the assertion of “ack,” the property will fail.

### 14.18.10 Sig1 throughout Seq1

The “throughout” operator (Fig. 14.57) makes it that much easier to test for condition (signal or expression) to be true throughout a sequence. Note that the LHS of “throughout” operator can only be a signal or an expression, but it cannot be a

**'T throughout Seq1'** matches along a finite interval of consecutive clock ticks provided Seq1 matches along the interval and T evaluates true at each clock tick of the interval



Useful when you want to describe that a logical condition must hold true (or not) throughout a transaction.

Fig. 14.57 Sig1 throughout seq1

#### application

1. When Burst Mode (bMode) is asserted, oe\\_ and dack\\_ must be asserted within 2 clocks.
2. oe\\_ and dack\\_ must remain asserted for minimum of 4 clocks
3. bMode must remain asserted *throughout* the duration of oe\\_ && dack\\_ assertion.

```
sequence data_transfer;
  ((dack_==0) && (oe_==0)) [*4];
endsequence

sequence checkbMode;
  (!bMode) throughout data_transfer;
endsequence

property prule1;
  @(posedge clk) $fell(bMode) |-> ##[1:2] ((dack_ && oe_) == 0) ##0 checkbMode;
endproperty
```

Sfell(bMode) :: Here's where the eval of sequence 'checkbMode' starts

((dack\_==0) && (oe\_==0)) [\*4]

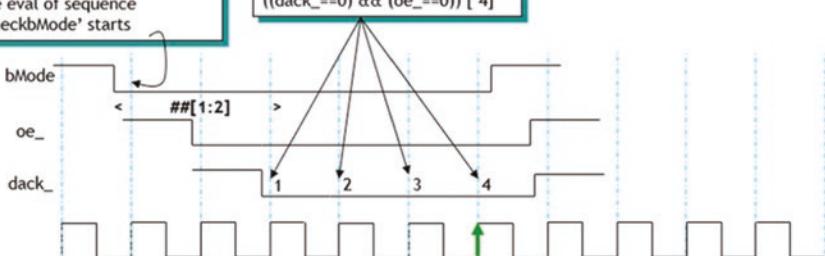


Fig. 14.58 Sig1 “throughout” seq1: application

bMode goes High a clock too early.

```

# 0 CLK #1 :: clk=1 bMode=1 oe_=1 dack_=1
#10 CLK #2 :: clk=1 bMode=1 oe_=1 dack_=1
#20 CLK #3 :: clk=1 bMode=0 oe_=1 dack_=1
#30 CLK #4 :: clk=1 bMode=0 oe_=0 dack_=1
#40 CLK #5 :: clk=1 bMode=0 oe_=0 dack_=0
#50 CLK #6 :: clk=1 bMode=0 oe_=0 dack_=0
#60 CLK #7 :: clk=1 bMode=0 oe_=0 dack_=0
#70 CLK #8 :: clk=1 bMode=1 oe_=0 dack_=0
    
```

(dack\_==0 && oe\_==0) does not hold for 4 clocks

```

#100 CLK #11 :: clk=1 bMode=1 oe_=1 dack_=1
#110 CLK #12 :: clk=1 bMode=0 oe_=1 dack_=1
#120 CLK #13 :: clk=1 bMode=0 oe_=0 dack_=1
#130 CLK #14 :: clk=1 bMode=0 oe_=0 dack_=0
#140 CLK #15 :: clk=1 bMode=0 oe_=0 dack_=0
#150 CLK #16 :: clk=1 bMode=0 oe_=0 dack_=0
#160 property pbrule1 FAIL
#160 CLK #17 :: clk=1 bMode=0 oe_=0 dack_=1
#170 CLK #18 :: clk=1 bMode=0 oe_=0 dack_=1
#180 CLK #19 :: clk=1 bMode=0 oe_=0 dack_=1
#190 CLK #20 :: clk=1 bMode=1 oe_=1 dack_=1
    
```

**Fig. 14.59** Sig1 “throughout” seq1: application simulation log

sequence (or subsequence). The RHS of “throughout” operator can be a sequence. So, what if you want a sequence on the LHS as well? That is accomplished with the “within” operator, discussed right after “throughout” operator.

Let us examine the application in Fig. 14.58, which will help us understand the throughout operator.

In Fig. 14.58, the antecedent in property pbrule1 requires bMode (burst mode) to fall. Once that is true, it first checks to see dack\_ && oe\_ == 0 within two clocks and, if so, requires checkbMode to execute.

checkbMode makes sure that the bMode stays low “throughout” the data\_transfer sequence. Note here that we are, in a sense, making sure that the antecedent remains true through the checkbMode sequence. If bMode goes high *before* data\_transfer is over, the assertion will fail. The data\_transfer sequence requires both dack\_ and oe\_ to be asserted (active low) and to remain asserted for four consecutive cycles. Throughout the data\_transfer, burst mode (bMode) should remain low.

There are two simulation logs presented in Fig. 14.59. Both are for *fail* cases! *Fail* cases are more interesting than the *pass* cases! The first simulation log (left-hand side) shows \$fell(bMode) at time 20. Two clocks later at time 40, oe\_=0 and dack\_=0, are detected. So far so good. oe\_ and dack\_ retain their state for three clocks. That is good too. But in the fourth cycle (time 70), bMode goes high. That is a violation because bMode is supposed to stay low throughout the data transfer sequence, which is four clocks long.

The second simulation log (right-hand side) also follows the same sequence as above but after three consecutive clocks that the oe\_ and dack\_ remain low and dack\_ goes high at time 160. That is a violation because data\_transfer (oe\_=0 and dack\_=0) is supposed to stay low for four consecutive cycles.

This also leads to a couple of other important points:

1. Both sides of the *throughout* operator must meet their requirements. In other words, if either the LHS or the RHS of the throughout sequence fails, the assertion will fail. Many folks assume that since bMode is being checked to see that it stays low (in this case), only if bMode fails, the assertion will fail. Not true as we see from the two failure logs.
2. **Important point:** In order to make it easier for the reader to understand this burst mode application, I broke it down into two distinct subsequences. But what if someone just gave you the timing diagram and asked you to write assertions for it?

Break down any complex assertion requirement into smaller chunks. This is probably the most important advice I can part to the reader. If you look at the entire AC protocol (the timing diagram) as one monolithic sequence, you will indeed make mistakes and spend more time debugging your own assertion than debugging the design under test.

**Exercise** How would you model this application using only the consecutive repetition [\*m] operator? Please experiment to solidify your concepts of both the *throughout* and the [\*m] operators.

### 14.18.11 Seq1 within Seq2

Analogous to “throughout,” the “within” operator sees if one sequence is contained within or of the same length as another sequence. Note that the “throughout” operator allowed only a signal or an expression on the LHS of the operator. “within” operator allows a sequence on both the LHS and RHS of the operator.

The property “within” ends when the larger of the two sequences ends, as shown in Fig. 14.60.

Let us understand “within” operator with the application in Fig. 14.61.

### 14.18.12 Application: Seq1 “within” Seq2

In this application, we again tackle the nasty protocol of burst mode! When burst mode is asserted, the master transmits (smtrx) must remain asserted for nine clocks, and the target Ack (tack) must remain asserted for seven clocks and that the “tack” sequence occurs within the “smtrx” sequence. This makes sense because from the

**'Seq1 within Seq2'** matches along a finite interval of consecutive clock ticks provided that Seq2 matches along the interval and Seq1 matches along some sub-interval of consecutive clock ticks.  
 Note that both Seq1 and Seq2 can be sequences.

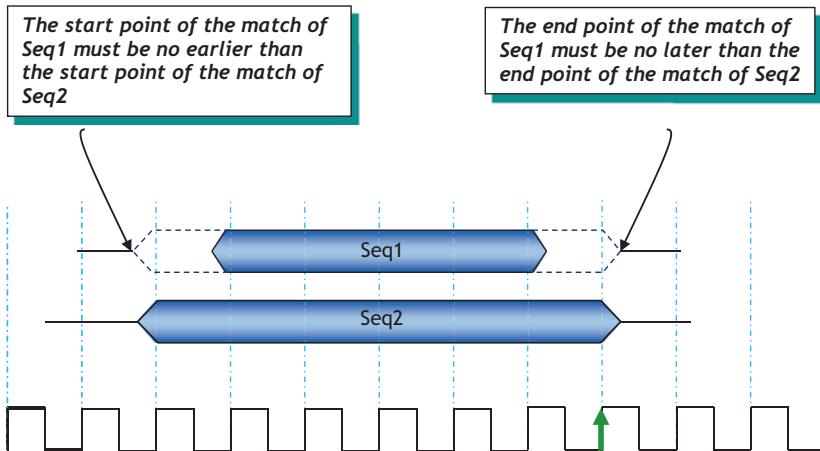


Fig. 14.60 Seq1 “within” seq2

protocol point of view, the target responds only after the master starts the request and the master completes the transaction after target is done.

As shown in the figure, the assertion of bMode (\$fell(bMode)) implies that “stack” is valid “within” “smtrx.” Now, carefully see the implication property “@ (posedge clk) \$fell(bMode)  $\Rightarrow$  stack **within** smtrx;”

LHS and RHS sequences start executing once the consequence fires. “stack” will evaluate to see if its condition remains true, and “smtrx” starts its own evaluation. At the same time, the “within” operator continually makes sure that “stack” is contained with “smtrx.” The annotations in Fig. 14.61 show how the property handles different parts of the protocol. Simulation logs are presented in Fig. 14.62.

On the left-hand side of Fig. 14.62, bMode is “1” at time 0 (not shown), and at time 10, it goes to “0.” That satisfies \$fell(bMode). After that, the consequent starts execution. *Both “stack” and “smtrx” sequences start executing*. As shown in the left-side simulation log, “mtrx” falls and stays low for nine clocks, as required. “tack” falls after “mtrx” falls, stays low for seven clocks, and goes high the same time when “mtrx” goes high (i.e., both sequences end at the same time). In other words, “tack” is contained within “mtrx.” This satisfies the “within” operator requirement and the property passes. Note that the operator “within” can have either sequence start or end at the same time. Similarly, the right-side log shows that both sequences start at the same time and “tack” is contained within “mtrx” and the property passes. Now let us look at fail cases.

**Specification:**

- Assertion of burst Mode (bMode) requires that Master Tx and Target Ack cycles follow the protocol below.
- Master Trx: mtrx must assert the clock after bMode and remains asserted for 9 clocks.
- Target Ack: tack must remain asserted for 7 clocks *within* mtrx transaction.

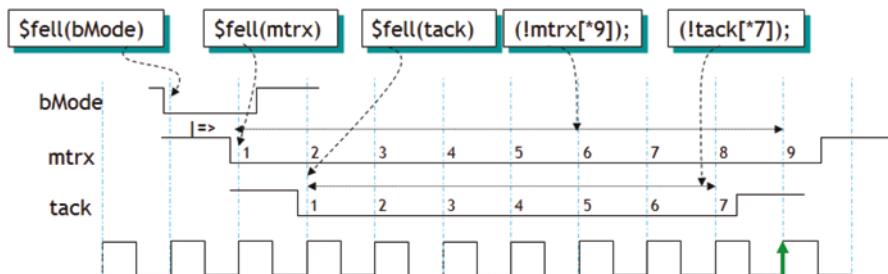
```

sequence stack;
  $fell(tack) ##0 !tack[*7];
endsequence

sequence smtrx;
  $fell(mtrx) ##0 (!mtrx[*9]);
endsequence

property pwin;
  @(posedge clk) $fell(bMode) |=> stack within smtrx;
endproperty

```



**Fig. 14.61** Seq1 “within” seq2: application

The simulation logs show different cases of failure. In the top log of Fig. 14.63, “tack” is indeed contained within “mtrx,” but “tack” does not remain asserted for required seven clocks and the property fails.

In the bottom log, again “tack” is contained within “mtrx,” but this time around, “mtrx” is asserted one clock too less.

The last log shows both “tack” and “mtrx” asserted for their required clks, but “tack” starts one clk before the falling edge of “mtrx,” thus violating the “within” semantics. Sequences on either side of “within” can start at the same time or end at the same time, but the sequence that is to be contained within the larger sequence cannot start earlier or end later than the larger sequence.

Another important point to note from these simulation logs is that the *property ends when the larger of the two sequences ends*. In our case, the property does not end as soon as there is a violation on “stack.” It waits for “smtrx” to end to make a judgment call on pass/fail of the property “pwin.”

```

10 CLK #2 :: clk=1 bMode=0 mtrx=1 tack=1
20 CLK #3 :: clk=1 bMode=0 mtrx=0 tack=1
30 CLK #4 :: clk=1 bMode=0 mtrx=0 tack=1
40 CLK #5 :: clk=1 bMode=0 mtrx=0 tack=0
50 CLK #6 :: clk=1 bMode=0 mtrx=0 tack=0
60 CLK #7 :: clk=1 bMode=0 mtrx=0 tack=0
70 CLK #8 :: clk=1 bMode=0 mtrx=0 tack=0
80 CLK #9 :: clk=1 bMode=0 mtrx=0 tack=0
90 CLK #10 :: clk=1 bMode=0 mtrx=0 tack=0
100 CLK #11 :: clk=1 bMode=0 mtrx=0 tack=0
    110 property pwin PASS
110 CLK #12:: clk=1 bMode=1 mtrx=1 tack=1

```

mtrx and tack deassert at the same.

That's OK, as long as tack did remain asserted for required clocks within mtrx.

```

140 CLK #15 :: clk=1 bMode=1 mtrx=1 tack=1
150 CLK #16:: clk=1 bMode=0 mtrx=1 tack=1
160 CLK #17:: clk=1 bMode=0 mtrx=0 tack=0
170 CLK #18 :: clk=1 bMode=0 mtrx=0 tack=0
180 CLK #19 :: clk=1 bMode=0 mtrx=0 tack=0
190 CLK #20 :: clk=1 bMode=0 mtrx=0 tack=0
200 CLK #21 :: clk=1 bMode=0 mtrx=0 tack=0
210 CLK #22 :: clk=1 bMode=0 mtrx=0 tack=0
220 CLK #23 :: clk=1 bMode=0 mtrx=0 tack=0
230 CLK #24 :: clk=1 bMode=0 mtrx=0 tack=1
240 CLK #25 :: clk=1 bMode=0 mtrx=0 tack=1
    250     property pwin PASS
250 CLK #26 :: clk=1 bMode=1 mtrx=1 tack=1

```

mtrx and tack assert at the same.

That's OK, as long as tack did remain asserted for required clocks within mtrx.

Fig. 14.62 “within” operator: simulation log – pass cases

### 14.18.13 Seq1 and Seq2

As the name suggests, “and” operator expects both the LHS and RHS side of the operator “and” to evaluate to true. It does not matter which sequence ends first as long as both sequences meet their requirements. The property ends when the longer of the two sequences ends. *But note that both the sequences must start at the same time.* Refer to Fig. 14.64.

The “and” operator is very useful, when you want to make sure that certain concurrent operations in your design start at the same time and that they both complete and match satisfactorily. As an example, in the processor world, when a read is issued to L2 cache, L2 will start a tag match and also issue a DRAM read, both at the same time, in anticipation that the tag may not match. If there is a match, it will abort the DRAM read. So, one sequence is to start tag compare, while other is to start a DRAM read (ending in DRAM read complete or abort). The DRAM read sequence is designed such that it will abort as soon as there is a tag match. This way we have made sure that both sequences start at the same time and that both end. Let us look at the following cases to clearly understand “and” semantics (Figs. 14.65 and 14.66). The figures explain the working with annotations.

```

#      0 CLK #1 :: clk=1 bMode=1 mtrx=1 tack=1
#     10 CLK #2 :: clk=1 bMode=1 mtrx=1 tack=1
#    20 CLK #3 :: clk=1 bMode=0 mtrx=1 tack=1
#    30 CLK #4 :: clk=1 bMode=0 mtrx=0 tack=1
#    40 CLK #5 :: clk=1 bMode=0 mtrx=0 tack=0
#   50 CLK #6 :: clk=1 bMode=0 mtrx=0 tack=0
#   60 CLK #7 :: clk=1 bMode=0 mtrx=0 tack=0
#   70 CLK #8 :: clk=1 bMode=0 mtrx=0 tack=0
#   80 CLK #9 :: clk=1 bMode=0 mtrx=0 tack=0
#  90 CLK #10 :: clk=1 bMode=1 mtrx=0 tack=0
# 100 CLK #11 :: clk=1 bMode=1 mtrx=0 tack=1
# 110 CLK #12 :: clk=1 bMode=1 mtrx=0 tack=1
#
#          property pwin FAIL

```

tack is asserted for 1 clock too less.

```

# 280 CLK #29 :: clk=1 bMode=1 mtrx=1 tack=1
# 290 CLK #30 :: clk=1 bMode=1 mtrx=1 tack=1
# 300 CLK #31 :: clk=1 bMode=0 mtrx=1 tack=1
# 310 CLK #32 :: clk=1 bMode=0 mtrx=0 tack=1
# 320 CLK #33 :: clk=1 bMode=0 mtrx=0 tack=0
# 330 CLK #34 :: clk=1 bMode=0 mtrx=0 tack=0
# 340 CLK #35 :: clk=1 bMode=0 mtrx=0 tack=0
# 350 CLK #36 :: clk=1 bMode=0 mtrx=0 tack=0
# 360 CLK #37 :: clk=1 bMode=0 mtrx=0 tack=0
# 370 CLK #38 :: clk=1 bMode=1 mtrx=0 tack=0
# 380 CLK #39 :: clk=1 bMode=1 mtrx=0 tack=0
# 390 CLK #40 :: clk=1 bMode=1 mtrx=1 tack=1
#
#          property pwin FAIL

```

mtrx is asserted 1 clock too less ...

tack is asserted for 7 clocks but started a clock too early, so was asserted 1 clock earlier 'within' the mtrx sequence

```

# 140 CLK #15 :: clk=1 bMode=1 mtrx=1 tack=1
# 150 CLK #16 :: clk=1 bMode=1 mtrx=1 tack=1
# 160 CLK #17 :: clk=1 bMode=0 mtrx=1 tack=0
# 170 CLK #18 :: clk=1 bMode=0 mtrx=0 tack=0
# 180 CLK #19 :: clk=1 bMode=0 mtrx=0 tack=0
# 190 CLK #20 :: clk=1 bMode=0 mtrx=0 tack=0
# 200 CLK #21 :: clk=1 bMode=0 mtrx=0 tack=0
# 210 CLK #22 :: clk=1 bMode=0 mtrx=0 tack=0
# 220 CLK #23 :: clk=1 bMode=0 mtrx=0 tack=0
# 230 CLK #24 :: clk=1 bMode=1 mtrx=0 tack=1
# 240 CLK #25 :: clk=1 bMode=1 mtrx=0 tack=1
# 250 CLK #26 :: clk=1 bMode=1 mtrx=0 tack=1
#
#          property pwin FAIL

```

Fig. 14.63 “within” operator: simulation log –fail cases

'Seq1 and Seq2' match if

- Both sequences start at the same time
- Both sequences match
- The end time of each sequence can be different.

The end time is the end time of either Seq1 or Seq2, whichever matches last.

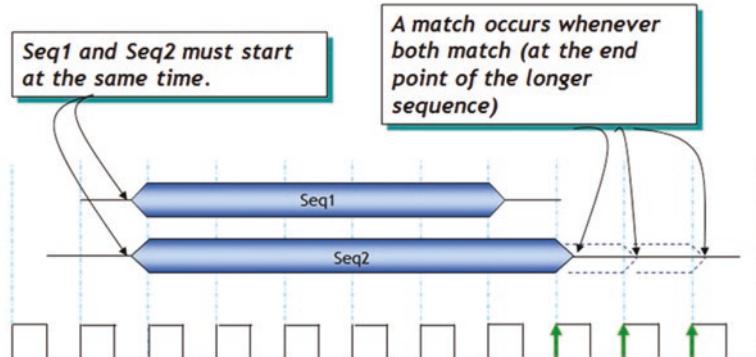


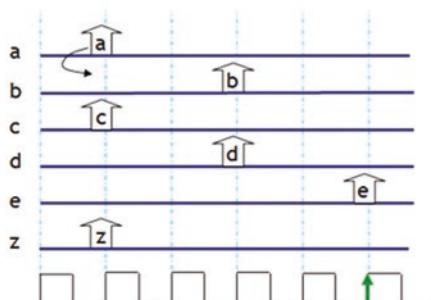
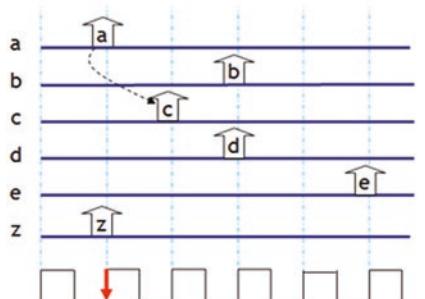
Fig. 14.64 Seq1 “and” seq2: basics

```
sequence ab;
  a #2 b;
endsequence

sequence cde;
  c #2 d #2 e;
endsequence

sequence abcde;
  ab and cde;
endsequence

property ands;
  @ (posedge clk) z |-> abcde;
endproperty
```



**property ‘ands’ PASS (at the end point of the longer sequence ‘cde’).**

**property ‘ands’ FAIL because ‘a’ and ‘c’ did not assert at the same time, so ‘ab and ‘cde’ sequences did not start at the same time.**

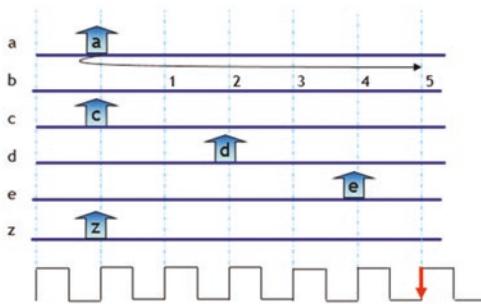
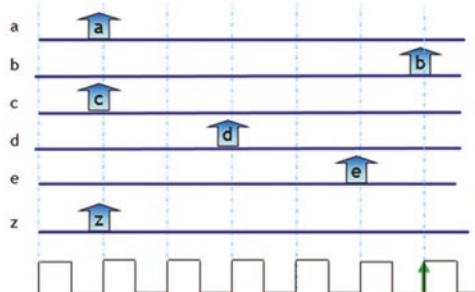
Fig. 14.65 “and” operator: application

```

sequence ab;
  a ##[1:5] b;
endsequence
sequence cde;
  c ##2 d ##2 e;
endsequence
sequence abcde;
  ab and cde;
endsequence

property ands;
  @(posedge clk) z |-> abcde;
endproperty

```



**property 'ands' PASS (at the end point of the longer sequence 'ab').**

**property 'ands' FAIL because 'b' does not assert within 1:5 clocks after 'a'. Property waits for 5 clocks after 'a' and fails when it does not detect asserted 'b'.**

Fig. 14.66 “and” operator: application II

#### 14.18.14 Application: “and” Operator

In Fig. 14.67, we “and” two expressions in a property. In other words, as noted before, an “and” operator allows a signal, an expression, or a sequence on both the LHS and RHS of the operator. The simulation log is annotated with pass/fail indication.

#### 14.18.15 Seq1 or Seq2

“or” of two sequences means that when either of the two sequences match its requirements, the property will pass. Please refer to Fig. 14.68 and examples that follow to get a better understanding.

The feature to note with “or” is that as soon as either of the LHS or RHS sequence meets its requirements, the property will end. This is in contrast to “and” where only after the longest sequence ends that the property is evaluated.

Note also that if the shorter of the two sides fails, the sequence will continue to look for a match on the longer sequence. The following examples make this clear.

Note that “seq1” and “seq2” need not start at the same time.

```
property ands;
  @(posedge clk) z |-> (a==b) and (c==d);
endproperty
```

```
# run -all
#      5 CLK # 1 :: clk=1 z=0 a=0 b=0 c=0 d=0
#     15 CLK # 2 :: clk=1 z=0 a=0 b=0 c=0 d=0
#    25 CLK # 3 :: clk=1 z=1 a=1 b=1 c=0 d=0
#    25 property ands PASS

#   35 CLK # 4 :: clk=1 z=0 a=0 b=0 c=1 d=1
#   45 CLK # 5 :: clk=1 z=1 a=1 b=1 c=1 d=1
#   45 property ands PASS

#   55 CLK # 6 :: clk=1 z=0 a=0 b=1 c=0 d=1
#   65 CLK # 7 :: clk=1 z=1 a=1 b=0 c=1 d=1
#   65 property ands FAIL

#   75 CLK # 8 :: clk=1 z=1 a=1 b=1 c=0 d=1
#   75 property ands FAIL
```

Note that you can do an ‘and’ of sequences or expressions or a combination of the two.

Fig. 14.67 “and” of expressions

‘Seq1 or Seq2’ match if

- operand ‘or’ is used when at least one of the two operand sequences is expected to match.

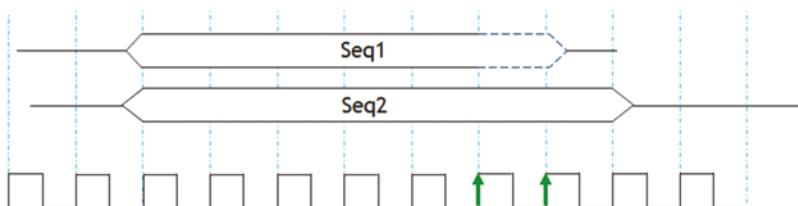


Fig. 14.68 Seq1 “or” seq2: basics

A simple property is presented in Fig. 14.69. Different cases of passing of the property are shown. On the top right of the figure, both “ab” and “cde” sequences start at the same time. Since this is an “or,” as soon as “ab” completes, the property completes and passes. In other words, the property does not wait for “cde” to complete anymore.

On the bottom-left corner, we see that “ab” sequence fails. However, since this is an “or,” the property continues to look for “cde” to be true. Well, “cde” does turn out to be true and the property passes (Fig. 14.70).

```

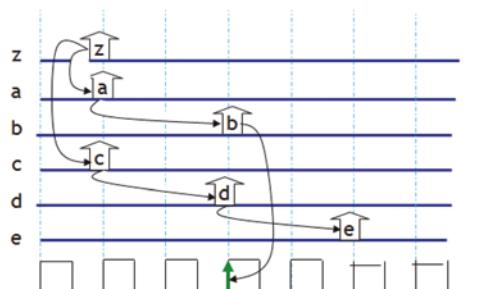
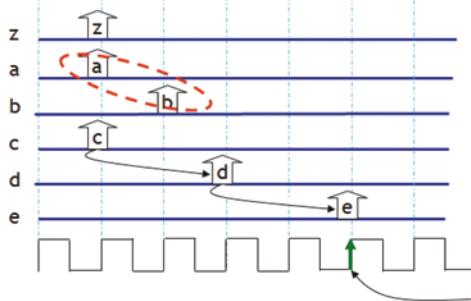
sequence ab;
  a ##2 b;
endsequence

sequence cde;
  c ##2 d ##2 e;
endsequence

sequence abcde;
  ab or cde;
endsequence

property ands;
  @(posedge clk) z |-> abcde;
endproperty

```



**Matches of both 'ab' and 'cde' are recognized.  
Property PASSes on the match of 'ab'**

**'ab' and 'cde' both start the same clock as 'z' (as required by |-> operator). But 'ab' fails, so the property continues to look for a match on 'cde' and PASS when sees a match on 'cde'**

Fig. 14.69 “or” operator: application

### 14.18.16 Seq1 “intersect” Seq2

So, with “throughout,” “within,” “and,” and “or” operators, who needs another operator that also seems to verify that sequences match?

“throughout” or “within” or “and” or “or” does *not* make sure that both the LHS and RHS sequences of the operator are of exactly the same length. They can be of the same length, but the operators do not care as long as the signal/expression or sequence meets their requirements. That is where “intersection” comes into picture (Fig. 14.71). It makes sure that the two sequences indeed *start at the same time and end at the same time* and satisfy their requirements. In other words, they intersect. Essentially, “intersect” is an “and” with length restriction.

As you can see, the difference between “and” and “intersect” is that “intersect” requires both sequences to be of the same length and that they both start at the same time and end at the same time, while “and” can have the two sequences of different lengths. I have shown that difference with timing diagrams further down the chapter. But first, here are some simple examples to better understand “intersect”.

```

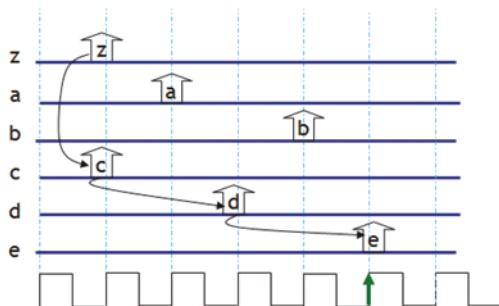
sequence ab;
  a ##2 b;
endsequence

sequence cde;
  c ##2 d ##2 e;
endsequence

sequence abcde;
  ab or cde;
endsequence

property ands;
  @(posedge clk) z |-> abcde;
endproperty

```



*Here 'a' is asserted 1 clock later and 'ab' does satisfy its requirement, but 'a' was not asserted the same time as 'z' (as required by overlap implication). however, 'c' was indeed asserted when 'z' was asserted, the property is looking for 'cde' to match. Since 'cde' does match, the property passes at the end of 'cde' (and not at the end of 'ab').*

```

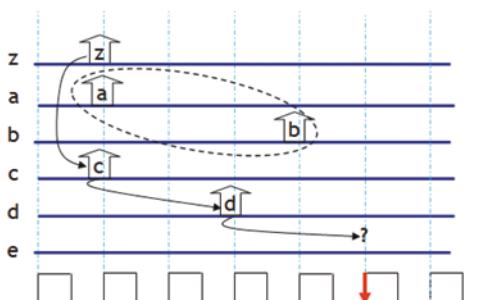
sequence ab;
  a ##2 b;
endsequence

sequence cde;
  c ##2 d ##2 e;
endsequence

sequence abcde;
  ab or cde;
endsequence

property ands;
  @(posedge clk) z |-> abcde;
endproperty

```



*Here, 'ab' does not match; but property keeps looking to see if 'cde' matches. But when 'e' does not follow 2 clocks after d, 'cde' also fails and the property FAILS at that time.*

Fig. 14.70 “or” operator: application II

### 14.18.17 Application: “intersect” Operator

Figure 14.72 shows two cases of failure with the “intersect” operator.

Property “isect” says that if “z” is sampled true at the posedge clk, sequence “abcde” should be executed and hold true. I have broken down the required sequence

**'Seq1 intersect Seq2' match if**

- Both sequences start at the same time
- Both sequences must match
- The lengths of the two matches of the operand sequences must be the same.

The end time is when both sequences match and end at the same time.

The main difference between 'and' and 'intersect' is the requirement on the length of the two sequences. For 'and' each sequence can be of any length. For 'intersect' they must be of the same length.

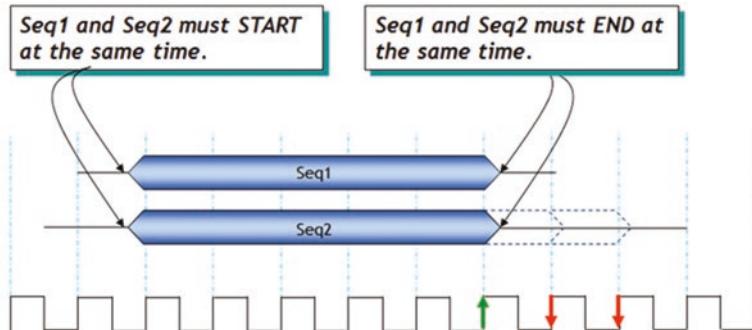


Fig. 14.71 Seq1 “intersect” seq2

into two subsequences. Sequence “ab” requires “a” to be true at posedge clk, and then “b” be true any time within one to five clks. Sequence “cde” is a fixed temporal domain sequence which requires c to be true at posedge clk, then “d” to be true two clocks later, and “e” to be true two clocks after “d.”

Top-right timing diagram in Fig. 14.72 shows that both “ab” and “cde” meet their requirements, but the property fails because they both do not end at the same time (even though they start at the same time). Similarly, the bottom-left timing diagram shows that both “cde” and “ab” meet their requirements but do not end at the same time and hence the assertion fails.

Ok, I admit this property (Fig. 14.73) could have been written simply as:

```
@ (posedge clk) $rose(Retry) |-> ##[1:4] $rose(dataRead);
```

So, why are we making it complicated? I just want to highlight an interesting way to use “intersect.”

When \$rose(Retry) is true, the consequent starts execution. The consequent uses “intersect” between `true[\*1:4] and (Retry ##[1:\$] \$rose(dataRead)). The LHS of “intersect” says that it will be true for consecutive four cycles. The RHS says that

```

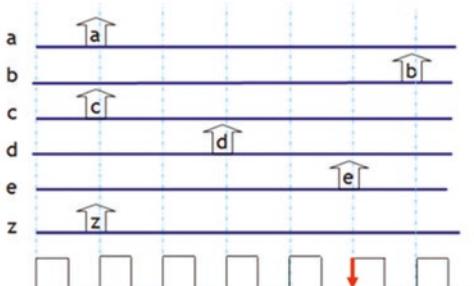
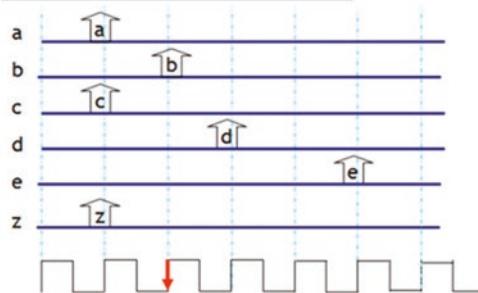
sequence ab;
  a ##[1:5] b;
endsequence

sequence cde;
  c ###2 d ###2 e;
endsequence

sequence abcde;
  ab intersect cde;
endsequence

property isect;
  @ (posedge clk) z |> abcde;
endproperty

```



*property 'isect' FAILs because even though both 'ab' and 'cde' do meet their requirements, they don't end at the same time.*

*property 'isect' FAILs because even though both 'ab' and 'cde' to meet their requirements, they don't end at the same time.*

Fig. 14.72 Seq1 “intersect” seq2: application

#### Specification:

See that dataRead is asserted within 4 clocks after a rising edge on Retry.

```

`define true 1'b1

property retryCheck;
  @ (posedge clk) $rose(Retry) |> `true[*1:4]      intersect
    (Retry ##[1:$] $rose(dataRead)) ;
endproperty

```

If the subsequence “(Retry ##[1:\$] \$rose(dataRead))” does not match within 4 clocks, the sequence “`true[\*1:4]” will end and the property will Fail.

Fig. 14.73 “intersect”: interesting application

`$rose(dataRead)` should occur anytime (`##[1:$]`) after “retry” has been asserted. Now, recall that “intersect” requires both the LHS and RHS to be of “same” length. If `$rose(dataRead)` does *not* arrive in four cycles, the RHS will continue to execute beyond four clocks. But since ``true[*1:4]` has now completed and since “intersect” requires both sides to complete at the same time, the assertion will fail.

If `$rose(dataRead)` does occur within four clks, the property will *pass*. Why? Let us say `$rose(dataRead)` occurs on the third clock. That sequence will end and at the same time ``true[*1:4]` will end as well, since it is ``true` anytime within the four clocks. This satisfies the requirements of “intersect” and the property passes.

So, what is the practical use of such a property. Any time you want to contain a large sequence to occur within a certain time period, it is very easy to use the above technique. A large sequence may have many time domains and temporal complexities, but with the above method, you can simply superimpose ``true` construct with “intersect” to achieve the desired result.

### first\_match(Seq)

- matches only the first of possibly multiple matches of the eval of Seq.
- useful for detecting the first occurrence in a *delay range*.

#### 14.18.18 Application: `first_match`

```
sequence bcORef;
  ( ##[2:5] (b && c) or
    ##[2:5] (e && f))
);
endsequence

property fms;
  first_match (bcORef) |=> $rose(a);
endproperty

baseP: assert property (@(posedge clk) d |-> fms) else gotoFail;
coverP: cover property (@(posedge clk) d |-> fms) gotoPass;
```

```
# run -all
# 5 CLK # 1 :: clk=1 d=0 b=0 c=0 e=0 f=0 a=0
#15 CLK # 2 :: clk=1 d=1 b=0 c=0 e=0 f=0 a=0
#25 CLK # 3 :: clk=1 d=0 b=0 c=0 e=0 f=0 a=0
#35 CLK # 4 :: clk=1 d=0 b=1 c=1 e=0 f=0 a=0
#45 CLK # 5 :: clk=1 d=0 b=1 c=1 e=0 f=0 a=1
#      45  property fms PASS
```

*On the first match of (b && c), the property looks for \$rose(a) the next clock; finds it and **PASSes***

Fig. 14.74 “first\_match”: application

```

#      55 CLK # 6 :: clk=1 d=1 b=0 c=0 e=0 f=0 a=0
#      65 CLK # 7 :: clk=1 d=0 b=0 c=0 e=1 f=1 a=0
#      75 CLK # 8 :: clk=1 d=0 b=1 c=1 e=0 f=0 a=0
#      85 CLK # 9 :: clk=1 d=0 b=0 c=0 e=0 f=0 a=1
# 85 property fms PASS

```

*After  $d==1$ ; ( $e \&& f$ ) is found to be true but not in the required [2:5] clock range; the property next finds ( $b \&& c$ ) to be true and \$rose(a) the next clock; so it PASSes*

```

#      95 CLK # 10 :: clk=1 d=1 b=0 c=0 e=0 f=0 a=0
#     105 CLK # 11 :: clk=1 d=0 b=0 c=0 e=0 f=0 a=0
#     115 CLK # 12 :: clk=1 d=0 b=0 c=0 e=1 f=1 a=0
#     125 CLK # 13 :: clk=1 d=0 b=1 c=1 e=0 f=0 a=0
# 125 property fms FAIL
#     135 CLK # 14 :: clk=1 d=0 b=0 c=0 e=0 f=0 a=1

```

*( $e \&& f$ ) is true 2 clocks after  $d==1$ ; but \$rose(a) is not true 1 clock later. So the property FAILs. Note that ( $b \&& c$ ) is true 3 clocks after  $d==1$  and \$rose(a) true 1 clock later. But since ( $e \&& f$ ) was true FIRST, that \$rose(a) had to be true the next clock.*

Fig. 14.74 (continued)

#### 14.18.18 first\_match

##### first\_match(Seq)

- matches only the first of possibly multiple matches of the eval of Seq.
- useful for detecting the first occurrence in a delay range.

#### 14.18.19 Application: first\_match

In Fig. 14.74, property “fms” says that on the first\_match of “bcORef,” “a” should rise. As you notice, the sequence “bcORef” has many matches because of the range operator and an “or.” As soon as the *first* match of “bcORef” is noticed, the property

**application**

The first time PCI bus goes IDLE, the state machine should transition to busIdle state.

```
sequence busIdleCheck;
  (##[2:$] (frame_ && irdy_));
endsequence

property fms;
  @(posedge clk) first_match (busIdleCheck) |-> (state ==
busIdle);
endproperty
baseP: assert property (fms);
```

**Fig. 14.75** “first\_match” operator: application

looks for \$rose(a). In the top log, that is the case, and the property passes. *Note that the rest of the matches are now ignored.* In the bottom log, \$rose(a) does not occur and the property fails – even though (and as noted in the log, of Fig. 14.74), (b && c) is indeed true three clocks after d==1 and even though the fact that this is an “or,” the first\_match looks for the very first match of either of the sequences in “bcORef” and looks for \$rose(a) right after that. So, as soon as (e && f) is true, the property looks for \$rose(a) – which does not occur, and the property fails.

Figure 14.75 application clarifies \$first\_match further. This is the classic PCI bus protocol application. As the figure shows, the first-time frame\_ && irdy\_ are high (de-asserted) that the bus goes into IDLE state. Note that once frame\_ and irdy\_ are de-asserted, the bus would remain in the IDLE state for a long time. But we want the very first time that the bus transaction ends (indicated by frame\_ && irdy\_ high) that the bus goes into the IDLE state. We do not want to evaluate any further busIdle conditions.

So, what would happen if you removed “first\_match” from the above property (Fig. 14.75)? The property will continue to look for state == busidle every clock that frame\_ && irdy\_ is high. Those will be totally redundant checks.

Note that in all the examples above, we have used first\_match ( ) in the antecedent. Why? Because *the consequent (RHS) of a property behaves exactly like first\_match by default.* The consequent is not evaluated once its first match is found (without the use of first\_match). But the antecedent will keep firing every time there is a match of its expression.

*Hence, it makes sense to use first\_match as part of antecedent sequence.*

```
not (property_expr);
```

- If the property\_expr evaluates to True, then the not (property\_expr) evaluates to False.
- If the property\_expr evaluates to False, then the not (property\_expr) evaluates to True.

```
sequence cde;
  c ###1 d ###1 e;
endsequence
```

```
property nots;
  @(posedge clk) a |-> (not(cde));
endproperty
```

```
baseP: assert property (nots) else
  gotoFail;
coverP: cover property (nots)
  gotoPass;
```

If 'cde' matches, the property fails. If 'cde' does not match, the property passes

*sequence 'cde' fails; so property 'nots' PASSES*

```
#15 CLK # 2 :: clk=1 a=1 b=0 c=1 d=0 e=0
#25 CLK # 3 :: clk=1 a=0 b=0 c=0 d=0 e=0
# 25 property nots PASS
```

*sequence 'cde' passes; so property 'nots' FAILS*

```
115 CLK # 12 :: clk=1 a=1 b=0 c=1 d=1 e=1
125 CLK # 13 :: clk=1 a=0 b=0 c=0 d=1 e=0
135 CLK # 14 :: clk=1 a=0 b=1 c=1 d=0 e=1
135 property ab not cde FAIL
```

Fig. 14.76 “not” operator: basics

#### Specification:

Once 'req' is asserted that you must get an 'ack' -*before*- the next request.

```
property strictlyOneAck;
  @(posedge clk) $rose(req) |=> (not (!lack[*0:$] ##1 $rose(req) ) );
endproperty
strictlyOneAckP: assert property (strictlyOneAck)
  else $display($stime,,,"t Error: strictlyOneAck FAIL");
```

```
KERNEL:      0  clk=1 req=0 ack=0
KERNEL: 10000  clk=1 req=1 ack=0
KERNEL: 20000  clk=1 req=0 ack=0
KERNEL: 30000  clk=1 req=0 ack=0
KERNEL: 40000  clk=1 req=1 ack=0
KERNEL: 40000  Error: strictlyOneAck FAIL
KERNEL: 50000  clk=1 req=0 ack=1
```

Fig. 14.77 “not” operator: application

### 14.18.20 not Operator

The “not” operator seems very benign. However, it could be easily misinterpreted because we are all wired to think positively – correct?

Figure 14.76 shows the use of “not.” Whenever “cde” is true, the property will fail because of “not” and pass if “cde” is not true.

Let us look at an interesting and very useful application.

Application in Fig. 14.77 is a very useful application. The specification says that once req is asserted (active high), we must get an ack *before* getting another request. Such a situation occurs in many designs.

Let us examine the assertion. Property strictlyOneAck says that when “req” is asserted (active high),!ack[\*0:\$] remains low until \$rose(req). If this matches, then the property fails (because of the “not”).

In other words, we are checking to see that ack remains low until the next req, meaning if ack does go high before req arrives, the sequence (!ack[\*0:\$] ##1 \$rose(req)) will fail and the “not” of it will make it pass. That is the correct behavior since we *do* want an ack before the next req.

Or looking at it conversely (and as shown in the log), if “ack” does remain low until the next “req” arrives, the sequence (!ack[\*0:\$] ##1 \$rose(req)) will pass and the “not” of it will fail. This is correct also, because we do not want ack to remain low until next req arrives. We want “ack” to arrive before the next “req” arrives.

```
if (expression) property_expr1;
OR
if (expression) property_expr1 else property_expr2;
```

```
property ife;
@(posedge clk) a ##1 (b || c) [->1] |->
  if (b)
    ##1 d
  else
    ##1 e;
endproperty

baseP: assert property (ife) else gotoFail;
coverP: cover property (ife) gotoPass;
```

The property reads as follows::

@(posedge clk) ‘a’ followed by at least one ‘b’ OR ‘c’;

implies (|->)

that if ‘b’ is true than 1 clock later ‘d’ is true else 1 clock later ‘e’ is true.

```
5 CLK # 1 :: clk=1 a=0 b=0 c=0 d=0 e=0
15 CLK # 2 :: clk=1 a=1 b=0 c=0 d=0 e=0
25 CLK # 3 :: clk=1 a=0 b=0 c=0 d=0 e=0
35 CLK # 4 :: clk=1 a=0 b=1 c=0 d=0 e=0
45 CLK # 5 :: clk=1 a=0 b=0 c=0 d=1 e=0
45 property PASS
```

```
55 CLK # 6 :: clk=1 a=1 b=0 c=0 d=0 e=0
65 CLK # 7 :: clk=1 a=0 b=0 c=0 d=0 e=0
75 CLK # 8 :: clk=1 a=0 b=0 c=0 d=0 e=0
85 CLK # 9 :: clk=1 a=0 b=0 c=1 d=0 e=0
95 CLK # 10 :: clk=1 a=0 b=0 c=0 d=0 e=0
95 property FAIL
```

Fig. 14.78 If...else

May seem a bit strange, and this property can be written in many different ways, but this will give you a good understanding of how negative logic can be useful.

### 14.18.21 **if** (*Expression*) *property\_expr1* Else *property\_expr2*

“if”...“else” constructs are similar to their counterpart in procedural languages and obviously very useful. As Fig. 14.78 annotates, we are making a decision in consequent based on what happens in the antecedent. The property “if” states that on “a” being true, either “b” or “c” should occur at least once, *any time* one clock after “a” (non-consecutive GoTo). If this antecedent is true, the consequent executes. Consequent expects “d” to be true if “b” is true and “e” to be true if “b” is false or “c” is true.

The simulation log in the bottom left of Fig. 14.78 shows that at time 15, “a==1” and one clock later “b” is true as required. Since “b” is true, “d” is true one clock later at time 45. Everything works as required and the property passes. In the bottom-right simulation log, “a==1” at time 55 and “c” goes true at 85. This would require “e” to be true one clock later, but it’s not and the property fails. This is just but one way to use if-else.

### 14.18.22 “*iff*” and “*implies*”

p **iff** q is an equivalence operator. This property is true *iff* (if and only if) properties “p” and “q” are both true. When “p” and “q” are Boolean expressions “e1” and “e2,” then e1 **iff** e2 is equivalent to e1  $\leftrightarrow$  e2.

p **implies** q is an implication operator. So, what is the difference between “implies” and the implication operator “ $| ->$ ”? In case of p  $| ->$  q, the evaluation of “q” starts at the match of “p.” In case of p **implies** q, *both* “p” and “q” start evaluating at the *same* time, and the truth results are computed using the logical operator “implies.” There is no notion of a match of antecedent to trigger the consequent.

For example:

```
x ##2 y |-> a ##2 b;
```

vs.

```
x ##2 y implies a ##2 b;
```

In the case of implication operator “ $| ->$ ,” evaluation of consequent “a ##2 b” starts at the match of antecedent x ##2 y’. Property fails if the consequent does not hold. If antecedent does not match, property waits for the next match of the antecedent (and vacuously pass).

**\$onehot (<expression>)**

*Returns True if only one bit of the expression is a ‘1’ (high).*

```
property bgcheck;
  @(posedge clk) bgack |->
$onehot(busgnt);
endproperty
```

```
# run -all
#      5  clk=1 bgack=1 busgnt=xxxxxxxx
#      5  property bgcheck FAIL
#
#      15 clk=1 bgack=1 busgnt=00000001
#      15 property bgcheck PASS
#
#      25 clk=1 bgack=1 busgnt=x0000001
#      25 property bgcheck PASS
#
#      35 clk=1 bgack=1 busgnt=z0000001
#      35 property bgcheck PASS
#
#      45 clk=1 bgack=1 busgnt=11111111
#      45 property bgcheck FAIL
#
#      55 clk=1 bgack=1 busgnt=00000000
#      55 property bgcheck FAIL
```

**\$onehot0 (<expression>)**

*Returns True if all bits of the expression are ‘0’ OR only one bit of the expression is a ‘1’.*

**Fig. 14.79** \$onehot and \$onehot0

In the case of “implies,” evaluation of *both* “`x ##2 y`” and “`a ##2 b`” starts at the *same* clock tick, and the property fails if *either* of the “`x ##2 y`” or “`a ##2 b`” fail.

## 14.19 System Functions and Tasks

### 14.19.1 \$onehot and \$onehot0

\$onehot ( ) and \$onehot0 ( ) are explained as shown in Fig. 14.79. Note that if the expression is all “Z” or all “X,” \$onehot or \$onehot0 will fail. \$onehot ( ) will not fail if there are “x”s and “z”s on the bus and – at least – one “1.” Similarly, \$onehot0 will not fail if there are some “x”s and “z”s but at least one “1.”

<b>\$isunknown (&lt;expression&gt;)</b>	Returns True if any bit of the expression is 'X' or 'Z'
<pre>property ucheck;   @(posedge clk) bgack  &gt; \$isunknown(busgnt); endproperty</pre>	<pre># run -all #      15 clk=1 bgack=1 busgnt=z0000001 #      15 property bgcheck PASS # #      25 clk=1 bgack=1 busgnt=x0000001 #      25 property bgcheck PASS # #      35 clk=1 bgack=1 busgnt=00000001 #      35 property bgcheck FAIL # #      45 clk=1 bgack=1 busgnt=z1111111 #      45 property bgcheck PASS # #      55 clk=1 bgack=1 busgnt=x1111111 #      55 property bgcheck PASS # #      65 clk=1 bgack=1 busgnt=11111111 #      65 property bgcheck FAIL</pre>

**Fig. 14.80** \$isunknown

### 14.19.2 \$isunknown

\$isunknown passes if the expression is unknown ("X" or "Z"). In other words, if the expression is not unknown, then the property will fail! Hence, if you do want a failure on detection of an unknown ("X" or "Z"), then you have to negate the result of \$isunknown. Simple but easy to miss.

Simulation log in Fig. 14.80 clarifies the concept. Property "ucheck" states that if "bgack" is true, then the "busgnt" is unknown. What? This is simply to show what happens if you use \$isunknown without a "not."

### 14.19.3 \$countones

\$countones is a very simple but powerful feature. Note that this system function can be used in a procedural block as well as in a concurrent property/assertion.

Figure 14.81 shows an application which states that if there is a bus grant acknowledge (bgack), then there can be only one bus grant (busgnt) active on the bus. Note that we are using \$countones in a procedural block in this example. Note

<b>\$countones (&lt;expression&gt;)</b>	Counts the number of '1's in a bit vector expression.  An 'x' or a 'z' is NOT counted towards the number of 1's
<b>application</b> <b>SPECIFICATION:</b> If Bus Grant Ack (bgack) is asserted there can only be 1 Bus Gnt (busgnt ). <pre>always @(posedge clk) begin   if (bgack)     begin       cones = \$countones(busgnt);       if (cones &gt; 1    cones = 0),         \$display(\$stime,,"`t`t FAIL:Number of 1's = %0d",cones);       else         \$display(\$stime,,"`t`t PASS:Number of 1's = %0d",cones);     end end</pre>	
<pre>#      5  clk=1 bgack=1 busgnt=xxxxxxxx #      5      FAIL:Number of 1's = 0 #     15  clk=1 bgack=1 busgnt=00000001 #     15      PASS:Number of 1's = 1 #     25  clk=1 bgack=1 busgnt=00000000 #     25      FAIL:Number of 1's = 0 #     35  clk=1 bgack=1 busgnt=11111111 #     35      FAIL:Number of 1's = 8</pre>	

Fig. 14.81 \$countones

also that if the entire “busgnt” is unknown (“X”) or tristate (“Z”), the assertion will fail.

#### 14.19.4 \$countbits

`$countbits(e, list_of_control_bits)` has the following arguments:

‘e’ = a bit vector, meaning a packed or an unpacked integral expression

‘list\_of\_control\_bits’ = `$countbits` returns the number of bits of its argument bit vector having the value of 1 of the control bits. These control bits can only have one of four values, namely, 1'b0, 1'b1, 1'bz, and 1'bx. You must specify at least one control bit and repeated control bits are ignored. You can have more than one control bit in the “list\_of\_control\_bits.”

For example:

If you want to make sure that none of the bits of a bus are in floating state (1'bz) when the bus is driven, you can write the property as follows:

```
property checkFloat;
@(posedge clk) Busdriven |-> ($countbits (bus, 1'bz) == 0);
endproperty
```

This property says that when “Busdriven” signal is high, none of the bits in “bus” are 1’bz. The way to read the property is this: the \$countbits(bus) returns the number of bits in “bus” that are in 1’bz state and sees that the count is zero, thus verifying that none of the bits are in 1’bz state.

Another example:

Make sure that none of the bits in a given vector are unknown. The following property can also be written using the system function \$isunknown. The following accomplishes the same with \$countbits. The example is to highlight that you can have more than one control bit:

```
property checkUnknown;
@(posedge clk) memread |-> ($countbits (data, 1'bx, 1'bz) == 0);
endproperty
```

\$countbits(data) returns a count of bits that are in 1’bx and 1’bz state. If this count is zero, then there are no unknown bits.

**\$assertoff (level,[list of module, instance or assertion\_identifier]);**

*Sassertoff stops the checking of all specified assertions until a subsequent \$asserton.*

*Note: If an assertion is already executing, it won't be affected.*

**\$assertkill (level,[module/module instance or assertion\_identifier]);**

*Sassertkill shall abort execution of any currently executing specified assertions and then stop the checking of all specified assertions until a subsequent \$asserton*

**\$asserton (level,[module/module instance or assertion\_identifier]);**

*Sasserton shall re-enable execution of all specified assertions.*

*By default assertions are ON with an 'assert' statement*

*'level' = 0 turns on/off assertions at ALL levels under the given module/instance  
= m (m>0) turns on/off assertions only at 'm' levels of hierarchy below the specified module / instance level.*

*assertion\_identifier :: Name of the property or the label used with 'assert'*

*module, instance name :: Can be relative or full hierarchical*

Fig. 14.82 \$asserton, \$assertoff, and \$assertkill

*Example shows that assertions are killed off during ‘reset\_’; are turned ON after reset\_ and are turned off when a machine check exception is detected.*

```
module assertion_control(input reset_, machinecheck_exception);
    always @ (negedge reset_) $assertkill(0,top.pcim, top.axim);
    always @ (posedge reset_) $asserton(0,top.pcim, top.axim); ← 'instance' level
    //always @ (negedge reset_) $assertkill(0,top);
    //always @ (posedge reset_) $asserton(0,top); ← 'module' level

    always @ (machinecheck_exception) $assertoff(0,top.datamodule.array);
    always @ (machinecheck_ISR_return) $asserton(0,top.datamodule.array);

endmodule
```

Fig. 14.83 Application: project-wide assertion control

### 14.19.5 *\$assertoff, \$asserton, and \$assertkill*

There are many situations when you want to have a global control over assertions at both module and instance levels. Recall that “disable iff” provides you a local control directly at the source of the assertion.

As noted in Fig. 14.82, \$assertoff temporarily turns off execution of all assertions. Note that if an assertion is under way when \$assertoff is executed, the assertion *will not* be killed. You restart assertion execution on a subsequent invocation of \$asserton. \$assertkill will kill *all* assertions in your design *including* already executing assertion. And it will not automatically restart when the next assertion starts executing. It will restart executing only on the subsequent \$asserton. \$asserton is the default. It is required to restart assertions after a \$assertoff or \$assertkill.

The following is a typical application deployed by projects to suppress assertion checking during “reset” or an exception (Fig. 14.83).

## 14.20 Multiply Clocked Sequences and Properties

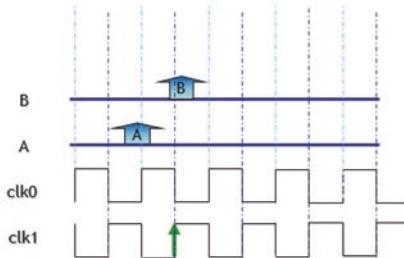
There are hardly any designs anymore that work only on a single clock domain. So far, we have seen properties that work off a single clock. But what if you need to check for a temporal domain condition that crosses clock boundaries? The so-called CDC (clock domain crossing) issues can be addressed by multiple clock assertions.

We will thoroughly examine how a property/sequence crosses clock boundary. What is the relationship between these two (or more) clocks? How are sampling edges evaluated once you cross the clock domain? Note that in a singly clocked system, the sampling edge is always single, mostly posedge or negedge of clock. Since there are two (or more) clocks in multiply clocked system, we need to

Multiply-paced sequences are built by concatenating singly-paced subsequences using the delay concatenation operators `##1` and `##0`.

*"##1 @(posedge clk1)" here does -not- necessarily mean a delay of one clock.*

It means on a match of 'A' @**(posedge clk0)**, the **##1 moves the time to the nearest strictly subsequent posedge clk1** and the sequence ends at that point with a match of B.



**Fig. 14.84** Multiply clocked sequences

understand how the sampling edges cross boundary from one clock to another. I think it is best to fully understand the fundamentals before jumping into applications.

### **14.20.1 Multiply Clocked Sequences**

The timing diagram in Fig. 14.84 shows that at (posedge clk0), “A” is true. The clocks are out of phase, so the very next clock edge of clk1 is half a clock delayed from posedge clk0. At the posedge of clk1, “B” is sampled true and the sequence “mclocks” passes. The point here is that “##1 @ (posedge clk1)” waited only for  $\frac{1}{2}$  clk1 and not a full clk1 because the *very next nearest strictly subsequent posedge clk1* arrived in  $\frac{1}{2}$  clock period. The next clock can come in any time after clock0 and that will be the “very next” edge taken as the sampling edge for that subsequence.

### ***Important Note***

The LRM 2005 requirement of ##1 between two subsequences have been removed from 1800-2009/2012. In the 2009/2012 standard you can have both ##1 and ##0 between two subsequences with different clocks. More on this is coming up.

As LRM puts it, multiclocked sequences are built by concatenating singly clocked subsequences using the single-delay concatenation operator `##1` or the zero-delay concatenation operator `##0`. The single delay indicated by `##1` is understood to be from the end point of the first sequence, which occurs at a tick of the first

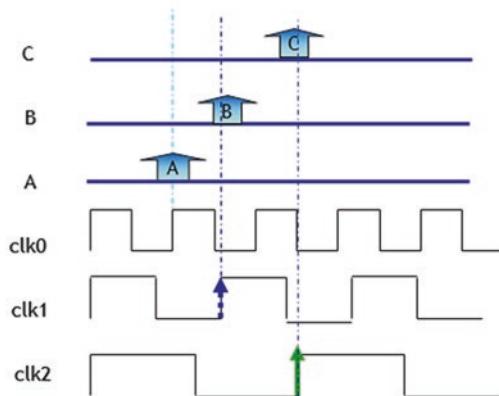
```

property mclocks;
  @(posedge clk1) b and @ (posedge clk2) c;
endproperty

baseP: assert property (@(posedge clk0) a |=> mclocks) else gotoFail;
coverP: cover property (@(posedge clk0) a |=> mclocks) gotoPass;

```

*'B' and 'C' must be true at immediate next posedge of clk1 and clk2 respectively after the posedge of clk0*



**Fig. 14.85** Multiply clocked properties: “and” operator between two different clocks

clock, to the nearest strictly subsequent tick of the second clock, where the second sequence begins. The zero delay indicated by `##0` is understood to be from the end point of the first sequence, which occurs at a tick of the first clock, to the nearest possibly *overlapping* tick of the second clock, where the second sequence begins.

### 14.20.2 Multiply Clocked Properties: “and” Operator

The concept of “and” of two singly clocked properties have been discussed before. But what if the clocks in the properties are different? The important thing to note here is the concept of the very next strictly subsequent edge. In Fig. 14.85, at the posedge of `clk0`, “`a`” is sampled high. That triggers the consequent that is an “and” of “`b`” and “`c`.“ Note that “`b`” is expected to be true at the very next posedge of `clk1` (after the posedge of `clk0`). *In other words, even though there is a non-overlapping operator in the property, we do not quite wait for one clock.* We simply wait for the very next posedge of `clk1` to check for “`b`” to be true. The same story applies to “`c`.“ “`c`” is expected to be true at the very next posedge of `clk2` (after the posedge of `clk0`). When both “`b`” and “`c`” occur as shown in Fig. 14.85, the property will pass.

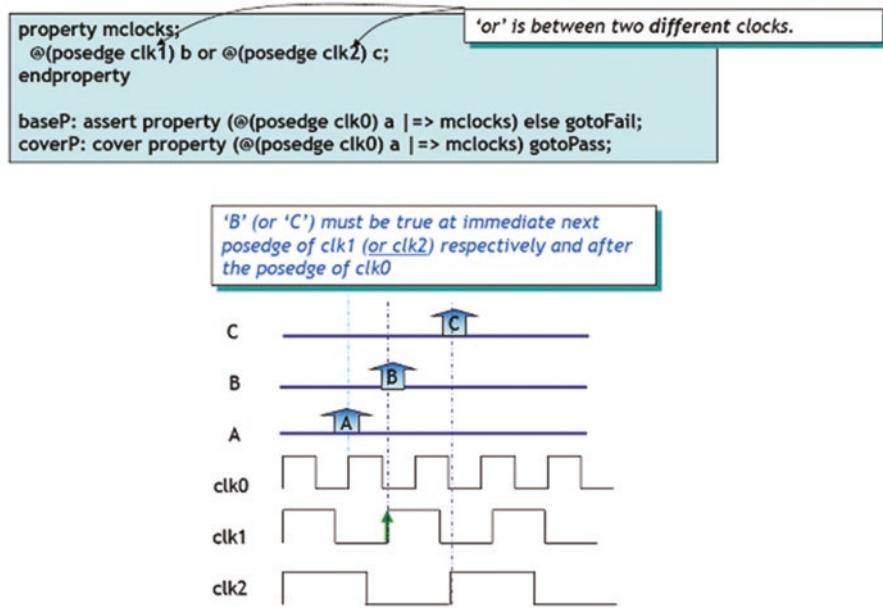


Fig. 14.86 Multiply clocked properties: “or” operator

As with the singly clocked “and,” the assertion passes at the match of the longest sequence “c.”

### 14.20.3 Multiply Clocked Properties: “or” Operator

All the rules of “and” apply to “or” – except as in singly clocked properties – when either of the sequence (i.e., either the LHS or RHS of the operator) passes that the assertion will pass. The concept of “the very next strictly subsequent clock edge” is the same as with “and.”

Please refer to Fig. 14.86 for better understanding of “or” of multiply clocked properties. The property passes when either @ (posedge clk1) “b” or @ (posedge clk2) “c” occurs. In other words, if @ (posedge clk2) “c” occurs before @ (posedge clk1) “b,” the property will pass at @ (posedge clk2) “c.” In our case, “@(posedge clk1) b” occurs first and the property passes at posedge clk1.

### 14.20.4 Multiply Clocked Properties – “not”: Operator

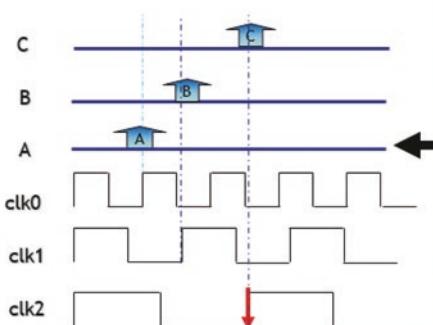
“not” is an interesting operator when it comes to multiply clocked properties.

```

property mclocks;
  @(posedge clk1) b and (not (@(posedge clk2) c));
endproperty

baseP: assert property (@(posedge clk0) a |=> mclocks) else gotoFail;
coverP: cover property (@(posedge clk0) a |=> mclocks) gotoPass;

```



```

# run -all
#
# 0 clk0,1,2=111 a=0 b=0 c=0
# 10 clk0,1,2=101 a=1 b=0 c=0
# 16 clk0,1,2=010 a=1 b=1 c=0
# 20 clk0,1,2=110 a=1 b=1 c=0
# 24 clk0,1,2=101 a=1 b=1 c=1
# 24 property mclocks FAIL
#
# 30 clk0,1,2=101 a=1 b=1 c=0
# 32 clk0,1,2=111 a=1 b=1 c=0
# 40 clk0,1,2=100 a=0 b=0 c=0
# 48 clk0,1,2=011 a=1 b=1 c=0
# 48 property mclocks PASS

```

Fig. 14.87 Multiply clocked properties: “not” operator

<pre> property mclocks;   @(posedge clk0) A  &gt;     if (D) @(posedge clk0) B; endproperty </pre>	<i>This is equivalent to</i> <code>@(posedge clk0) A  &gt; if (D) B;</code>
<pre> property mclocks;   @(posedge clk0) A  &gt;     if (D) @(posedge clk0) B     else @(posedge clk0) (~B); endproperty </pre>	<i>This is equivalent to</i> <code>@(posedge clk0) A  &gt; if (D) B else (~B);</code>
<pre> property mclocks;   @(posedge clk0) A  &gt;     if (D) @(posedge clk0) B     ##1 @(posedge clk1) Z     else @(posedge clk0) (~B); endproperty </pre>	<i>This is equivalent to</i> <code>@(posedge clk0) A  &gt; if (D) B ##1 @(posedge clk1) Z else (~B);</code>

Fig. 14.88 Multiply clocked properties: clock resolution

The assertion in Fig. 14.86 works as follows. At posedge clk0, “a” is true which triggers the consequent mclocks. The property mclocks specifies that @ posedge clk1 “b” needs to be true and “c” should – not – be true @ posedge clk2. The timing diagram shows that “a” is true at posedge clk0. At the very next subsequent edge of clk1, “b” should be true, and since it is indeed true, the property moves along. Because of an “and,” it looks for “c” to be *not* true at the very next subsequent

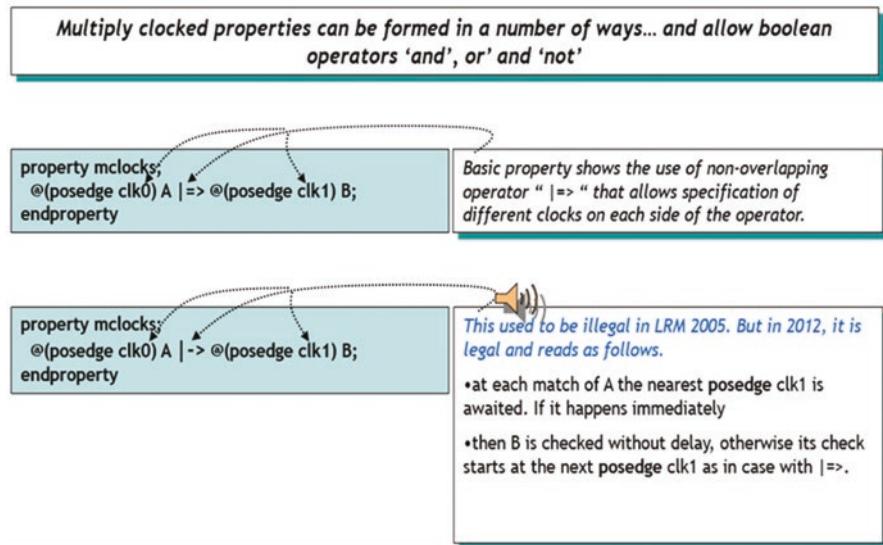


Fig. 14.89 Multiply clocked properties: legal and illegal conditions

posedge clk2 . Well, “c” is indeed true, but since we have a “not” in front of @ (posedge clk2), the property will fail. The concept of “not” is the same as that of singly clocked properties except for the edge of the clock when it is evaluated. The simulation log clarifies the concept (Fig. 14.87).

#### 14.20.5 Multiply Clocked Properties: Clock Resolution

Figure 14.88 explains the rules on how clocks flow from one part of the property to another.

#### 14.20.6 Multiply Clocked Properties: Legal and Illegal Conditions

In 2012 LRM, the multiclocked overlapping implication  $| ->$  has the following meaning: at the end of the antecedent, the nearest tick of the consequent clock is awaited. If the consequent clock happens at the end of the antecedent, the consequent starts checking immediately. Otherwise, the meaning of the multiclocked overlapping implication is the same as the meaning of the multiclock non-overlapping implication (Fig. 14.89).

**Local variables are dynamic variables.**

**They are dynamically created when needed within an instance of a sequence and removed when the end of the sequence is reached.**

**'local vars' is one of the most powerful features of SVA language because it allows checking of complex pipelined behavior of the design.**

**application**

```

sequence rdC;
  ##[1:5] rdDone;
endsequence

sequence dataCheck;
  int local_data;

  (rdC,local_data=rData)  ##5  (wData == (local_data+'hff));

endsequence

baseP: assert property (@(posedge clk) RdWr |> dataCheck) else gotoFail;

```

*a new copy of local\_data is created with every instance of dataCheck*

**sequence dataCheck reads as ::**

*on matching 'rdC', store rData in the local var called local\_data and ##5 clocks later wData must match local\_data+'hff*

*Note that dataCheck is triggered when 'RdWr' is true. 'RdWr' can be true every clock and dataCheck would be triggered every clock. For every trigger of dataCheck, a new copy of local\_data is created which will store rData and check for wData 5 clocks later.*

Fig. 14.90 Local variables: basics

## 14.21 Local Variables

Local variable is a feature you are likely to use very often. They can be used both in a sequence and a property. They are called *local* because they are indeed local to a sequence and are not visible or available to other sequences or properties. Of course, there is a solution to this restriction, which we will study further into a section that follows.

Figure 14.90 points out key elements of a local var. The most important and useful aspect of a local variable is that it *allows multi-threaded application and creates a new copy of the local variable with every instance of the sequence in which it is used*. User does not need to worry about creating copies of local variables with each invocation of the sequence. Above specification says that whenever "RdWr" is sampled high at posedge clk, on completion of rdC, "wData" is compared with "rData

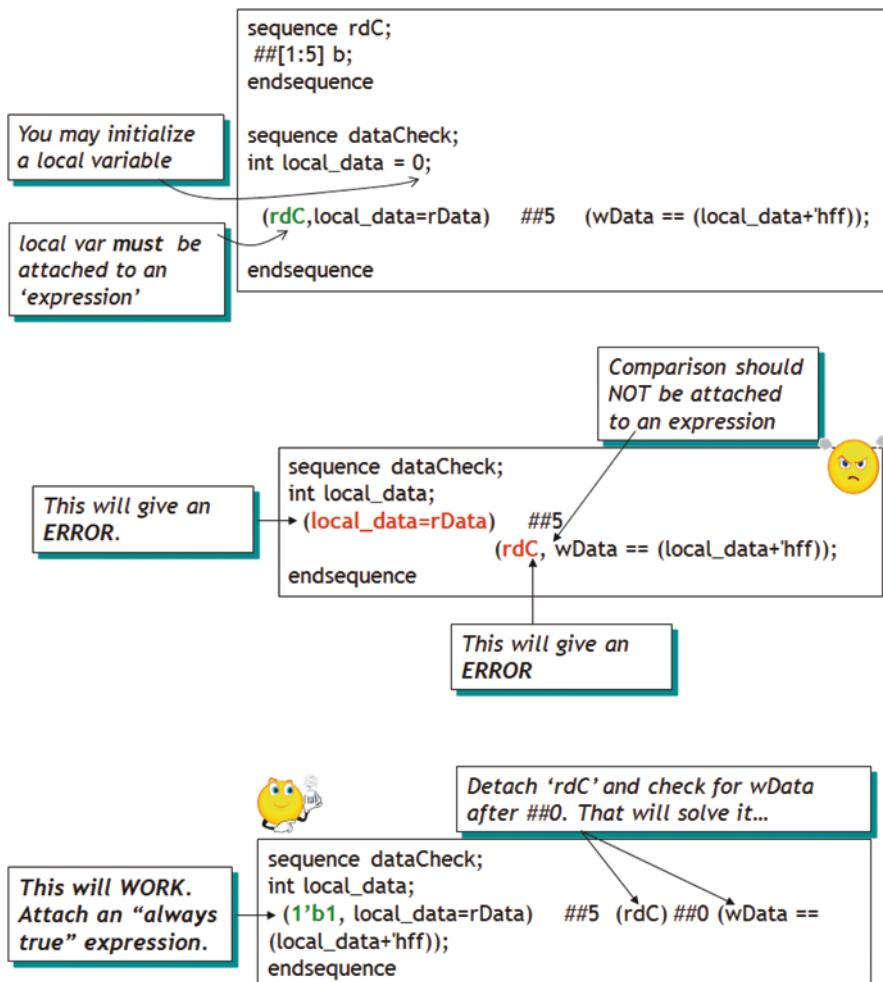


Fig. 14.91 Local variables: do's and do not's

+ 'hff' five clocks later. The example shows how to accomplish this specification. Local variable "int local\_data" stores "rData" at posedge of clk and then compares it with wData five clocks later. Note that "RdWr" can be sampled true at every posedge clk. Sequence "data\_check" will enter every clock, create a new copy of local\_data, and create a new pipelined thread that will check for local\_data+'hff with "wData" five clocks later.

**Important Note** The sampled value of a local variable is defined as the current value (and not the value in preponed region).

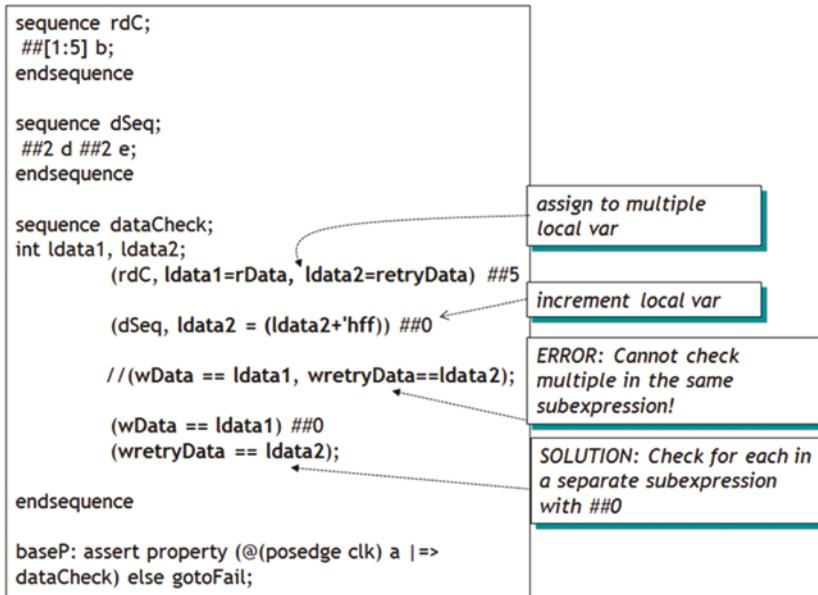


Fig. 14.92 Local variables: further nuances

Moving along, Fig. 14.91 shows other semantics of local variables. Pay close attention to the rule that local variable must be “attached” to an expression, while comparison cannot be attached to an expression!?

As shown in Fig. 14.91, a local variable must be attached to an expression when you store a value into it. But when you compare the value stored in a local variable, it must not be attached to an expression.

In the topmost example, “local\_data=rData” is attached to the sequence “rdC.” In other words, assignment “local\_data=rData” will take place only on completion of sequence “rdC.” Continuing with this story of storing a value into a local variable, what if you do not have anything to attach to the local variable when you are storing a value? Use 1'b1 (always true) as an expression. That will mean whenever you enter a sequence, the expression is always true, and you can store the value in the local variable.

Note that local variables do *not* have default initial values. A local variable without an initialization assignment will be unassigned at the beginning of the evaluation attempt. The expression of an initialization assignment to a given local variable may refer to a previously declared local variable. In this case the previously declared local variable must itself have an initialization assignment, and the initial value assigned to the previously declared local variable will be used in the evaluation of the expression assigned to the given local variable. More on this later.

Ok, so what if you want to compare a value on an expression being true? As shown in the figure, you can indeed accomplish this by “detaching” the expression as shown. The resulting sequence (the last sequence in Fig. 14.91) will read as “on

entering dataCheck, store rData into local\_data, wait for five clocks and then if “b” (of sequence rdC) is true within five clocks, compare wData with stored local\_data + ‘hff’ .

The following discusses further nuances.

Figure 14.92 describes further rules governing local variables. First, you can assign to multiple local variables, attached to a single expression. Second, you can also manipulate the assigned local data in the same sequence (as is the case for ldata2). But as before, there are differences in assigning to (storing to) local variables and comparing their stored value. *You cannot compare multiple local variable values in a single expression* in a sequence as is the case in the line “// (wData == ldata1, wretryData ==! ldata2) .” This is illegal. Of course, there is always a solution as shown in the figure. Simply separate comparison of multiple values in two subsequences with no delay between the two. The “solution” annotation in the figure makes this clear.

Here are some legal and illegal conditions of local variables:

```
property illegal_legal_declarations;
    data; //ILLEGAL. 'data' needs an explicit data type.

    logic data = 1'b0;
    //LEGAL. Note that unlike SystemVerilog variables, local
variables have no default initial value. Also, the assignment can
be any expression and need not be a constant value

    byte data [ ]; //ILLEGAL - dynamic array type not
allowed.

endproperty
```

Also, you can have multiple local data variable declarations as noted above. And a second data variable can have dependency on the first data variable. *But the first data variable must have an initial value assigned.* Here is an example:

```
property legal_data_dependency;
    logic data = data_in, data_add = data + 16'h
FF; //LEGAL
endproperty

property illegal_data_dependency;
    logic data, data_add = data + 16'FF;
    //ILLEGAL. 'data' used in expression assignment of 'data_
add' is not initialized.
endproperty

sequence illegal_declarations (
```

```
// ILLEGAL: 'local' keyword is not specified with direction.
output logic a,
```

```
// ILLEGAL: default actual argument illegal for inout
local inout logic b, c = 1'b0,
```

```
// ILLEGAL: type must be specified explicitly
local d = expr,
```

```
// ILLEGAL: 'event' type is not allowed
local event e
);
```

Here are some more rules that must be followed.

A special note on the use of method “.triggered” with a local variable. A local variable passed into an instance of a named sequence to which sequence method (.triggered) is applied is not allowed. For example, the following is illegal:

```
sequence check_trdy (cycle_begin);
    cycle_begin ##2 irdy ##2 trdy;
endsequence

property illegal_use_of_local_with_triggered;
    bit local_var;
    524(1'b1, local_var = CB) |-> check_trdy(local_var).
triggered; //ILLEGAL
endproperty
```

### 14.21.1 Application: Local Variables

The application in Figure 14.93 is broken down as follows:

$(\$rose(read), localID=readID)$

On  $\$rose(read)$ , the  $readID$  is stored in the  $localID$ .

$not ((\$rose(read) \&& readID==localID) [*1:$])$

Then we check to see if another read ( $\$rose(read)$ ) occurs and its  $readID$  is the same as the one we stored for the previous read in  $localID$ . We continue to check this consecutively until

$\#\#0 (\$rose(readAck) \&& readAckID == localID)$  occurs.

If the consecutive check does result in a match, that would mean that we did get another  $\$rose(read)$  with the same  $readID$  with which the previous read was issued.

**Once a ‘read’ has been issued, another ‘read’ for the same readID cannot be re-issued until a readAck with the same ID has returned.**

```
property checkRead;
  int localID;
  ($rose(read),localID = readID) |=>
    not ( ($rose(read) && readID==localID) [*1:$] ) ##0
    ($rose(readAck) && readAckID == localID);
endproperty

baseP: assert property (@(posedge clk) (checkRead)) else
  $display($stime,,,"`t property FAIL");
```

Fig. 14.93 Local variables: application

That is a violation of the specs. This is why we take a “not” of this expression to see that it turns false on a match and the property would fail and end.

If the consecutive check does not result in a match until ##0 (\$rose(readAck) && readAckID == localID) arrives, then we may get a readAck with the same readAckID with which the original read was issued. The property will then pass (or fail if readAckID is not equal to localID).

In short, we have proven that once a “read” has been issued that another “read” from the same readID cannot be re-issued until a “readAck” with the same ID has returned.

One more example.

Example: This example shows a simple way to track time. Here, on falling edge of Frame\_, rising edge of IRDY cannot arrive for at least MinTime.

Solution:

```
property FrametoIRDY (integer minTime);
  integer localBaseTime;
  @ (posedge clk) ($fall(Frame_), localBaseTime = $time)
    |=>
    $rose(IRDY) && $time >= localBaseTime + minTime;
endproperty
measureTime: assert property (FrametoIRDY (.minTime
(MINIMUM_TIME)));
```

## 14.22 End Point of a Sequence (.triggered)

Before we learn how .triggered works, here is what has changed in the 1800-2009/2012 standard.

*.triggered is a method on a sequence (that returns true or false).*

*Whenever the end point of a sequence is reached, .triggered will be true -regardless- of when the sequence started.*

- .triggered allows another way to create smaller subsequences leading to more complex ones.*

*Any sequence that will have a method attached to it must have an explicit clock.*

*"Use of a method on an unclocked sequence is illegal".*

### application

```
sequence branch(a ,b ,c ,d);
  @(posedge clk) $fell(a) ##[1:5] $rose(b) ##1 c [=2] ##1 d;
endsequence

property endCycle;
  @(posedge clk) $rose( endBranch ) |=> branch(a, b ,c , d).triggered;
endproperty
```



*The source and destination clocks -must- be the same.*

Fig. 14.94 .triggered: end point of a sequence

*The 2009/2012 standard gets rid of .ended and in place supports .triggered. In other words, .triggered has the same exact meaning as .ended, only that .triggered can be used both where .ended gets used as well as where .triggered was allowed in previous versions. In other words, .triggered can be used in a sequence as well as in procedural block and in level-sensitive “wait” statement.*

The following is from the 2012 LRM:

*IEEE Std 1800-2005 17.7.3 required using the .ended sequence method in sequence expressions and the .triggered sequence method in other contexts. Since these two constructs have the same meaning but mutually exclusive usage contexts, in this version of the standard, the .triggered method is allowed to be used in sequence expressions, and the usage of .ended is deprecated and does not appear in this version of the standard.*

*Note that the entire discussion devoted to .triggered in this chapter applies directly to .ended.*

If you are mainly interested in the *end* of a sequence regardless of when it started, .triggered is your friend. The main advantage of methods that detect the end point of a sequence is that you do *not* need to know the start of the sequence. All you care for is when a sequence ends.

Figure 14.94 shows that behavior. Sequence “branch” is a complete sequence for a branch to complete. It could have started any time. The property endCycle wants to make sure that the “branch” sequence has indeed ended when endBranch flag

goes high. In other words, whenever \$rose(endBranch) is detected to be true that the next clock, branch must end.

Important Note: What if you simply write the assertion as “at the end of ‘branch’ see that endBranch goes high” as in “branch(a, b, c, d) |=> \$rose(endBranch).” What is wrong with that? Well, what if \$rose(endBranch) goes high when the “branch” is still executing? That \$rose(endBranch) would go unnoticed until the end of the sequence “branch.” The .triggered operator would catch this. If endBranch goes high when sequence “branch” is still executing, the property will fail. That is because the property endCycle expects “branch” to have ended when endBranch goes high. Since endBranch could have risen prematurely, the property will see that at \$rose(endBranch) the “branch” sequence has not ended, and the property would fail. The forward-looking property would not catch this.

*Also, note that the clock in both the source and destination sequence must be the same.* But what if you want the source and destination clocks to be different? That is what “.matched” does, soon to be discussed.

Some examples.

Here is an example of .triggered usage in a procedural assignment using level-sensitive control:

```
sequence busGnt;
  @ (posedge clk) req ##[1:5] gnt;
endsequence
initial begin
  wait (busGnt.triggered) $display($stime,,, "Bus Grant
given");
end
```

Note that .matched cannot be used in the above “wait” statement. That is illegal. “.matched” is discussed in next section.

Here is an example that shows the use of .triggered in sequences:

```
sequence abc;
  @ (posedge gclk) a ##[1:5] ##1 b [*5] ##1 c;
endsequence

sequence myseq;
  @ (posedge gclk) d ##1 abc.triggered ##1 !d;
endsequence
```

Finally, the following is *illegal*:

```

logic x,y,z;
//...
Xillegal: assert property (@(posedge clk)
    z |-> (x ## 3).triggered //ILLEGAL

```

Why is this illegal? Recall that .triggered can only be applied to a ““named sequence instance” or a formal argument. (x ## 3) is neither of the two.

The following is *illegal* as well because the clocks on the two sides of the implication differ. They need to be the same:

```

sequence clk1Seq;
    @(posedge clk1) x ##2 y;
endsequence
Zillegal: assert property (@(posedge clk) z |-> clk1Seq.triggered);
//ILLEGAL

```

### 14.22.1 End Point of a Sequence (.matched)

The main difference between .triggered and .matched is that .triggered requires both the source and destination sequences which have the same clock. .matched allows you to have different clocks in the source and destination sequences.

Since the clocks can be different, understanding of .matched gets a bit complicated. But it follows the same rules as that for multiply clocked properties. As shown in Fig. 14.95, sequence “e1” uses “clk” as its clock, while sequence “e2” uses “sysclk” as its clock. Sequence “e2” says that after “reset,” one clock later, “inst” must be true, and one “clk” later (nearest subsequent “clk” edge), sequence “e1” must match (i.e., end) at least once, and one “sysclk” later, branch\_back must be true. This is a very interesting way of “inserting” a .matched (or .triggered for that matter) within a sequence. Sequence “e1” is running on its own and in parallel to the calling sequence. What we really care for is that it matches (ends) when we expect it to.

In Fig. 14.96, sequence “RdS” uses Busclk while the property checkP uses sysclk. “@ (negedge sysclk) RdS.matched” means that at the negedge of sysclk, the sequence RdS (which is running off Busclk) must end. “RdS” could have completed slightly (i.e., when the very preceding posedge of Busclk would have arrived) earlier than the negedge sysclk. That is ok because we are transitioning from Busclk to sysclk (as long as the sequence RdS completes at the *immediately preceding* posedge Busclk).

*.matched is a method on a sequence (that returns true or false).*

*.matched is used when the clocks of the sequences are different.* To reiterate, .ended (or .triggered) can be used only when the clocks of the source and destination sequences are the same.

"Unlike .ended (or .triggered), .matched uses synchronization between the two clocks, by storing the result of the source sequence match until the arrival of the first destination clock tick after the match."

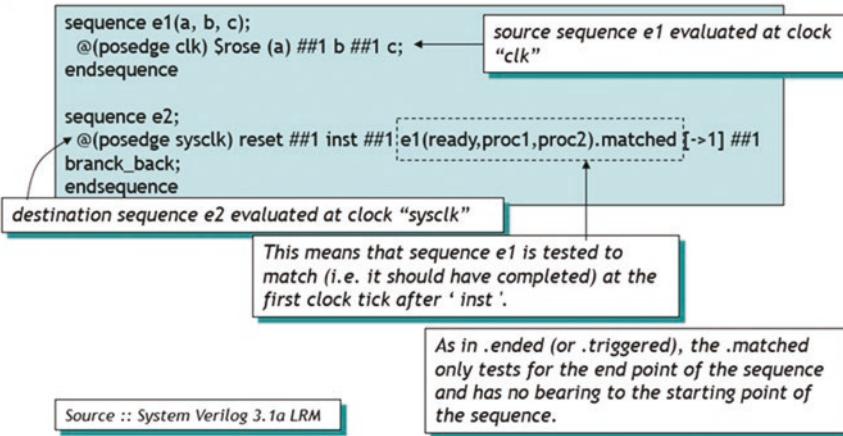


Fig. 14.95 .matched: basics

```

sequence RdS;
  @ (posedge Busclk) $fell (as_) ##1 rd ##[1:5] oe_;
endsequence

property checkP;
  @(negedge sysclk) RdS.matched |=> rdDataLatch ##0 ($isunknown
  (data) == 0);
endproperty
  
```

*The matched value of sequence RdS is sampled at the negedge of sysclk and if it is true (i.e. the sequence has completed) it implies that rdDataLatch is asserted the next negedge sysclk and that the data is not unknown at that clock*

Fig. 14.96 .matched: application

**'expect' statement is a procedural blocking statement that allows waiting on a property evaluation.**

The 'expect' statement accepts the same syntax used to assert a property.

The 'expect' statement can accept a named property as well.

```

initial <--> (or an 'always' block)
begin
    $display($stime,,,"Hello before expect");
    expect (@(posedge clk) c |-> c ##2 d ##2 e);
        $display($stime,,,"texpect pass");
    else
        $display($stime,,,"texpect fail");
    $display($stime,,,"Goodbye after expect");
end
endmodule

```

What would happen if you changed 'expect' with an 'assert'?

```

# 0 Hello before expect
# 5 CLK # 1 :: clk=1 a=0 b=0 c=1 d=0 e=0
# 15 CLK # 2 :: clk=1 a=1 b=0 c=0 d=0 e=0
# 25 CLK # 3 :: clk=1 a=1 b=0 c=0 d=1 e=0
# 35 CLK # 4 :: clk=1 a=0 b=0 c=0 d=0 e=0
# 45 CLK # 5 :: clk=1 a=0 b=0 c=0 d=0 e=1
# 45 expect pass
# 45 Goodbye after expect
# 55 CLK # 6 :: clk=1 a=0 b=0 c=0 d=0 e=1

```

Fig. 14.97 “expect”: basics

### 14.23 “expect”

“expect” takes on the same syntax as “assert property” in a procedural block. Note that “expect” must be used only in a procedural block. It cannot be used outside of a procedural block as in “assert” or property/sequence (recall that “assert property” can be used both in the procedural block and outside of it in a sequence or a property). So, what is the difference between “assert” and “expect”?

“expect” is a blocking statement, while “assert property” is a non-blocking statement. Blocking means, the procedural block will wait until “expect” sequence completes (pass or fail). For “assert property” non-blocking means that the procedural block will trigger the “assert property” statement and continue with the next statement in the block. “assert property” condition will continue to execute in parallel to the procedural code. Note that “assert property” behavior is the same whether it is outside or inside a procedural block. It always executes in parallel on its own thread with the rest of the logic (Fig. 14.97).

`$changed(expression [, clocking event]);`

Returns True if the expression changed from the previous tick of the clocking event. Otherwise it returns False.

#### Notes:

- The [, clocking event] is optional and usually derived from the clocking event of the assertion or from the inferred clock of the procedural block where the function is used
- When this function is called at or before the simulation time step in which the first clocking event occurs, the results are computed by comparing the sampled value of the expression with its default sampled value
- This function can be used in property/sequence as well as in procedural code as expression
- `$changed(expr)` is true if the sampled value of ‘expr’ in the pre-poled region of current time stamp changed from the sampled value in the pre-poled region of the previous time stamp.

Fig. 14.98 \$changed

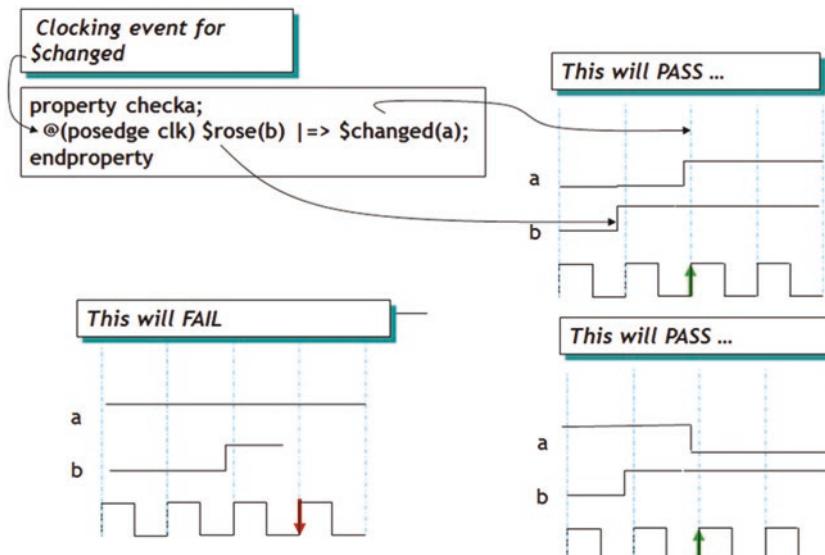


Fig. 14.99 \$changed - pass and fail cases

## 14.24 IEEE 1800-2012/2017 Features

### 14.24.1 \$changed

SystemVerilog 2009/2012 adds \$changed sampled value function in addition to the ones we have already seen such as \$past, \$rose, \$fell, and \$stable. Refer to Figs. 14.98 and 14.99.

Here is a simple example.

Specification: Make sure that “toggleSig” toggles every clock. In other words, see that “toggleSig” follows the pattern 101010...or 010101...

Solution: First inclination will be to write the assertion as follows:

```
tP: assert property (@(posedge clk) toggleSig ##1 !toggleSig);
```

But will this work? No. This property simply states that toggleSig be true every clock that it is false the next clock. What that also means is that the next clock, we are checking for toggleSig to be both true and false at the same time! The assertion (most likely) will fail at this next clock since at this clock toggleSig could be inverted. That is not what we are checking for.

Here is where \$changed comes to rescue. The following property will verify the toggle specification:

```
tP: assert property (@(posedge clk) ##1 $changed(toggleSig));
```

### 14.24.2 Future Global Clocking Sampled Value Functions

The *future* sampled value functions are:

```
$future_gclk(expression)
$rising_gclk(expression)
$falling_gclk(expression)
$steady_gclk(expression)
$changing_gclk(expression)
```

The future sampled value functions use the subsequent (future) value of the expression. Note that the global clocking *future* sampled value functions may be invoked only in *property\_expr* or in *sequence\_expr*. This implies that *they cannot be used in assertion action blocks*.

Please refer to Sect. 19.2 for the discussion on global clocking.

`$future_gclk(expression)` returns the sampled value of expression at the next global clocking tick.

`$rising_gclk(expression)` returns a Boolean true if the sampled value of the *least significant bit* of the expression changes to 1 at the next global clocking event. Else it returns false.

`$falling_gclk(expression)` returns a Boolean true if the sampled value of the *least significant bit* of the expression changes to 0 at the next global clocking event. Else it returns false.

`$steady_gclk(expression)` returns true (1'b1) if the sampled value of the expression does not change at the next global clock tick. Otherwise, it returns false (1'b0).

`$changing_gclk(expression)` is the complement of `$steady_gclk`, i.e., `!$steady_gclk(expression)`. Returns true (1'b1) if the sampled value of the expression changes at the next global clock tick.

Example:

```
a1: assert property (@$global_clock $changing_gclk(req) |->
$falling_gclk(req))
      else $error("req is not stable");
```

In the above example, a future (one global clock tick away) changing “req” implies that it is a falling “req” at that same global clock tick (since there is an overlapping operator).

Note that the following are illegal assertions using future sampled value functions:

```
//ILLEGAL - cannot use in procedural assignment
always @(posedge_clk) a <= $future_gclk(b) && c;

//ILLEGAL - can't use in disable iff
a2_illegal: assert property (@(posedge clk) disable iff (rst
|| $rising_gclk(interrupt)) req |=> gnt);

//ILLEGAL - can't have nested sampled value functions
a3_illegal: assert property (@(posedge clk) req |-> $future_
gclk (ack && $rising_gclk(gnt));
```

### 14.24.3 past Global Clocking Sampled Value Functions

The *past* sampled value functions are:

```
$past_gclk (expression)
$rose_gclk (expression)
$fell_gclk (expression)
$stable_gclk(expression)
$changed_gclk(expression)
```

The globally clocked *past* sampled value functions work the same way as the non-global clocking sampled value function. If you recall, these past sampled value functions take an explicit clocking event. So, \$rose\_gclk (expr) is equivalent to \$rose(expr, @ \$global\_clock). Please refer to the non-global clocking past sampled value functions in Sect. 14.17 to understand how these functions work. The functionality is the same, except that the past global clocking sampled value functions work off a global clock.

The global clocking *past* sampled value functions are a special case of the sampled value functions, and therefore the regular restrictions imposed on the sampled value functions and their arguments apply. In particular, the global clocking past sampled value functions are usable in general procedural code and action blocks.

#### 14.24.4 “followed by” Properties # -# and # =#

The *followed by* properties has the following form:

```
sequence_expression # -# property_expression
sequence_expression # =# property_expression
```

# -# is overlapped property and # =# is non-overlapped, just as in | -> and | =>. But there are differences between the *implication* operators and the *followed by* operators.

For *followed by* to succeed, *both* the antecedent sequence\_expression and the consequent property\_expression must be true. *If the antecedent sequence\_expression does not have any match, then the property fails*. If the sequence\_expression has a match, then the consequent property\_expression must match.

This is the fundamental difference between the implication operators (| -> and | =>) and the *followed by* operators. Recall that with implication operators, if the antecedent does not match, you get a vacuous pass and not a fail.

For overlapped followed by, there must be a match for the antecedent sequence\_expr, where the end point of this match is the start point of the evaluation of the consequent property\_expr. For non-overlapped

followed by, the start point of the evaluation of the consequent property\_expr is the clock tick after the end point of the sequence\_expr match.

Obviously, # -# being an overlapped operator, it starts the consequent evaluation the same time that the antecedent match ends (and succeeds). Consequently, the # =# non-overlapped operator will start the consequent evaluation of the clock after the antecedent match ends and succeeds.

Here is a simple example:

```
property p(a, b, c)
  @ (posedge clk) c | -> a #== b;
endproperty
assert property (p(req[*5], gnt, c ));
```

Request needs to remain asserted (high) for five consecutive clocks. One clock later gnt must be asserted (high). If request does *not* remain asserted for five consecutive clocks, the assertion will fail. If it does remain asserted for five clocks and the next clock gnt is not asserted, the assertion will fail. If both the antecedent and consequent match in the required temporal domain, the property will pass.

Let us look at the simulation log for above property “p”:

```
run 200
#
#      0  clk=0 c=0 req=0 gnt=0
#      5  clk=1 c=0 req=0 gnt=0
#     10  clk=0 c=1 req=1 gnt=0
#     15  clk=1 c=1 req=1 gnt=0
#     20  clk=0 c=0 req=1 gnt=0
#     25  clk=1 c=0 req=1 gnt=0
#     30  clk=0 c=0 req=1 gnt=0
#     35  clk=1 c=0 req=1 gnt=0
#     40  clk=0 c=0 req=1 gnt=0
#     45  clk=1 c=0 req=1 gnt=0
#     50  clk=0 c=0 req=1 gnt=0
#     55  clk=1 c=0 req=1 gnt=0
#     60  clk=0 c=0 req=0 gnt=1
#     65  clk=1 c=0 req=0 gnt=1
# At 65ns 'followed by' PASS
#      70  clk=0 c=0 req=0 gnt=0
#      75  clk=1 c=0 req=0 gnt=0
#     80  clk=0 c=1 req=1 gnt=0
#     85  clk=1 c=1 req=1 gnt=0
#     90  clk=0 c=0 req=1 gnt=0
#     95  clk=1 c=0 req=1 gnt=0
#    100  clk=0 c=0 req=1 gnt=0
#    105  clk=1 c=0 req=0 gnt=0
#    110  clk=0 c=0 req=0 gnt=0
#    115  clk=1 c=0 req=0 gnt=0
# At 115ns 'followed by' FAIL
#      120  clk=0 c=0 req=0 gnt=0
#      125  clk=1 c=0 req=0 gnt=0
#      130  clk=0 c=0 req=0 gnt=0
#      $finish      : followedby.sv(35)
```

Simulation log reads this way.

At time 10, “c” (the antecedent) is 1 and hence the consequent starts evaluation. After that, “req” is high for five clocks consecutively. Then one clock later, “gnt” is high and the property passes as expected.

At time 80, the property fires again since “c” is 1. But this time “req” remains high only for two clocks. So, the property fails.

#### **14.24.5 “always” and “s\_always” Property**

“always” property behaves exactly as you would expect. The syntax for “always” (and its variations) is:

1. **always property\_expression** (weak form)
2. **always [cycle\_delay constant\_range expression] property\_expression** (weak form with *unbounded* range)
3. **s\_always [constant\_range] property\_expression** (strong form with *bounded* range)

s\_always is the strong form with bounded range. As LRM puts it (for s\_always), property “s\_always [constant\_range] property\_expression” evaluates to true if and only if all current and future clock tick specified by constant\_range exist and property\_expr holds at each of these clock ticks:

```
property reset_always;
    @ (posedge clk) POR[*5] |=> always !reset;
endproperty
```

The property says that once POR (power on reset) signal has remained high for five consecutive clocks starting the next clock, reset would remain low “always” (forever).

“always” makes it simple to specify the continuous longevity of an assertion.

Next, let us see how always [cycle delay constant range] works:

```
property p1;
    @ (posedge clk) a |-> always [3:$] b;
endproperty
```

property p1 says that if “a” is true that “b” will be true three clocks *after* “a” and will remain true “always” (forever) after the three clocks. Note that “always [n:m]” allows an unbounded range.

In contrast “s\_always” allows only bounded range.

So, let us see what s\_always does:

```
property p2;
  @ (posedge clk) a |-> s_always [3:10] b;
endproperty
```

The property says that if “a” is true, then “b” remains true from the third clock to the tenth clock after “a” was detected true. This is a “strong” property. Recall strong property that we discussed earlier. This “s\_” property also works the same way. In other words, if you run out of simulation ticks for “s\_always,” the property may fail (at least with the simulators I have tried). Strong property requires that some terminating condition happen in the future, and this includes the requirement that the property clock ticks enough time to enable the condition to happen.

*But*, why do we need “always”? Do not the concurrent assertions always execute at every clock tick? The answer is yes which means we do not always need an “always” operator with a concurrent assertion. For example, in the following, “always” is redundant:

```
P1: assert property p1p (@ (posedge clk) always bstrap1==0);
```

There is no reason for an “always” in the above concurrent assertion. It is the same as:

```
P1: assert property p1p (@ (posedge clk) bstrap1==0);
```

One more:

```
property xx_chk (logic aStrobe, logic data);
  @ (posedge clk) disable iff(rst)
    $rose(aStrobe) [->1] |=> always (!$isunknown(data) &&
    $stable(data));
endproperty
```

If the aStrobe goes high at least once, the data cannot be unknown and must be stable. So once aStrobe goes high, the property will “always” check for data integrity at all future posedge clocks.

### **Exercise**

How would you write this property – without – the “always” operator?

Hint:

Use consecutive repetition operator.

#### 14.24.6 “eventually” and “s\_eventually”

There are two types of this operator, the “weak” kind (“eventually”) and the “strong” kind (“s\_eventually”). Here are three forms of these two properties:

**s\_eventually property\_expr** (*strong* property without range)  
**s\_eventually [cycle\_delay\_constant\_range] property\_expr** (*strong* property with range)

- The constant\_range can be unbounded.

**eventually [constant\_range] property\_expr** (*weak* property with range)

- The constant\_range must be bounded.

Some examples:

```
property p1;
    s_eventually $fell(frame_);
endproperty
```

Eventually PCI cycle will start with assertion of frame\_ (frame\_ goes low). If frame\_ does not assert until the end of simulation time, the property will fail (as per the simulator I have tried) since this is a *strong* property (there must exist at least n+1 ticks of the clock for the property to be true). Strong property requires that some terminating condition happen in the future, and this includes the requirement that the property clock ticks enough time to enable the condition to happen. Note that frame\_ can be true in current clock tick or any future clock tick:

```
property p2;
    s_eventually [2:5] $fell(frame_);
endproperty p2;
```

A new PCI cycle must start (frame\_ goes low) within the range of two clocks from now and eventually by the fifth clock (second and fifth clock inclusive). Note that as with any *strong* property, s\_eventually[n : m] property\_expr evaluates to true if, and only if, there exist at least n+1 ticks of the clock of the eventually property, including the current time step, and property\_expr evaluates to true beginning in one of the n+1 to m+1 clock ticks, where counting starts at the current time step.

**Exercise** Are the following properties “p3” and “p2” equivalent? Hint: Simulate from “initial” condition to know the subtle difference:

```

property p2;
    s_eventually [2:5] $fell(frame_);
endproperty p2;
property p3;
    frame_ |-> ##[2:5] $fell(frame_);
endproperty

```

Following is **s\_eventually** with unbounded range.

```

property p4;
    s_eventually [2:$] $fell(frame_);
endproperty

```

A new PCI cycle must start two clocks from now or any time after that:

```

property p5;
    eventually [2:$] $fell(frame_);
        // ILLEGAL. Weak property must be bound.
endproperty
property p6;
    s_eventually always a;
endproperty

```

“a” will eventually (starting current clock tick) go high and then remain high at every clock tick after that until the end of simulation.

*Simulation performance efficiency tips:*

- (1) *Checking property “s\_eventually always p” in simulation may be costly, especially if it is not in the scope of an initial procedure.*
- (2) *Consider the following assertions:*

```
a1: assert property ( req |-> ##[1:10000] gnt); //Inefficient
```

*This assertion should be written as:*

```
a2: assert property (req |-> s_eventually gnt); //efficient
```

*Although assertions a1 and a2 are equivalent, simulation performance of a1 may be worse. Avoid using large temporal domain delays.*

#### 14.24.7 “until,” “s\_until,” “until\_with,” and “s\_until\_with”

There are four forms of “until” property:

1. property\_expression1 **until**property\_expression2 (weak form – non-overlapping)
2. property\_expression1 **s\_until** property\_expression2 (strong form – non-overlapping)
3. property\_expression1 **until\_with** property\_expression2 (weak form – overlapping)
4. property\_expression1 **s\_until\_with** property\_expression2 (strong form – overlapping)

Let us start with “until”:

```
property p1;
    req until gnt;
endproperty
```

property p1 is true if “req” is true until “gnt” is true. In other words, “req” must remain true as long as “gnt” is false. “req” need *not* be true at the clock tick when “gnt” is found to be true. In other words, **until** is non-overlapping. An **until** property of the non-overlapping form evaluates to true if “req” evaluates to true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until at least one tick *before* a clock tick where “gnt” evaluates to true. If “gnt” is never true, “req” will remain true at every current and future clock tick. Based on the simulators that I have tried, since **until** is of weak form, if this property never completes (i.e., “gnt” is never true), the property will *not* fail. *Disclaimer: Different simulators have given different results when it comes to strong and weak properties:*

```
property p1;
    req s_until gnt;
endproperty
```

**s\_until** is identical to **until** except that if “gnt” never arrives and you run out of simulation time, the property may fail (based on the simulators I have tried).

To reiterate the difference between strong and weak properties, an “until” property of one of the strong forms requires that a current or future clock tick *exists* at which “gnt” evaluates to true, while an “until” property of one of the weak forms does not make this requirement. Strong properties require that some terminating condition happen in the future, and this includes the requirement that the property clock ticks enough time to enable the condition to happen. Weak properties do not impose any requirement on the terminating condition:

```
property p1;
  req until_with gnt;
endproperty
```

property p1 is true if “req” is true until and *including* a clock tick when “gnt” is true. In other words, “req” must remain true as long as “gnt” is false. “req” must be true at the *same* clock tick when “gnt” is found to be true. If “gnt” is never true, “req” will remain true at every current and future clock tick. In other words, **until\_with** is an overlapping property. Property **until\_with** requires “req” and “gnt” to be true at the *same* clock tick when “gnt” is found to be true. **until** does not have this requirement:

```
property p1;
  req s_until_with gnt;
endproperty
```

Same as **until\_with** but if you run out of simulation tick (e.g., end of simulation), and if “gnt” is never found to be true, this property may fail (simulator results on this may vary).

#### 14.24.8 “*nexttime*” and “*s\_nexttime*”

“*nexttime*” *property\_expression* evaluates to true, if *property\_expression* is true at time  $t+1$  clock tick.

There are four forms of “*nexttime*”:

**nexttime** *property\_expression* (weak form)

The weak *nexttime* property **nexttime** *property\_expr* evaluates to true if, and only if, either the

*property\_expr* evaluates to true beginning at the *next clock tick* or there is no further clock tick.

**s\_nexttime** *property\_expression* (strong form)

The strong *nexttime* property **s\_nexttime** *property\_expr* evaluates to true if, and only if, there

exists a next clock tick and *property\_expr* evaluates to true beginning at that clock tick.

**nexttime [constant\_expression]** *property\_expression* (weak form)

The indexed weak *nexttime* property **nexttime [constant\_expression]** *property\_expr* evaluates to

true if, and only if, either there are not *constant\_expression* clock ticks or *property\_expr* evaluates to

true beginning at the last of the next *constant\_expression* clock ticks.

The “constant\_expression” is useful as shown in the following example.

Let us say you want to write a property that looks for “req” to be true after three clocks:

```
nexetime nextime nextime req;
```

This is cumbersome and difficult to write. You can write the same property as follows:

```
nextime [3] req;
```

*Note:* **nextime** with a big “constant\_expression” is inefficient both in simulation and in formal verification. Try to keep the constant number small, especially in complex assertions. In simulation, it is recommended not to exceed several hundreds for the “constant\_expression” and in formal verification not to exceed a couple of tens. The common rule is the smaller, the better:

#### **s\_nextime [constant\_expression] property\_expression** (strong form)

The indexed strong nextime property **s\_nextime [constant\_expression] property\_expr** evaluates to true if, and only if, there exist *constant\_expression* clock ticks and *property\_expr* evaluates to true beginning at the last of the next *constant\_expression* clock ticks.

Let us examine the following simple example:

```
property p1;  
  @ (posedge clk) nextime req;  
endproperty
```

The above property says that the property will pass, if the clock ticks once more and “req” is true at the next clock tick ( $t+1$ ). In addition, since this is the weak form, if you run out of simulation ticks (i.e., there is no  $t+1$ ), this property will not fail.

Some examples.

What if you want to check to see that “req” remains asserted for all the clocks following the next clock? Following will do the trick:

```
property p1;  
  @ (posedge clk) nextime always req;  
endproperty
```

Or what if you want to see that starting next clock, “req” will eventually become true? The following will do the trick:

```
property p1;
  @ (posedge clk) nexttime eventually req;
endproperty
```

What if you want to see that “req” is true after a certain exact number (#) of clocks? The following will do the trick:

```
property p1;
  @ (posedge clk) nexttime[5] req;
endproperty
```

This property says that “req” shall be true at the fifth future clock tick (provided that there are indeed five clock ticks in the future, of course):

```
property p1;
  @ (posedge clk) s_nexetime req;
endproperty
```

Same as “nexetime” except that if you run out of simulation ticks after the property is triggered (i.e., there is no (t+1), the property will fail (based on simulators I have tried). Other way to look at this is that there exists a next clock and “req” should be true at that next clock, else the property will fail.

Similarly, the following property says that there must be at least five clock ticks and that “req” will be true at the fifth future clock tick:

```
property p1;
  @ (posedge clk) s_nexetime [5] req;
endproperty
```

The following example: when “seq1” ends(matches) at “t,” the next time tick (“t+1”) “seq\_expr” must be equal to ‘hff:

```
property
  @ (posedge clk) (seq1.matched nexttime seq_expr == 'hff);
endproperty
```

One more real-life issue we face that can be solved with **nexttime**. Initial “x” condition can always give us false failures. This can be avoided with the use of **next-time**. For example:

Let us say you are using \$past to do a simple “xor” of past value and present value (Gray encoding):

```
property
@ (posedge clk)
    $onehot (fifoctr ^ $past (fifoctr));
endproperty
```

At time “initial,” \$past (fifoctr) will return “x” (unknown) and the “xor” would fail right away. This is a false failure, and you may spend unnecessary time debugging it. Here is how **nexttime** can solve that problem:

```
property
@ (posedge clk)
    nexttime $onehot (fifoctr ^ $past (fifoctr));
endproperty
```

**nexttime** will avoid the initial \$past value of fifoctr and move the comparison to the next clock tick when (hopefully) you have cleared the fifoctr and the comparison will not fail due to the initial “x.”

Similarly, if you want to know that a signal stays stable forever (e.g., bootstrap signals), you may write a property as follows:

```
property
@ (posedge clk)
    $stable (bstrap);
endproperty
```

But this will sample the value “x” (e.g., for a “logic” type which has not been explicitly initialized) at time tick 0 and then continue to check to see that it stays at “x.” You end up checking for a stable “x.” Completely opposite of what you want to accomplish. Again, **nexttime** comes to rescue:

```
property
@ (posedge clk)
    nexttime $stable (bstrap);
endproperty
```

This will ensure that you start comparing the previous value of bstrap with the current value, starting *next* clock tick. Obvious, but easy to miss.

We discussed multiclock properties in Sect. 14.20. Here is an example of how **nexttime** can be used in a multiclock property:

N1: assert property

```
@ (posedge clk1) x | -> nexttime @ (posedge clk2) z;
```

**Important Note:** It is very important to understand how this property works. The “posedge clk1” flows through to “nextime” – in other words, “nextime” does not use “@(posedge clk2)” to advance time to the next tick. So, when “x” is true at (“posedge clk1”), the “nextime” causes advance to the next occurrence of “posedge clk1” strictly after when “x” was detected true before looking for a concurrent or subsequent occurrence of “posedge clk2” at which to evaluate “z.”

**Exercise** How would the following property behave in contrast with the one above?

```
N1: assert property
  @ (posedge clk1) x | -> @ (posedge clk2) nextime z;
```

## 14.25 Abort Properties: reject\_on, accept\_on, sync\_reject\_on, and sync\_accept\_on

Recall “disable\_IfExists” disable condition which preempts the entire assertion, if true. “disable\_IfExists” is an asynchronous abort (or reset) condition for the entire assertion. It is also asynchronous in that its expression is *not* sampled in the prepended region, but the expression is evaluated at every time stamp (i.e., in between clock ticks and at the clock ticks), and whenever the “disable\_IfExists” expression turns true, the entire assertion will be abandoned (no pass or fail).

With that background, 1800-2009/2012 adds four more abort conditions. “reject\_on” and “accept\_on” are asynchronous abort conditions (as in disable\_IfExists), and “sync\_reject\_on” and “sync\_accept\_on” are synchronous (i.e., sampled) abort condition. Note that “accept\_on” is an abort condition for *pass*, even though that may seem a bit counterintuitive at first. In other words, if “accept\_on” aborts an evaluation, the result is a *pass*. If “reject\_on” aborts an evaluation, the result is *fail*.

The syntax for all four is the same:

```
accept_on (abort condition expression) property_expression
sync_accept_on (abort condition expression) property_expression
reject_on (abort condition expression) property_expression
sync_reject_on (abort condition expression) property_expression
```

Before we see examples, here are high-level points to note:

1. One note off the bat to distinguish “disable\_IfExists” from the abort properties is that “disable\_IfExists” works at the “entire concurrent assertion” level, while these abort properties work at the “property” level. Only the property\_expression associated with the abort property will get “aborted” – not the entire assertion as with “disable\_IfExists.” More on this later.
2. The operators “accept\_on” and “reject\_on” work at the granularity of simulation time step (i.e., asynchronously).

3. In contrast, the operators “sync\_accept\_on” and “sync\_reject\_on” do *not* work at the granularity of simulation time step. They are sampled at the simulation time step of the clocking event (i.e., the sampling event).
4. You can nest the four abort operators “accept\_on,” “reject\_on,” “sync\_accept\_on,” “sync\_reject\_on.” Note that nested operators are in the lexical order “accept\_on,” “reject\_on,” “sync\_accept\_on,” and “sync\_reject\_on” (from left to right). While evaluating the inner abort property, the outer abort property takes precedence over the inner abort condition in case both conditions occur at the same time tick.
5. Abort condition cannot contain any reference to local variables or the sequence methods .triggered and .matched.
6. An abort is a property, so the result of an evaluation is either pass or fail. An aborted evaluation results in pass for the “accept” operators and fail for the “reject” operators.
7. There are no default abort conditions.

Now let us look at some examples to nail down the concepts:

```
property p1;
    @ (posedge clk) $fell(bMode) |-> reject_on(bMode)
data_transfer[*4];
endproperty
assert property (p1);
```

The above example specifies that on the falling edge of burst mode (bMode), data\_transfer should remain asserted for four consecutive clocks and that the bMode should – not – go high during those four data transfers. The way the property reads is to look for the falling edge of bMode, and starting that clock *reject* (fail) the property “(data\_transfer[\*4])” if at any time (i.e., asynchronously – even between clock ticks), it sees bMode going high. As noted before, “reject\_on” abort means failure. Hence consequent will *fail* and so will the property p1.

The important thing to note here is that the evaluation of the abort property, namely, data\_transfer[\*4] and the reject condition reject\_on(bMode) start at the *same* time. In other words, as shown below, this is like a “throughout” operator where the LHS is checked to see if it holds for the entire duration of RHS. Similarly, here we check to see that while we are monitoring “data\_transfer[\*4]” to hold, “bMode” should not go high. If it does go high at any time during “data\_transfer[\*4],” the property will be rejected, i.e., it will fail.

The same property can be written using sync\_reject\_on, only that the “bMode” will not be evaluated asynchronously (any time including in-between clock ticks) but will be sampled only at the sampling edge, clock tick.

Note that the above property can be written using “throughout” as well. Please refer to Sect. 14.18.9 on “throughout” operator to see a similar example:

```

property p1;
    @ (posedge clk) $fell(bMode) |-> ! (bMode) throughout
data_transfer[*4];
endproperty
assert property (p1);

```

Let us look at an example of “accept\_on”:

```

property reqack;
    @ (posedge clk) accept_on(cycle_end) req |-> ##5 ack;
endproperty
assert property (reqack);

```

This property uses “accept\_on(cycle\_end)” as the abort condition on the property “req |-> ##5 ack.” When “req” is sampled high on a posedge clk, the property “req |-> ##5 ack” starts evaluating waiting for ack to arrive after five clocks. At the same time, “cycle\_end” is also monitored to see if it goes high. Here are the scenarios that take place.

“cycle\_end” arrives within the five clocks that the property is waiting for “ack.” The accept\_on condition will be true in that case, and the property will be considered to pass. The next evaluation will again start the next time “req” is sampled high on posedge clk.

“cycle\_end” does not arrive within five clocks when the property is waiting for “ack.” The property will evaluate as with any concurrent assertion, and if “ack” does not come in high at the fifth clock, the property will fail. If “ack” does come in high at the fifth clock, the property will pass.

“cycle\_end” arrives exactly the same time as “ack” at the five clocks. The abort condition takes precedence. Since in this case, both “ack” arrived and the accept\_on were triggered at the same time, the accept\_on aborts the evaluation with a pass and so the assertion will pass. What if we used “reject\_on” instead of “accept\_on” in such a scenario?

In short, the property evaluation aborts on “accept\_on” (and passes) or “reject\_on” (and fails), OR it will finish on its own (and pass/fail) if the abort condition does not arrive.

Here are some more examples courtesy of 1800-2009/2012 LRM:

```

property p; (accept_on(a) p1) or (reject_on(b) p2);
endproperty

```

Recall that “or” requires either the LHS or the RHS to complete and pass. In the same scenario as above, if “a” becomes true first during the evaluation of p1, p1 is aborted and will pass (i.e., accepted) and the property “p” will pass. Similarly, if “b” becomes true first, then p2 is aborted and property p will fail (i.e., rejected).

Note that nested operators are in the lexical order “accept\_on,” “reject\_on,” “sync\_accept\_on,” and “sync\_reject\_on” (from left to right). If two nested operator

conditions become true in the same time tick during the evaluation of the property, then the outermost operator takes precedence:

```
property p; accept_on(a) reject_on(b) p1; endproperty
```

Note there is no operator between accept\_on and reject\_on.

If “a” goes high first, the property is aborted on accept\_on and will pass. If “b” goes high first, the reject\_on succeeds and the property p will fail. If both “a” and “b” go high at the same time during the evaluation of p1, then accept\_on takes precedence and “p” will pass.

Another simple example:

```
Frame_accept_reject: assert property (
    @ (posedge clk)
        accept_on (Frame_)
            Cycle_start |=> reject_on(Tabort)
        )
    else $display ("Frame_ FAIL");
```

This is another example of nested asynchronous aborts. The outer abort (**accept\_on** (Frame\_)) has the scope of the entire property of the concurrent assertion. The inner abort (**reject\_on**(Tabort)) has the scope of the consequent of **|=>**. The inner abort does not start evaluation until Cycle\_start is true. Note that the outer abort (“**accept\_on**”) takes precedence over the inner abort (“**reject\_on**”).

**Exercise** Try the same property with **s\_accept\_on** and **s Reject\_on** and note the differences. Try both examples with different triggers of Frame\_ and Tabort, and see when/how the property passes and fails.

## 14.26 \$assertpassoff, \$assertpasson, \$assertfailoff, \$assertfailon, \$assertnonvacuouson, and \$assertvacuousoff

These system tasks add further control over assertion execution during simulation. We have seen \$asserton, \$assertoff, and \$assertkill before. Here is a brief explanation of what these new system tasks do.

**\$assertpassoff**: This system task turns off the action block associated with *pass* of an assertion. This includes *pass* indication because of both the vacuous and non-vacuous successes. To re-enable the *pass* action block, use **\$assertpasson**. It will turn on the *pass* action of both the vacuous and non-vacuous successes. If you want to turn on only the non-vacuous *pass*, then use **\$assertnonvacuouson** system task. Note that these system tasks do not affect an assertion that is already executing.

**\$assertfailoff:** This system task turns off the action block associated with the *fail* of an assertion. In order to turn it on, use **\$assertfailon**. Here also, these system tasks do not affect an assertion that is already executing.

**\$assertvacuousoff** system task turns off the *pass* indication based on a vacuous success. An assertion that is already executing is not affected. By default, we get a *pass* indication on vacuous pass.

All the system tasks take arguments, as we have seen before with \$asserton, \$assertoff, and \$assertkill. The first argument indicates how many levels of hierarchy below each specified module instance to apply the system tasks. The subsequent arguments specify which scopes of the model to act upon (entire modules or instances).

## 14.27 Embedding Concurrent Assertions in Procedural Block

Yep, you can indeed assert a concurrent property from a procedural block. Note that property and sequence itself are declared outside the procedural block.

Off the bat, what is the difference then between embedding an immediate assertion and embedding a concurrent assertion in the procedural block? Well, immediate assertion is invoked with “assert,” while embedded concurrent assertion is invoked with “assert property.” A concurrent assertion that is embedded in a procedural block is the regular concurrent assertion “assert property” (i.e., it can be temporal). In other words, an immediate assertion embedded in the procedural code will complete in zero time, while the concurrent assertion may or may not finish in zero time. But is the concurrent assertion in the procedural code blocking or non-blocking? Hang on to this thought for a while.

Figure 14.100 points out that the *condition* under which you want to fire an assertion is already modeled behaviorally and do not need to be duplicated in an assertion (as an antecedent). The example shows both ways of asserting a property. `ifdef P shows the regular way of asserting the property that we have seen throughout this book. `else shows the same property being asserted from an always block. But note that in the procedural block the assertion is preceded by “if (bState == CycleStart)” condition. In other words, a condition that could be already in the behavioral code is used to condition an assertion. If the property was “asserted” outside of the procedural block, you would have to duplicate the condition that is in the procedural block, as an antecedent in the property.

Let us turn our attention back to embedded concurrent assertion being blocking or non-blocking. What happens when you fire a concurrent assertion from the procedural code, and it does not finish in zero time? What happens to the procedural code that follows the concurrent assertion? Will it stall until the concurrent assertion finishes (blocking)? Or will the following code continue to execute in parallel to the fired concurrent assertion (non-blocking)? That is what Fig. 14.101 explains.

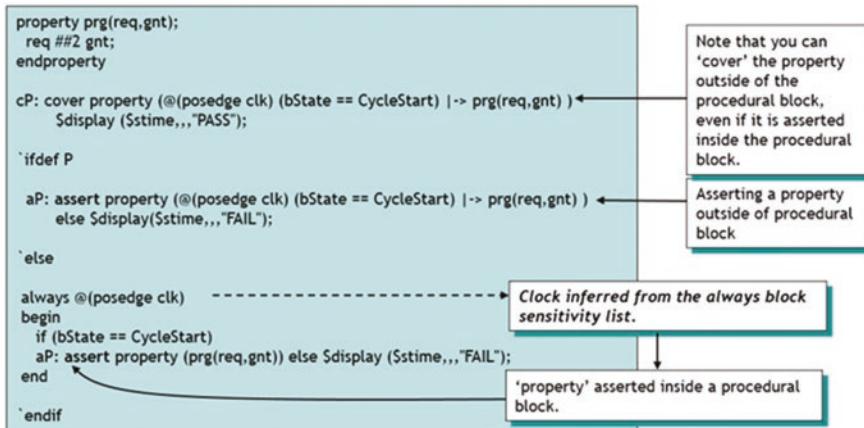


Fig. 14.100 Embedding concurrent assertion in procedural block

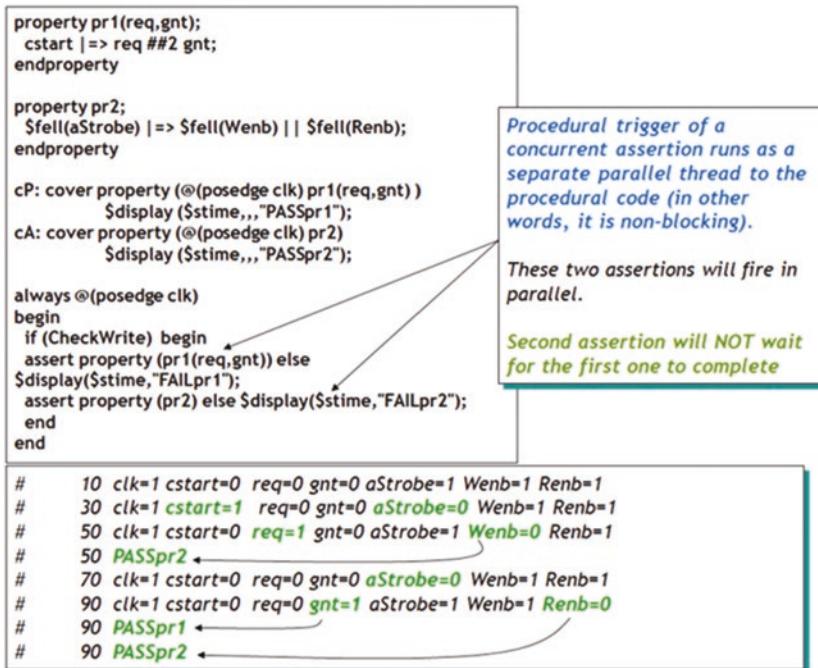


Fig. 14.101 Concurrent assertion embedded in procedural block is non-blocking

There are two properties “pr1” and “pr2.” They both “consume” time, i.e., they advance time. The procedural block “`always @ (posedge clk)`” asserts both these properties one after another without any time lapse between the two. How will

```

sequence bSeq;
##[1:5] b;
endsequence

sequence dSeq;
##2 d ##2 e;
endsequence

property mclocks;
@(posedge clk) a |> bSeq ##1 c |> dSeq;
endproperty

#    165 CLK # 17 :: clk=1 a=0 b=0 c=0 d=0 e=1
#    175 CLK # 18 :: clk=1 a=1 b=0 c=1 d=0 e=0 //a=1 so start
#    185 CLK # 19 :: clk=1 a=0 b=1 c=0 d=0 e=0
//b=1 next clock; so 'bSeq' matches

#    195 CLK # 20 :: clk=1 a=0 b=0 c=0 d=0 e=0
//But c NE 1 the next clock
//and the property does NOT fail.

#    205 CLK # 21 :: clk=1 a=0 b=0 c=0 d=0 e=0
#    215 CLK # 22 :: clk=1 a=0 b=0 c=1 d=0 e=0

//So, when 'c' does go '1' a couple of clocks later, the 'dSeq' seq
//won't start eval at that time. The fact that 'c' did not go '1'
//the very next clock after 'bSeq' matches that the entire
//property will simply not get evaluated for pass or fail; until // 'a' is asserted
again.

#    225 CLK # 23 :: clk=1 a=0 b=0 c=0 d=0 e=0
#    235 CLK # 24 :: clk=1 a=0 b=0 c=0 d=1 e=0
#    245 CLK # 25 :: clk=1 a=0 b=0 c=0 d=0 e=0
#    255 CLK # 26 :: clk=1 a=0 b=0 c=0 d=0 e=1
#    265 CLK # 27 :: clk=1 a=0 b=0 c=0 d=0 e=1

```

**Fig. 14.102** Nested implications in a property

this code execute? The procedural code will encounter ““assert property (pr1 ...)”” and fire it. “pr1” will start its evaluation by looking for cstart to be high and follow on through with the consequent. In other words, “pr1” is waiting for something to happen in time domain. *But the procedural code that fired it will not wait for “pr1” to complete.* It will move on to the very next statement which is “assert property (pr2)” and fire it as well. So, now you have “pr1” that is already under execution and “pr2” that is just fired both executing in parallel, and the procedural code moves on to the other code that sequentially follows.

*In short, a concurrent assertion in procedural code is non-blocking.*

As shown in the simulation log, at time 10, (@ posedge clk) we fire “pr1.” At the same time (since the very next statement is “assert property (pr2 ...)”), we fire “pr2.” At time 30, “cstart==1” and “aStrob==0.” This means the antecedent condition of both “pr1” and “pr2” has been met. At 50, “Wenb==0” completes the property

“pr2,” and the property passes as shown at time 50 in simulation log. Therefore, the first thing you notice here is that even though “pr1” was fired first, “pr2” finished first. In other words, since both properties were non-blocking and executing on their own parallel threads, there is no temporal relationship between them or among “pr1” and “pr2” and the procedural code. Following the same line of thought, see why both “pr1” and “pr2” pass at the same time at 90.

## 14.28 Nested Implications

Can you have multiple nested implications in a property? Sure, you can. However, you need to very carefully understand the consequences of nested implications in a property. Let us look an example and understand how this works.

In Fig. 14.102, the property `mclocks` (at first glance) looks very benign. But play close attention and you will see two implications. `@ (posedge clk)` if “`a`” is true implies “`bSeq' ##1 c`,” which implies “`dSeq`.`”` One antecedent implies a consequent which acts as the antecedent for another consequent.

Now, let us look at the simulation log. At time 175, `a=1`, so the property starts evaluation and implies “`bSeq ##1 c`.`”` At time 185 “`bSeq`” matches, so the property now looks for `##1 c`. At time 195, `c` is – not – equal to “1,” but the property does not fail. Wow! Reason? Note that “`bSeq ##1 c`” is now an antecedent for “`dSeq`” and as we know if there is no match on antecedent, the consequent will not be evaluated, and the property will not fail. Here that seems to apply even though “`bSeq ##1 c`” is a consequent, it is also an antecedent. Language anomaly? Not really, but the behavior of such properties is not quite intuitive. Since “`bSeq ##1 c`” did not match, the entire property is discarded, and the property again waits for the next “`a==1`” to start all over again.

Confusing? Well, it is. Hence, please do not use such nested implication properties unless you are absolutely sure that that is what you want. I have seen engineers use it because the logic seems intuitive but may not understand the behavior.

# Chapter 15

## SystemVerilog Functional Coverage



**Introduction** This chapter explores SystemVerilog functional coverage in detail. It discusses methodology components, covergroups, coverpoint, and various types of “bins” including binsof, intersect, cross, transition, wildcard, ignore\_bins, illegal\_bins, etc. The chapter also discusses sample/strobe methods and ways to query coverage.

Ah, so you have done everything to check the design. But what have you done to check your testbench? How do you know that your testbench has indeed covered everything that needs to be covered? That is where functional coverage comes into picture.

The origin of functional coverage can be traced back to the 1990s with the emergence of constrained random simulation. Obviously, one of the value propositions of constrained random stimulus generation is that the simulation environment can automatically generate thousands of tests that would have normally required a significant amount of manual effort to create directed tests. However, one of the problems with constrained random stimulus generation is that you never know exactly what functionality has been tested without the tedious effort of examining waveforms after a simulation run. Hence, functional coverage was invented as a measurement to help determine exactly what functionality a simulation regression tested without the need for visual inspection of waveforms.

So, functional coverage and constrained random verification go hand in hand. You need to evaluate the functional coverage gaps and, based on those gaps, constrain the input stimuli to cover those gaps.

It would be better if you could automatically determine, after the application of tests, what features of the design were covered and what should be the next

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_15](https://doi.org/10.1007/978-3-030-71319-5_15)) contains supplementary material, which is available to authorized users.

constrained random stimuli. There should be a loop where you apply stimulus, measure coverage, constrain your stimulus, run simulation again, and measure coverage and this process loops. The good news is that the latest product offerings from EDA vendors provide just such a methodology. That makes verification that much more effective.

But first let us understand the difference between good old code coverage and the latest in functional coverage.

## 15.1 Difference Between Code Coverage and Functional Coverage

### 15.1.1 Code Coverage

Code coverage is derived directly from the design code. It is not user specified. One of the advantages of code coverage is that it automatically evaluates the degree to which the source code of a design has been activated during simulation/regression, thus identifying structures in the source code that have not been activated during testing. One of the key benefits of code coverage, unlike functional coverage, is that creating the structural coverage model is an automatic process. Hence, integrating code coverage into your existing simulation flow is easy and does not require a change to either your current design or verification approach:

- Evaluates to see if design structure has been covered (i.e., line, toggle, assign, branch, expression, states, state transition, etc.)
- But does not evaluate the *intent* of the design:
  - If the user-specified busGnt = busReq && (idle || !(reset))
  - Instead of the real *intent* busGnt = busReq && (idle && !(reset))  
Code coverage will not catch it. For intent, you need both a robust testbench to weed out functional bugs and a way to objectively predict how robust the testbench is.

Here is a brief snapshot of what code coverage targets (structurally). You may refer to code coverage manuals from EDA vendors for further analysis. Here is a brief description.

#### 15.1.1.1 Line Coverage

Line coverage is a code coverage metric used to identify which lines of the source code have been executed during simulation. A line coverage metric report will have a count associated with each line of source code indicating the total number of times the line has executed. The line execution count value is not only useful for identifying lines of source code that have never executed but also useful when the engineer feels that a minimum line execution threshold is required to achieve sufficient testing.

Line coverage analysis will often reveal that a rare condition required to activate a line of code has not occurred due to missing input stimulus. Alternatively, line coverage analysis might reveal that the data and control flow of the source code prevented it either due to a bug in the code or dead code that is not currently needed under certain IP configurations. For unused or dead code, you might choose to exclude or filter this code during the coverage recording and reporting steps, which allows you to focus only on the relevant code.

### 15.1.1.2 Statement Coverage

Statement coverage is a code coverage metric used to identify which statements within the source code have been executed during simulation. In general, most engineers find that statement coverage analysis is more useful than line coverage since a statement often spans multiple lines of source code or multiple statements can occur on a single line of source code.

A metrics report used for statement coverage analysis will have a count associated with each line of source code indicating the total number of times the statement has executed. This statement execution count value is not only useful for identifying lines of source code that have never executed but also useful when the engineer feels that a minimum statement execution threshold is required to achieve sufficient testing.

One hundred percent statement coverage implies 100% line coverage. But the opposite is not true: a line may contain multiple statements, whereas a test may not succeed in executing all of them.

### 15.1.1.3 Branch Coverage

Branch coverage (also referred to as decision coverage) is a code coverage metric that reports whether Boolean expressions tested in control structures (such as the *if*, *case*, *while*, *repeat*, *forever*, *for*, and *loop* statements) evaluated to both true and false. The entire Boolean expression is considered one true-or-false predicate regardless of whether it contains logical-and or logical-or operators.

### 15.1.1.4 Finite-State Machine Coverage

Today's code coverage tools can identify finite-state machines within the RTL source code. Hence, this makes it possible to automatically extract FSM code coverage metrics to measure conditions, for example, the number of times each state of the state machine was entered and the number of times the FSM transitioned from one state to each of its neighboring states and even sequential arc coverage to identify state visitation transitions.

But note that the code coverage tool (obviously) does *not* know which transitions are valid. It will report all state transitions that have not been covered. You have to weed out which of these uncovered state transitions are don't care.

### 15.1.2 Functional Coverage

The idea behind functional coverage is to see that we have covered the *intent* of the design. Have we covered everything that the design specification requires? For example, you may have 100% code coverage, but your functional coverage could only be 50%. Covering just the structure of the RTL code (code coverage) does not guarantee that we have functionally covered what RTL actually intended to design. That being the case, it is obvious that the functional coverage matrices cannot be automatically created. One has to meticulously study the design specs and manually create functional coverage matrices using “`covergroup`,” “`coverpoint`,” “`bins`,” etc.

So, functional coverage is:

- User specified; identify features of design specs that require functional coverage.  
A manual process.
- Based on design specification (as we have already seen with “`cover`” of an assertion).
- Measures coverage of design *intent*.
- Control-oriented coverage:
  - Have I exercised all possible protocols that read cycle supports (burst, non-burst, etc.)?
  - *Transition* coverage:
    - Did we issue transactions that access byte followed by Qword and multiple Qwords? (use SystemVerilog *transition* coverage).
    - A write to L2 is followed by a read from the same address (and vice versa). Again, the *transition* coverage will help you determine if you have exercised this condition.
  - *Cross* coverage:
    - Tag and data errors must be injected at the same time (use SystemVerilog *cross* coverage).
- Data-oriented coverage:
  - Have we accessed cache lines at all granularity levels (odd bytes, even bytes, word, quad-word, full cache line, etc.)?

## 15.2 Functional Coverage Methodology

There are three components to a functional coverage methodology. Two are based on language features, while the third is a methodology components.

### **cover**

The first component of functional coverage methodology is “cover” (part of SystemVerilog Assertions language). As we saw under SystemVerilog Assertions chapter, “cover” uses SVA temporal syntax. Please refer to that chapter for a detailed overview of “cover.”

“cover” is basically a shadow of “assert.” In other words, you get double mileage, in that the same property can be used for both assertions and collecting functional coverage in combinatorial and temporal domain.

“cover” provides structural-level sequential coverage. It can only be placed in modules, programs, and interfaces and cannot be placed in a “class.”

### **Covergroup, Coverpoint, Bins**

The second component of the functional coverage methodology is based on covergroup, coverpoint, bins etc. of the functional coverage language.

Covergroups provides coverage of design variables. They record the number of occurrences of various values specified as coverpoints (of design variables).

Coverpoints query whether certain values or scenarios of the design variables have occurred. “cross” of coverpoints is a very important feature of the language that is essential to cover simultaneously occurring events (write and read cycles occur in parallel).

### **Testbench and Test Plan**

The third component (higher-level methodology component) is your testbench and test plan.

Your test plan is (obviously) based on what functions you want to test (i.e., cover). Create a functional cover matrix based on your test plan that includes each of the functions (control and data) that you want to test. Here’s a high-level description of the methodology:

- Identify in the functional coverage matrix all your functional covergroups/coverpoints/bins.
- Measure their coverage during verification/simulation process.
- Measure effectiveness of your tests from the coverage reports:
  - For example:
    - Your tests are accessing mostly 32-byte granules in your cache line; you will see byte, word, and quad-word coverage low or not covered. Change or add new tests to hit bytes/words, etc. Use constrained random methodology to narrow down the target of your tests.

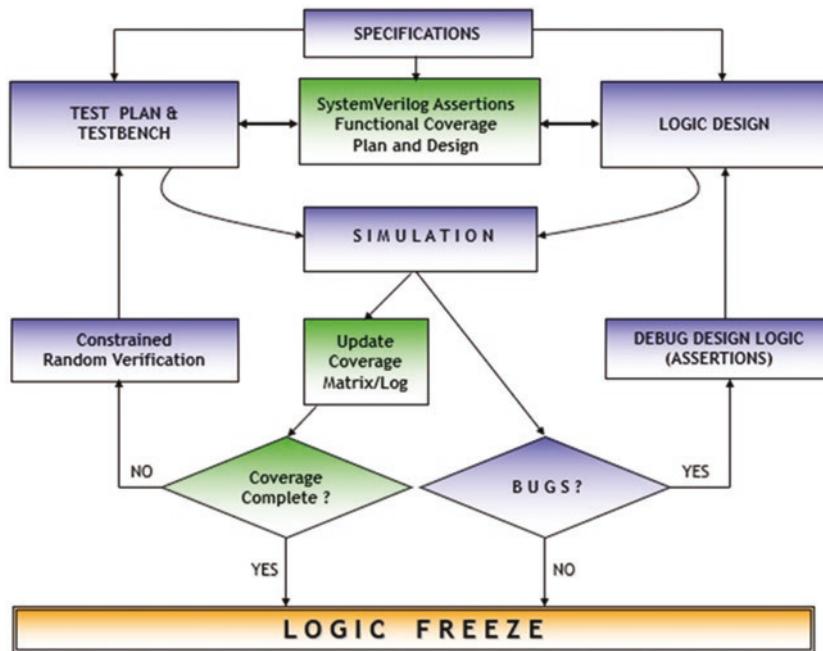
- The tests do not fire transactions that access byte followed by Qword and multiple Qwords. Check this with *transition* coverage.
- Or that tag and data errors must be injected at the same time (*cross* coverage between tag and data errors). Enhance your testbench to cover such “cross.”
- “cover” temporal domain assertions.
- Add more *coverpoints* for critical functional paths through design:
  - For example, a write to L2 is followed by a read from the same address and that this happens from both processors in all possible write/read combinations.
- Remember to update your functional cover plan as verification progresses:
  - Just because you created a plan at the beginning of the project does not mean it is an end all.
  - As your knowledge of the design and its corner cases increase, so should the exhaustiveness of your test plan and the functional cover plan.
  - Continue to add *coverpoints* for any function that you did not think of at the onset.

Refer to Fig. 15.1 for the overview of the methodology. You start with the design specification and create an assertions and functional coverage plan, in addition to starting to develop a testbench. The assertions feed into the testbench *and* the design, while coverage plan feeds into the testbench. Then you simulate the design and check to see if it has bugs. If it has bugs, then you debug the design and also assess the assertions plan and enhance it. In parallel, you check to see if coverage is adequate. If not, you start on constrained random verification methodology to target logic where coverage is insufficient. Once there are no bugs and coverage is complete, you freeze the logic for eventual tapeout.

### 15.3 Covergroup: Basics

Figure 15.2 explains the basics with its annotations. Key syntax of the covergroup and coverpoint is pointed out. A few points to reiterate are as follows:

1. *covergroup* without a *coverpoint* is useless, *and* the compiler will not give a warning (at least the simulators that the author have tried).
2. *covergroup*, as the name suggests, is a group of *coverpoints*, meaning you can have multiple *coverpoints* in a *covergroup*.
3. You must instantiate the *covergroup* with “new” just as you would with a class.
4. You may provide (not mandatory) a sampling edge (e.g., a clock edge) to determine when the *coverpoints* in a *covergroup* get sampled. If the clocking event is omitted, you must procedurally trigger the coverage sample window using a built-in method called *sample* ( ). We will discuss *sample* ( ) later in the chapter.



**Fig. 15.1** Comprehensive assertions- and functional coverage-based methodology

5. The sampling of covergroup takes place at the *instant* the sampling edge occurs. It does *not* take place in the preponed region as with “assert,” “cover,” “assume,” “restrict” semantic of SystemVerilog Assertions.
6. A “covergroup” can specify an optional list of formal arguments (discussed later in the chapter). The actual values are provided when you instantiate the covergroup (i.e., with the “new” operator). Actual arguments are evaluated when the “new” operator is executed. Note that the formal arguments are only be “input” to the covergroup. An output of inout will result in an error.
7. A *covergroup* can be declared in:
  - (a) Package
  - (b) Interface
  - (c) Module
  - (d) Program
  - (e) Class

Other points are annotated in Fig. 15.2. Carefully study them so that the rest of the chapter is easier to digest.

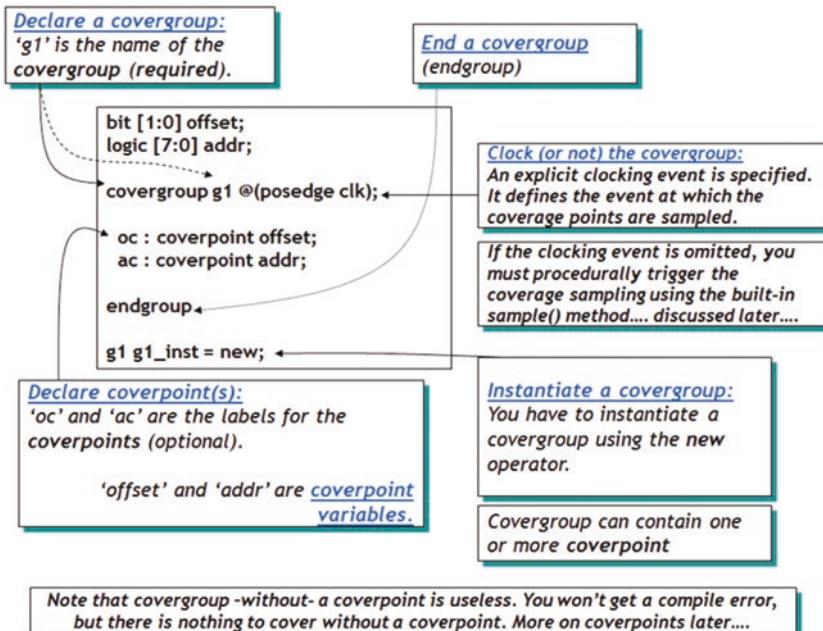


Fig. 15.2 Covergroup and coverpoint basics

## 15.4 Coverpoint: Basics

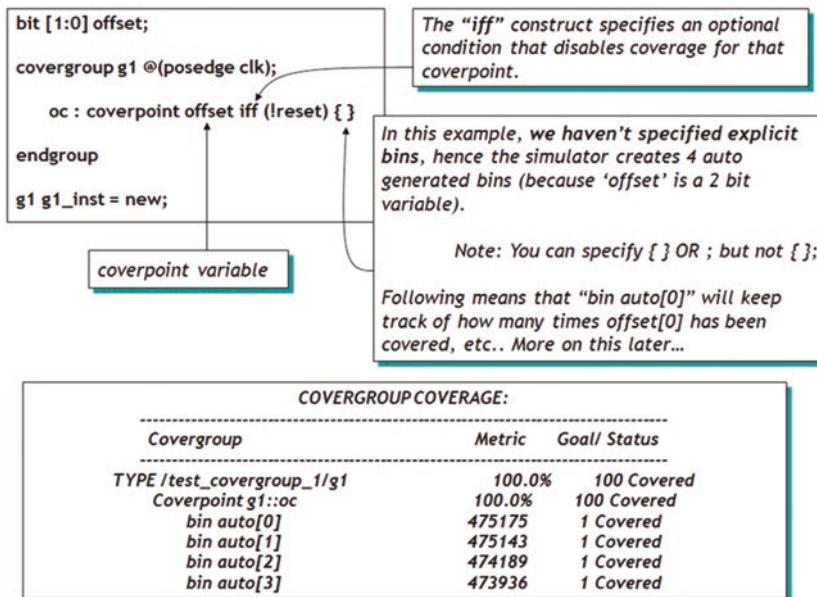
**Syntax:** Here is the formal syntax:

```

cover_point ::=
  [ [ data_type_or_implicit ] cover_point_identifier : ] cover-
  point_expression [ iff ( expression ) ] bins_or_empty
  
```

Coverpoint and bins associated with the coverpoint do all the work. The syntax for coverpoint is as shown in Fig. 15.3. “covergroup g1” is sampled at (posedge clk). “oc” is the coverpoint name (or label). This is the name by which simulation log refers to this coverpoint. “oc” covers the 2-bit variable “offset.”

We have not yet covered “bins,” so please hang on with the following description for a while. We will cover plenty of “bins” in the upcoming sections. In this example, you do not see any “bins” associated with the coverpoint “oc” for variable “offset.” Since there are no bins to hold coverage results, simulator will create those for you. In this example, the simulator will create four bins because “offset” is a 2-bit variable. If “offset” were a 3-bit vector, there would be eight bins and so on. We will discuss a lot more on “bins” in upcoming sections, and hence I am showing only the so-called auto-bins created by the simulator.



**Fig. 15.3** Coverpoint basics

I have not shown the entire testbench, but the simulation log (at the bottom of Fig. 15.3) shows that there are four auto-generated bins called “bin auto[0]” ... “bin auto[3].” Each of these bins covers 1 value of “offset.” For example, auto[0] bin covers “offset == 0.” In other words, if “offset==0” has been simulated, then auto[0] will be considered covered. Since all four bins of coverpoint “oc” have been covered, the coverpoint “oc” is considered 100% covered, as shown in the simulation log.

A data type for the coverpoint may also be specified explicitly or implicitly in *data\_type\_or\_implicit* field. In either case, it is understood that a data type is specified for the coverpoint. The data type must be an integral type. If a data type is specified, then a *cover\_point\_identifier* must also be specified.

If a data type is specified, then the coverpoint expression must be assignment compatible with the data type. Values for the coverpoint will be of the specified data type and will be determined as though the coverpoint expression was assigned to a variable of the specified data type.

If no data type is specified, then the inferred data type for the coverpoint will be the self-determined type of the coverpoint expression.

The expression within the *iff* construct specifies an optional condition that disables coverage for that coverpoint.

If the guard expression evaluates to false at a sampling point, the coverage point is ignored:

```
covergroup AR;
    coverpoint s0 iff(!reset);
endgroup
```

In the preceding example, coverage point s0 is covered only if the value of “reset” is low.

### 15.4.1 Covergroup/Coverpoint Example

PCI protocol consists of many different types of bus cycles. We want to make sure that we have covered each type of cycle. The enum type pciCommands (Fig. 15.4) describes the cycle types. The covergroup specifies the *correct* sampling edge (that being the “negedge FRAME\_”) for the PCI commands. In other words, the sampling edge is quite important for performance reasons. If you sampled the same covergroup @ (posedge clk), there will be a lot of overhead because FRAME\_ will fall only when a PCI cycle is to start. Sampling unnecessarily will indeed affect simulation performance and clutter your coverage log with redundant samples.

In this example, we have not specified any bins. So, the simulator creates 12 auto-bins for the 12 bus cycle types in the enum. Every time FRAME\_ falls, the simulator will see if any of the cycle types in “enum pciCommands” is simulated. Each of the 12 auto-bins corresponds to each of the enum types. When a cycle type is exercised, the auto-bin corresponding to that cycle (i.e., that “enum”) will be

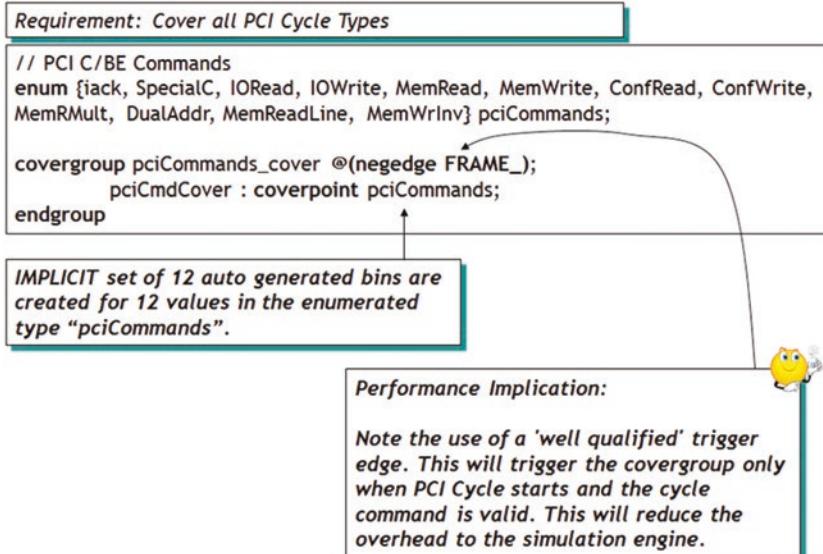


Fig. 15.4 Covergroup/coverpoint example

considered covered. Now, you may ask why wouldn't code coverage cover this? Code coverage will indeed do the job. But I am building a small story around this example. We will see how this covergroup will be then reused for transition coverage which *cannot* be covered by code coverage.

Before we move on to “bins,” here are a few different ways coverpoints can be created.

### 15.4.2 Coverpoint Using a Function or an Expression

```
typedef enum {QUIET, BUSY} stateM;

function stateM cPoint_State_function(bit valid, ready);
    if (valid == 0) return QUIET;
    if ((valid & ~ready) == 0) return BUSY;
endfunction
...
covergroup cg (valid, ready) @(posedge clk);
    cpStateM: coverpoint cPoint_State_function (valid, ready);
endgroup
cg cgInst = new(1'b0,1'b1);
```

In this example, we define a function “cPoint\_State\_function” of type stateM (which is a typedef enum). This function inputs “valid” and “ready.” When “valid” == 0, we want to cover the state QUIET, and when ((valid & ~ready) == 0), we want to cover the state BUSY.

In other words, this is an example of the number of “bins” that are created *based on some condition*. There are two “bins” auto-generated by the simulator (since you did not specify explicit “bins” for the function and the function returns two explicit values (*quiet* and *busy*) of the typedef enum “stateM”). And these two bins of the function will be covered based on the conditions provided in the function.

*One more example:*

```
cExample: coverpoint (my_variable + your_variable) % 4 ;
```

covers the bits created by a mod 4 of (my\_variable + your\_variable).

You can also use part-select for a coverpoint.

```
cExample: coverpoint(address[31:16]);
```

*You can cover a Ref variable as well:*

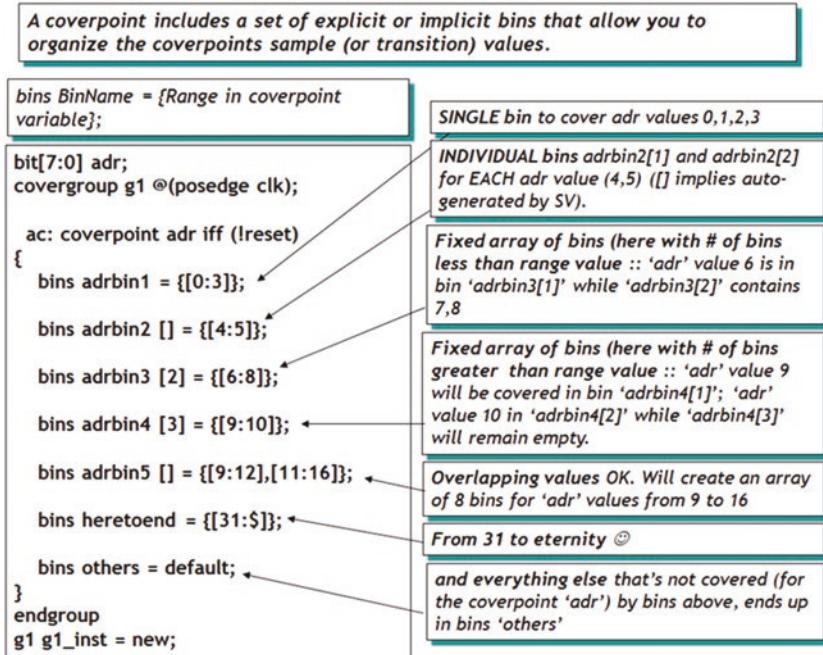


Fig. 15.5 “bins”: basics

```

covergroup (ref int ref_addr) cGroup;
  cPoint: coverpoint ref_addr;
endgroup

```

We'll see more on coverpoints as we progress through the chapter.

## 15.5 “bins”: Basics

What is a “bin”? A “bin” is something that collects coverage information (collect in a “bin”). Bins are created for coverpoints. A coverpoint is covering a variable (let us say the 8-bit “adr” as shown in Fig. 15.5) and would like to have different values of that variable be collected (covered) in different collecting entities; the “bins” will be those entities. “bins” allows you to organize the coverpoint sample values.

You can declare bins many different ways for a coverpoint. Recall that bins collect coverage. From that point of view, you have to carefully choose the declaration of your bins.

Ok, here is the most important point *very* easy to misunderstand. In the following statement, how many bins will be created? 16 or 4 or 1 and what will it cover?

```
bins adrbin1 = {[0:3]};
```

Answer: 1 bin will be created to cover “adr” values equal to 0 or 1 or 2 or 3.

Note that “bins adrbin1” is without the [ ] brackets. In other words, “bins adrbin1” will *not* auto-create four bins for “adr” values {[0:3]}; it will rather create *only one bin* to cover “adr” values 0, 1, 2, and 3.

*Very important point:* Do not confuse {[0:3]} to mean that you are asking the bin to collect coverage for adr0 to adr15. {[0:3]}. It literally means cover “adr” value =0, =1, =2, =3. Not intuitive, but that is what it is!

Another important point. “bins adrbin1 = {[0:3]},” also says that if we hit *either* of the “adr” value (0, 1, 2, or 3), that the single bin will be considered *completely* covered. Again, you do not have to cover all four values to have “bins adrbin1” considered covered. You hit any one of those four values, and the “adrbin1” will be considered 100% covered.

But what if you want each value of the variable “adr” be collected in separate bins so that you can indeed see if each value of “adr” is covered explicitly. That is where “bins adrbin2[ ] = {[4:5]},” comes into picture. Here “[ ]” tells the simulator to create two explicit bins called adrbin2[1] and adrbin2[2] each covering the two “adr” values =4 and =5. adrbin2[1] will be considered covered if you exercised adr == 4, and adrbin2[2] will be considered covered if adr == 5 is exercised.

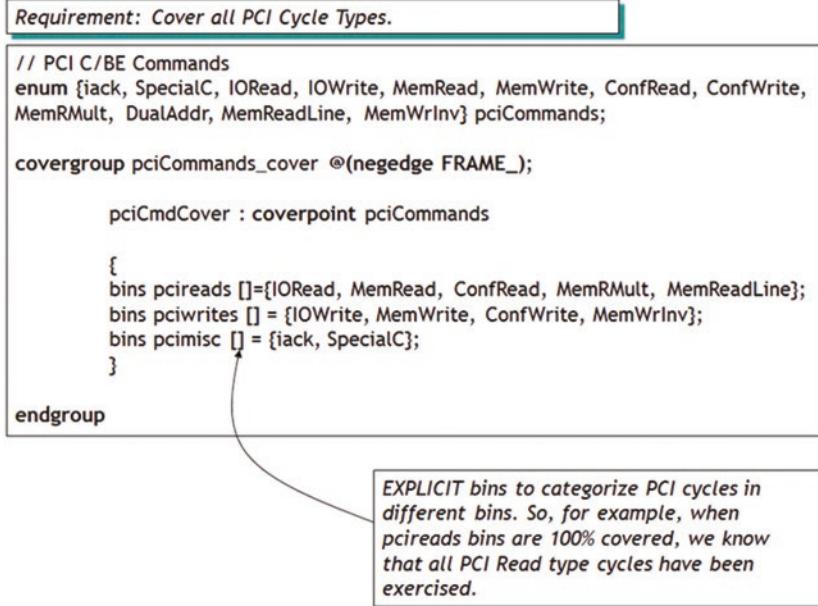
Other ways of creating bins are described in Fig. 15.5 with annotation to describe the nuances. Note that you can have “less” or “more” number of bins than the “adr” values on the RHS of a bin assignment. How will “bins” be allocated in such cases is explained in the figure. Note also the case {[31:\$]} called “bins heretoend.” What does “\$” mean in this case? It means [32:255] since “adr” is an 8-bit variable.

The rest of the semantics is well described with annotation in the figure. Do study them carefully, since they will be very helpful when you start designing your strategy to create “bins.”

Here is another example of how “bins” are created. Take a bit of intuition to understand its semantics:

```
bins Dbins [4] = { [1:10], 11,12,14};
```

Let us see how this works. First, four bins are created since we are explicitly asking for four bins (Dbins[4]). Next, how many values are present on the right-hand side? 13. That is values 1 to 10, value 11, value 12, and value 14. Now, since there are only 4 bins and 13 values, how will these values be spread out among 4 bins?



**Fig. 15.6** Covergroup/coverpoint example with “bins”

Dbins[1] will be considered covered when we hit any of the values 1,2,3  
 Dbins[2] will be considered covered when we hit any of the values 4,5,6  
 Dbins[3] will be considered covered when we hit any of the values 7,8,9  
 Dbins[4] will be considered covered when we hit any of the values 10,11,12,14

Study this carefully so that you are sure how bins are created and how they get covered with different values.

### 15.5.1 Covergroup/Coverpoint Example with “bins”

Recall the example on PCI that we started in previous section. Its story continues here.

In Fig. 15.6, we are assigning different groups of PCI commands to different bins.

Recall that [ ] means auto-generated bins. Hence, for example, since “bins pcireads[ ]” in Fig. 15.6 has five variables (enum type in this example), five bins will be created – one for each enum type. This way of creating bins is very useful because

in the complex maze of features to cover, it is practical to have a clear distinction among different functional groups. Here pcireads[ ] (five bins) covers all PCI read cycles; pciwrites[ ](four bins) covers all PCI write cycles, and for the special cycles, there is the pcimisc[ ] (two bins).

### 15.5.2 “bins” Filtering

“bins” filtering is very important in containing the number of bins created. For example, if you simply did a coverpoint of an 8-bit variable, 256 bins will be created. This gets worse, when we use “cross” (Sect. 15.8) of coverpoints. There are ways to restrict the number of bins created. Using “ignore\_bins,” “illegal\_bins,” etc. will be discussed later in the chapter.

The following shows how the “with” clause comes in handy to restrict the creation of bins.

The “with” clause specifies that only those values that satisfy the given expression are included in the bins. In the expression, the name *myValue* is used to represent the candidate value. The candidate value is of the same type as the coverpoint.

Consider the following example:

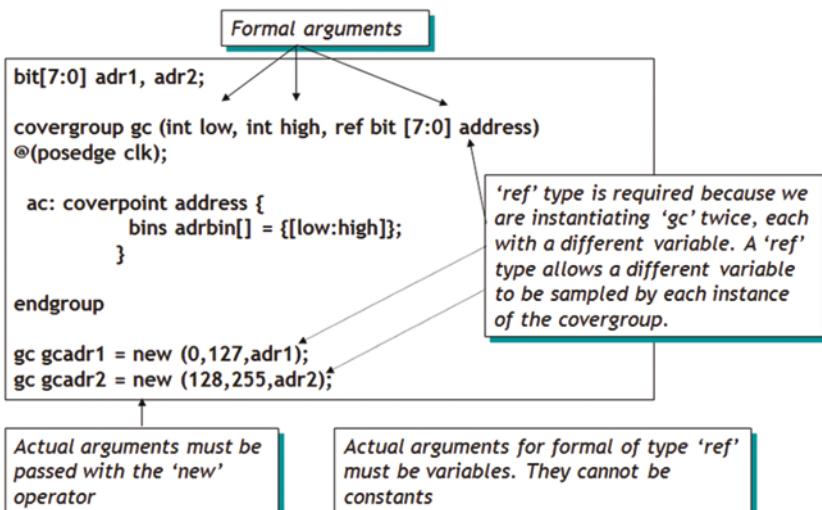


Fig. 15.7 Covergroup: formal and actual arguments

```

int myValue, x;
a: coverpoint x
{
  bins mod6[ ] = {[0:255]} with (myValue % 6 == 0);
}

```

This bin definition selects all values from 0 to 255 that are evenly divisible by 6. One more:

```

a: coverpoint x
{
  bins mod3[ ] = {[0:255]} with ((myValue % 2) || (yourValue
% 3) || no_value);
}

```

In short, filtering expressions using “with” clause provides a powerful and convenient new way to define the bins of your coverpoint. They are especially useful for cross points, where the language rules specify that bins are automatically created for any values that are not otherwise mentioned in the cross-point definition. This means that, in all realistic situations, it is essential for your cross-point definition to specify bins (some of which may be “ignore\_bins” or “illegal\_bins”) that cover every possible combination of values. If you do not do this, you will find the cross contains unwanted auto-generated bins that will distort your coverage figures. In a regular coverpoint, it is possible to use the “default” specification to create an ignored bin that collects all otherwise unspecified values of the coverpoint, but there is no such “default” bin specification for cross points. We will discuss “cross” in Sect. 15.8.

## 15.6 Covergroup: Formal and Actual Arguments

Figure 15.7 outlines the following points:

1. Covergroup can be parameterized for reuse.
2. Actual arguments are passed when you instantiate a covergroup. When the covergroup specifies a list of formal arguments, its instances will provide to the new operator all the actual arguments that are not defaulted. Actual arguments are evaluated when the new operator is executed.
3. This is an example of *reusability*. Instead of creating two covergroups, one for “adr1” and another for “adr2,” we have created only one covergroup called “gc” which has a formal called “address.” We pass “adr1” to “address” in the instance gcadr1 and pass “adr2” to “address” in the instance gcadr2. We also pass the range of adr1 and adr2 to be covered with each instance. In short, it is a good idea

to create parameterizable covergroups, as the situation permits. They can be useful not only within a project but also across projects.

4. An output or inout is illegal as a formal argument.
5. The formal arguments of a covergroup cannot be accessed using a hierarchical name (the formals cannot be accessed outside the covergroup declaration).
6. If an automatic variable is passed by reference, behavior is undefined.
7. “ref” type is required when you pass variables as actual to a formal. In other words, if you were passing a constant, you would not need a “ref” type.

**Exercise:** How many bins will be created for “bins adrbin[ ]” for each instance (“gcadr1” and “gcadr2”) of covergroup “gc”?

## 15.7 SystemVerilog Class-Based Coverage

### 15.7.1 *Class: Embedded Covergroup in a Class*

```
class xyz;
    bit[3:0] m_x;
    bit[7:0] m_y;
    bit m_z;

    covergroup xyzCover @ (m_z);
        coverpoint m_x;
        coverpoint m_y;
    endgroup

    function new ();
        xyzCover = new ;
    endfunction
endclass
```

One of the best places to embed a coverage group is within a “class.” Why a class? Here are some reasons:

- An embedded covergroup can define a coverage model for protected and local class properties without any changes to the class data encapsulation.
- Class members can be used in coverpoint expressions, coverage constructs, option initialization, etc.
- By embedding a coverage group within a class definition, the covergroup provides a simple way to cover a subset of the class properties.
- This style of declaring covergroups allow for modular verification environment development.

Let us see what Sect. 15.7.1 depicts

“covergroup xyzCover” is sampled on any change on variable “m\_z.” This covergroup contains two coverpoints, namely, “m\_x” and “m\_y.” Note that there are no explicit bins specified for the coverpoints. How many bins for each coverpoint will be created? As an exercise, please refer to previous sections to figure out.

Note that covergroup is instantiated within the “class.” That makes sense since the covergroup is embedded within the class. Obviously, if you do not instantiate a covergroup in the “class,” it will not be created, and there will not be any sampling of data.

### ***15.7.2 Multiple Covergroups in a Class***

A “class” can indeed have more than one “covergroup” as shown below:

```
class multi;
  bit [3:0] m_1;
  bit [7:0] m_2;
  bit m_z, m_a;

  covergroup m_xCover @(m_z); coverpoint m_1; endgroup
  covergroup m_yCover @(m_a); coverpoint m_2; endgroup

  function new ();
    m_xCover = new;
    m_yCover = new;
  endfunction

endclass : multi
```

### ***15.7.3 Overriding Covergroups in a Class***

A covergroup defined in a class can indeed be overridden in an extended (child) class. Here is an example of that.

First, let us define a “class parent” and declare a “covergroup pCov.” Note that the “bins parentbins [ ]” creates 256 bins for the “byte pByte”:

```

class parent;
rand byte pByte;

covergroup pCov;
    coverpoint pByte

        //256 bins for 256 values of pByte.
        { bins parentBins [ ] = {[0:255]}; }

endgroup

function new ();
    pCov = new;
endfunction

endclass

```

Next, define a “class child” which extends the “class parent.” In this class, we redefine (i.e., override) “covergroup pCov” of “class parent.” Note that the “bins childBinsAllinOne” creates only one bin for all values of the byte pByte. In other words, if any of the values of pByte is hit, the bin will be covered 100%:

```

class child extends parent;
//'class child' overrides 'covergroup pCov' of 'class parent'
    covergroup pCov;
        coverpoint pByte

        //ONE bin for all values
        { bins childBinsAllinOne = {[0:255]}; }

endgroup

function new ();
    super.new();
    pCov = new;
endfunction

endclass

```

Let us verify this code with a simple testbench:

```

module testOverride;
parent p1;
child c1;

initial begin
p1 = new( );
c1 = new ( );

for (int i=0; i < 10; i++) begin
p1.randomize( );
c1.randomize ( );

p1.pCov.sample( );
c1.pCov.sample( );
#1;

end

$display("p1.pCov Instance Coverage = %0.2f %%", p1.
pCov.get_inst_coverage( ) );
$display("c1.pCov Instance Coverage = %0.2f %%", c1.
pCov.get_inst_coverage( ) );

end
endmodule

```

*Simulation log:*

```

p1.pCov Instance Coverage = 3.52 %
c1.pCov Instance Coverage = 100 %

```

Here is the interpretation of the simulation log. “class parent” has “bins parentBins [ ] = {[0:255]};” which means you need to hit all 256 values of pByte to cover all 256 bins. With our randomization of pByte from the “for loop,” we hit only 3.52% of the 256 values, and hence the instance coverage for “p1.pCov” is 3.52%.

In “class child,” we overrode the “covergroup pCov” and used “bins childBinsAllinOne = {[0:255]};” to declare a single bin that covers all the 256 values of “pCov.” In other words, our randomization hit at least one value of pByte, and the “c1.pCov” instance was 100% covered.

This proves that we can indeed override a covergroup definition in an extended class.

- 'cross' coverage is specified between two (or more) coverpoints or variables.
- 'cross' of N coverpoints is defined as the coverage of all combinations of all bins associated with the N coverpoints.
- 'cross' coverage is allowed only between coverpoints defined within the same covergroup.
- Expressions cannot be used directly in a cross

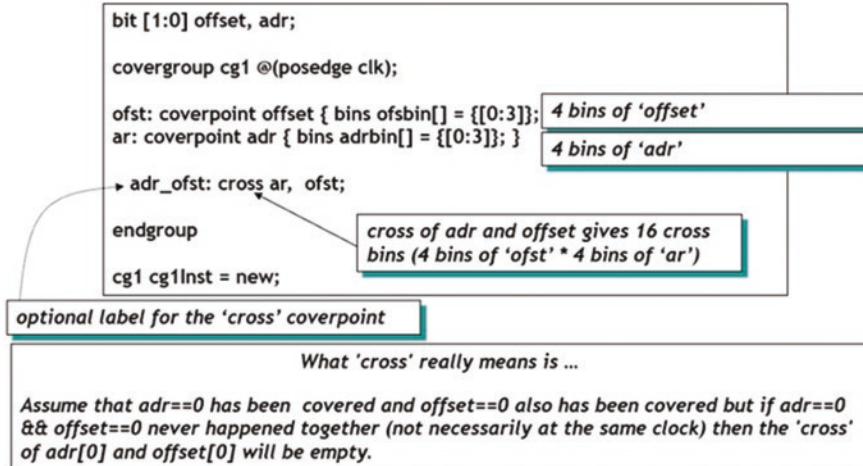


Fig. 15.8 "cross" coverage: basics

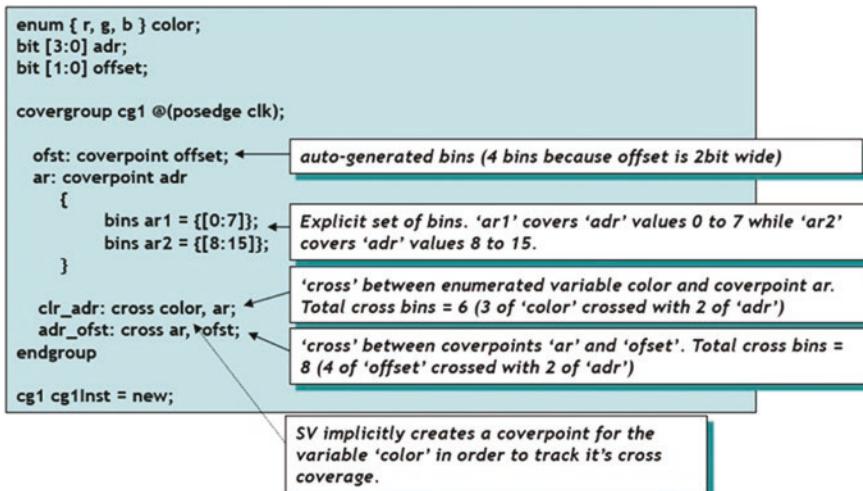


Fig. 15.9 "cross": example

### 15.7.4 Parameterizing Coverpoints in a Class

This section describes a simple way of parameterizing coverpoints through a parameterized class. Note that these parameters do not affect the way the component fits into its enclosing environment. They affect only the way its covergroup is built and used. This class can then be instantiated with different LBound and HBound parameters that will affect how coverpoint bins are created:

Covergroup	Metric	Goal / Status
<b>TYPE /test_covergroup_cross/cg1</b>		96.7% 100
<b>Uncovered</b>		
<b>Coverpoint cg1::ofst</b>	100.0%	100 Covered
<b>bin auto[0]</b>	8	1 Covered
<b>bin auto[1]</b>	3	1 Covered
<b>bin auto[2]</b>	5	1 Covered
<b>bin auto[3]</b>	4	1 Covered
<b>Coverpoint cg1::ar</b>	100.0%	100 Covered
<b>bin ar1</b>	11	1 Covered
<b>bin ar2</b>	9	1 Covered
<b>Coverpoint cg1::color</b>	100.0%	100 Covered
<b>bin auto[r]</b>	2	1 Covered
<b>bin auto[g]</b>	1	1 Covered
<b>bin auto[b]</b>	17	1 Covered
<b>Cross cg1::clr_adr</b>	83.3%	100 Uncovered
<b>bin &lt;auto[r],ar1&gt;</b>	1	1 Covered
<b>bin &lt;auto[g],ar1&gt;</b>	1	1 Covered
<b>bin &lt;auto[b],ar1&gt;</b>	9	1 Covered
<b>bin &lt;auto[r],ar2&gt;</b>	1	1 Covered
<b>bin &lt;auto[g],ar2&gt;</b>	0	1 ZERO
<b>bin &lt;auto[b],ar2&gt;</b>	8	1 Covered
<b>Cross cg1::adr_ofst</b>	100.0%	100 Covered
<b>bin &lt;ar1,auto[0]&gt;</b>	4	1 Covered
<b>bin &lt;ar1,auto[1]&gt;</b>	2	1 Covered
<b>bin &lt;ar1,auto[2]&gt;</b>	3	1 Covered
<b>bin &lt;ar1,auto[3]&gt;</b>	2	1 Covered
<b>bin &lt;ar2,auto[0]&gt;</b>	4	1 Covered
<b>bin &lt;ar2,auto[1]&gt;</b>	1	1 Covered
<b>bin &lt;ar2,auto[2]&gt;</b>	2	1 Covered
<b>bin &lt;ar2,auto[3]&gt;</b>	2	1 Covered

Fig. 15.10 “cross” example: simulation log

```

class useParams #(parameter int LBound=1, HBound=16);
  bit [15:0] addr;
  bit [7:0] data;
  covergroup cGroup;
    caddr: coverpoint addr {
      bins b1 [ ] = {[LBound : HBound]};
      illegal_bins il = {[LBound-1:HBound-8]};
    }
    cdata: coverpoint data;
  endgroup
  //constructor
  function new;
    cGroup = new;
  endfunction
  ....
endclass

useParams useP1 = new (32,64);

```

Here is another example on the same line of thought:

```

class configurable_cGroup_class;
  int x;
  covergroup construct_bins_cg (input int min, max,
num_bins);
  coverpoint x {
    bins lowest = {min};
    bins highest = {max};
    bins middle[num_bins] = {[min+1:max-1]};
  }
endgroup
function new(int min, int width);
  int max = min + width - 1;
  int n_bins = (width - 2)/4 ; // aim for 4 values per bin
  construct_bins_cg = new(min, max, n_bins);
endfunction
endclass
configurable_cGroup_class cgClass = new(16, 32);

```

## 15.8 “cross” Coverage

“cross” is a very important feature of functional coverage. This is where code coverage completely fails. Figure 15.8 describes the syntax and semantics.

## Syntax:

```
cover_cross ::= [ cross_identifier : ] cross list_of_cross_
items [ iff ( expression ) ] cross_body
```

Two variables “offset” and “adr” are declared. Coverpoint for “offset” creates four bins called ofsbin[0] ... ofsbin[3] for the four values of “offset,” namely, 0, 1, 2, and 3. Coverpoint “adr” also follows the same logic and creates adrbin[0]...adrbin[3] for the four values of “adr,” namely, 0, 1, 2, and 3.

adr\_ofst is the label given to the “cross” of ar and ofst. First of all, the cross” of “ar” (label for coverpoint adr) and “ofst” (label for coverpoint offset) will create another set of 16 bins ( four bins of “adr” \* four bins of “offset”). These “cross” bins will keep track of the result of “cross.” However, what does “cross” mean?

Four values of “adr” need to be covered (0, 1, 2, 3). Let us assume adr == 2 has been covered (i.e., adrbin[2] is covered). Similarly, there are four values of “offset” that need to be covered (0, 1, 2, 3) and that offset == 0 has also been covered (i.e., ofsbin[0] has been covered). However, have we covered “cross” of adr=2 (adrbin[2]) and offset=0 (ofsbin[0])? Not necessarily. “cross” means that adr=2 and offset=0 must be true “together” at some point in time. This does *not* mean that they need to be “covered” at the *same* point in time. It simply means that, e.g., if adr == 2 that it should remain at that value until offset == 0 (or vice versa). This will make both of them true “together.” If that is the case, then the “cross” of adrbin[2] and ofsbin[0] will be considered “covered.”

Here is an example:

Figure 15.9 shows how cross is achieved between an enum type and a coverpoint. Idea is to show how cross coverpoint/bins are calculated. The figure describes different ways “cross” bins are calculated.

Recall that a “cross” can be defined only between N coverpoints, meaning you must have an explicit coverpoint for a variable in order to cross with another coverpoint. However, note that the enum-type “color” has no coverpoint defined for it. Yet, we are able to use it in “cross.” That is because the language semantics implicitly creates a coverpoint for the enum-type “color” (or any other type of variable) and tracks its cross coverage.

Please study this and other figures carefully since they will act as guideline for your projects. A simulation log of this example is presented in Fig. 15.10. The annotations describe what is going on.

**Important Note:** *Cross coverage is allowed only between coverage points defined within the same covergroup. Coverage points defined in a coverage group other than the one enclosing the “cross” cannot participate in a cross. Attempts to cross items from different coverage groups will result in a compiler error.*

Note that “cross” is not limited to only two coverpoints/variables. You can have multiple coverpoints/variables crossed with each other.

For example, if you want to make sure that bits of a “control\_reg” are all covered together at some point in time, you can create a “cross” as follows:

**A very important feature of functional coverage is the ability to see if required ‘transitions’ in a design have been exercised.**

**This temporal domain coverage is not possible with code coverage (except for state transition coverage which is restricted strictly to the state machines ‘derived’ by the code coverage tool).**

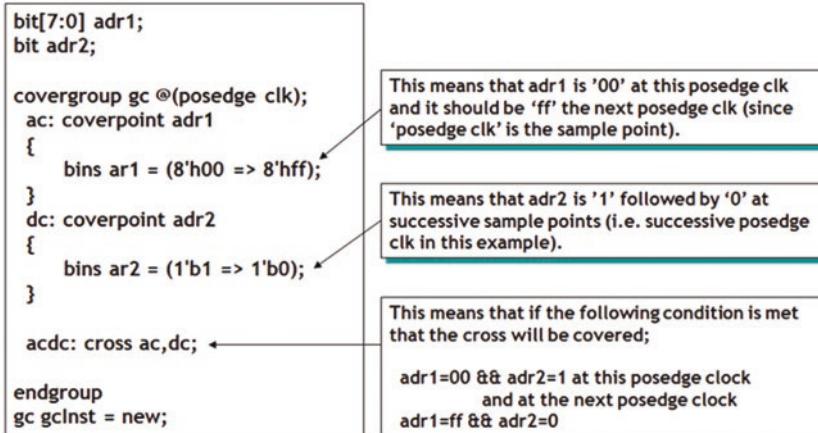


Fig. 15.11 “bins” for transition coverage

```

covergroup control_reg_cg() with function sample(bit[4:0]
control_reg);
    option.name = "control_reg_cg";

    CR1: coverpoint control_reg[0];
    CR2: coverpoint control_reg [1];
    CR3: coverpoint control_reg [2];
    CR4: coverpoint control_reg [3];
    CR5: coverpoint control_reg [4];

    multiCross: cross CR1, CR2, CR3, CR4, CR5;
endgroup

```

In this example, we are creating a “cross” of five coverpoints. This will ensure that each bit of control\_reg is covered and that they are all covered together.

Further examples of “cross” coverage:

**Example 1:**

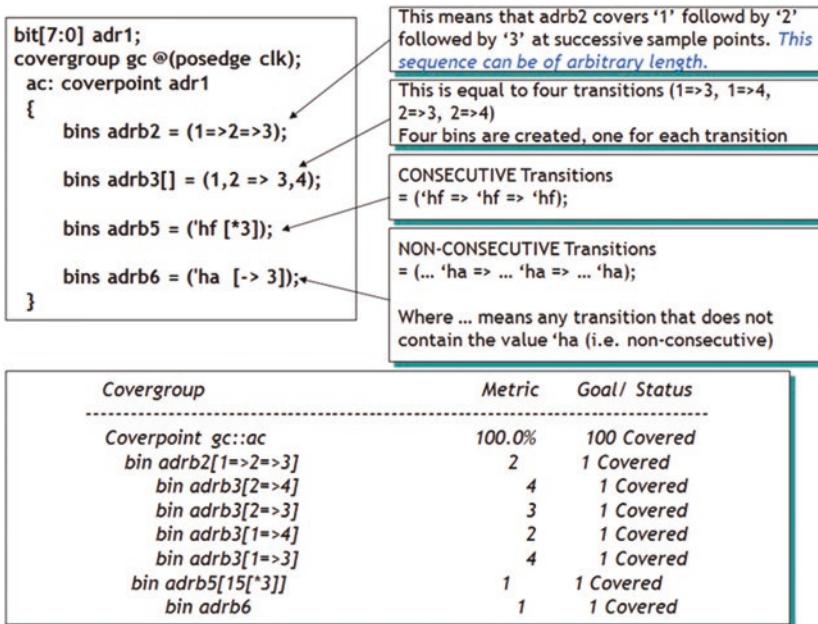


Fig. 15.12 “bins” for transition: further nuances

*Requirement: Cover all PCI Cycle Types and transitions among Read and Write cycles.*

```

// PCI C/BE Commands
enum {iack, SpecialC, IORead, IOWrite, MemRead, MemWrite, ConfRead, ConfWrite,
MemRMult, DualAddr, MemReadLine, MemWrInv} pciCommands;

covergroup pciCommands_cover @(posedge clk);
    pciCmdCover : coverpoint pciCommands
    {
        bins pcireads [] = {IORead, MemRead, ConfRead, MemRMult, MemReadLine};
        bins pciewrites [] = {IOWrite, MemWrite, ConfWrite, MemWrInv};
        bins pcimisc [] = {iack, SpecialC};

        bins R2W [] = (IORead, MemRead, ConfRead, MemRMult, MemReadLine =>
IOWrite, MemWrite, ConfWrite, MemWrInv);
        bins W2R [] = (IOWrite, MemWrite, ConfWrite, MemWrInv => IORead,
MemRead, ConfRead, MemRMult, MemReadLine);
    }
endgroup

```

Fig. 15.13 Example of PCI cycles transition coverage

```

bit [7:0] b1, b2, b3;
covergroup cov @(posedge clk);
    b2b3: coverpoint b2 + b3;;
    b1Crossb2b3 : cross b1, b2b3;

```

```
endgroup
```

The thing to note in this example is that b2b3 coverpoint is an expression. It is indeed possible to “cross” between a coverpoint (which is an expression) and a variable.

**Example 2:**

```
bit [3:0] bNibble;
bit [7:0] a_var;
covergroup myCov @(posedge clk);
    A: coverpoint a_var { bins yy[ ] = { [0:9] }; }
    CC: cross bNibble, A;
endgroup
```

In this example, coverpoint of a\_var (labeled “A”) creates explicitly created 10 bins (yy[0], yy[1]...yy[9]), while bNibble creates auto-generated 16 bins (auto[0], auto[1], .... auto[15]).

So, the cross of bNibble and A produces 160 cross bins (16 auto-generated bins of “bNibble” x 10 bins of “a\_var”):

```
<auto[0], yy[0]>
<auto[0], yy[1]>
...
<auto[0], yy[9]>

<auto[1], yy[0]>
...
<auto[15], yy[9]>
```

**Example 3:** Some simple do’s and do not’s of “cross.” Please refer to Sect. 15.13 for the description of “binsof.”

Consider the following code:

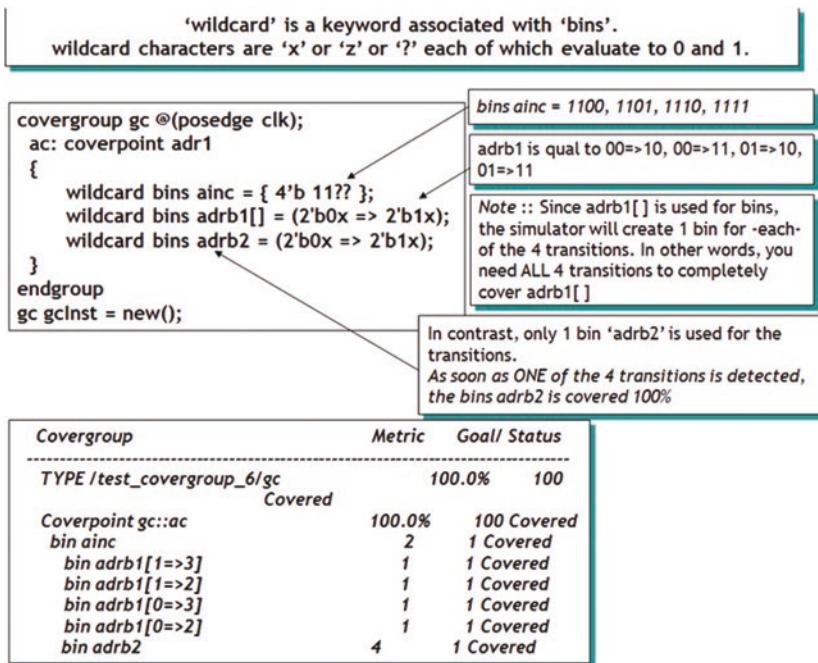


Fig. 15.14 “wildcard bins”

```

A: coverpoint a {
  bins b1[ ] = {2, [4:5]};
  bins b2[ ] = {6, 7};
  bins b3[ ] = default;
}
B: coverpoint b {
  bins b1[ ] = {3, [7:9]};
}
C: coverpoint c {
  bins b1[ ] = {1,4};
}
cross A, B {
  bins cb1 = binsof (A); // correct use
  bins cb2 = binsof (A.b1); // correct use
  bins cb3 = binsof (B.b2); //incorrect use as b2 is not a
  bin of B
  bins cb4 = binsof (C); //incorrect use as cross does not
  include C
}
//The expression within the binsof construct should not be a
default bin.

```

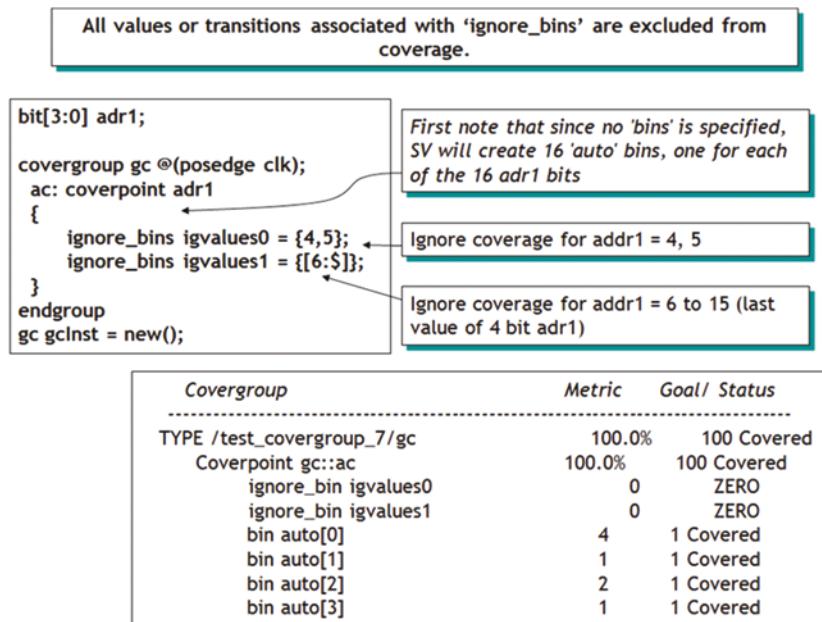


Fig. 15.15 “ignore\_bins”

```

bins cb5 = binsof(A.b3);
//incorrect use because b3 is a default bin

//You cannot declare default bins inside a cross.
cross A, B {
    bins cb1 = default; // incorrect use
}

```

Please refer to Sect. 15.13 for the description of “binsof.”

## 15.9 “bins” for Transition Coverage

As noted in Fig. 15.11, this is an especially useful feature of functional coverage. Transaction-level transitions are very important to cover. For example, did CPU issue a read followed by write invalid? Did you issue a D\$miss followed by a D\$hit cycle? Such transitions are where the bugs occur, and we want to make sure that we have indeed exercised such transitions.

Note that we are addressing both the “transition” and the “cross” of “transition” coverage.

There are two transitions in the example:

`bins ar1 = (8'h00 => 8'hff)`, which means that adr1 should transition from “0” to “ff” on two consecutive posedge of clk. In other words, the testbench must exercise this condition for it to be covered.

Similarly, there is the “bins ar2” that specifies the transition for adr2 ( $1 \Rightarrow 0$ ).

The cross of transitions is shown at the bottom of the figure. Very interesting how this works. Take the first values of each transition (viz.,  $\text{adr1}=0 \&\& \text{adr2}=1$ ). This will be the start points of cross transition at the posedge clk. If at the next (posedge clk) values are  $\text{adr1}='ff' \&\& \text{adr2}=0$ , the cross transition is covered.

More on the “bins” of transition is shown in the Fig. 15.12. In the figure, different styles of transitions have been shown. “bins adrb2” requires that “adr1” should transition from  $1 \Rightarrow 2 \Rightarrow 3$  on successive posedge clk. Of course, this transition sequence can be of arbitrary length. “bins adrb3[ ]” shows another way to specify multiple transitions. The annotation in the figure explains how we get four transitions.

“bins adrb5” is (in some sense) analogous to the consecutive operator of assertions. Here ‘hf [\*3]’ means that  $\text{adr1}='hf$  should repeat three times at successive posedge clk.

Similarly, the non-consecutive transition (‘ha [->3]’) means that adr1 should be equal to ‘ha, 3 times and not necessarily at consecutive posedge clk. Note that just as in non-consecutive operator, here also ‘ha need to arrive 3 times with arbitrary number of clocks in-between their arrival and that ‘adr1’ should *not* have any other value in between these three transitions. The simulation log shows the result of a testbench that exercises all the transition conditions.

Now let us turn back to our favorite PCI example that we have been following. We started with simple coverage, moved to “bins” coverage, and now we will see the “transition” coverage. This is the reason we were building on the same example showing how such a coverage model can be derived starting with a simple model.

Figure 15.13 shows the same enum type and same bins. But in addition, it now defines two transition bins named R2W (for read to write) and W2R (for write to read).

`bins R2W[ ]` means that all possible read cycle types are followed by all possible write cycles. Each of the transition covers a separate transition to cover all possible cycle transitions on a PCI bus.

In this example, we must cover the following transactions for R2W[ ]:

```

    IORRead => IOWWrite; IORRead => MemWWrite; IORRead =>
ConfWWrite; IORRead => MemWrInv;
        MemRead => IOWWrite; MemRead => MemWWrite; MemRead
=> ConfWWrite; MemRead => MemWrInv;
        ConfRead => IOWWrite; ConfRead => MemWWrite; ConfRead
=> ConfWWrite; ConfRead => MemWrInv;
        MemRMult => IOWWrite; MemRMult => MemWWrite; MemRMult
=> ConfWWrite; MemRMult => MemWrInv;
        MemReadLine => IOWWrite; MemReadLine => MemWWrite;
MemReadLine => ConfWWrite; MemReadLine => MemWrInv;

```

Same type of transitions are covered by bins W2R[ ].

## 15.10 “wildcard bins”

Since no one likes to type a sequence repeatedly, we create do not care (or as the functional coverage lingo calls it “wildcard” bins). As shown in Fig. 15.14, you can use either an “x” or a “z” or “?” (doesn’t this look familiar to Verilog?) to declare “wildcard” bins. Note that such bins must precede with the keyword “wildcard.”

“wildcard bins ainc” specifies that adr1 values 1100, 1101, 1110, and 1111 need to be covered.

Note also “wildcard bins adr1[ ]” and “wildcard bins adr2.” One creates implicit ([ ]) four bins, while the other creates only one bin. The one that creates four explicit bins will check to see that – *each of the* – four transitions take place, while adr2 that creates only one bin will be considered covered if – *any* – of the four transitions take place.

## 15.11 “ignore\_bins”

“ignore\_bins” is very useful when you have a large set of bins to be defined. Instead of defining every one of those, if you can identify the ones that are not of interest, then you can simply define “ignore\_bins.” The ones that are specified to be ignored will indeed be ignored and rest will be covered. “ignore\_bins” is especially useful with “cross” since a “cross” can create a huge number of bins and there must be a way to restrict those bins using some mechanism such as “ignore\_bins.”

As we noted at the onset of this chapter, when there is no explicit “bins” defined for a coverpoint, the simulator creates all possible bins that the “covered” variable requires. In Fig. 15.15, only “ignore\_bins” are specified in the coverpoint adr1 but no “bins.” This means that the simulator will first create all 16 bins (adr1 is 4-bit wide) for adr1. Then it will ignore value 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, and 15

<p>All values or transitions associated with ‘illegal_bins’ are excluded from coverage and will give a run time ERROR if encountered.</p>																																																								
<pre>bit[3:0] adr1;  covergroup gc @ (posedge clk);     ac: coverpoint adr1     {         illegal_bins ilvalues0 = {0};     } endgroup gcinst = new();</pre>	<p>First note that since no ‘bins’ is specified, SV will create 1 ‘auto’ bin for each of the 16 adr1 bits</p>																																																							
	<p>If adr1 == ‘0’ ever during simulation, this will give a run time ERROR</p>																																																							
	<p># ** Error: Illegal range bin value=b0000 got covered. illegal_bins ilvalues0 = {0};</p>																																																							
<table border="1"> <thead> <tr> <th>TYPE /test_covergroup_7/gc</th> <th>33.3%</th> <th>100 Uncovered</th> </tr> </thead> <tbody> <tr> <td>Coverpoint gc::ac</td> <td>33.3%</td> <td>100 Uncovered</td> </tr> <tr> <td>    <b>illegal_bin ilvalues0</b></td> <td><b>4</b></td> <td><b>Occurred</b></td> </tr> <tr> <td>        bin auto[1]</td> <td>2</td> <td>1 Covered</td> </tr> <tr> <td>        bin auto[2]</td> <td>2</td> <td>1 Covered</td> </tr> <tr> <td>        bin auto[3]</td> <td>2</td> <td>1 Covered</td> </tr> <tr> <td>        bin auto[4]</td> <td>0</td> <td>1 ZERO</td> </tr> <tr> <td>        bin auto[5]</td> <td>0</td> <td>1 ZERO</td> </tr> <tr> <td>        bin auto[6]</td> <td>0</td> <td>1 ZERO</td> </tr> <tr> <td>        bin auto[7]</td> <td>0</td> <td>1 ZERO</td> </tr> <tr> <td>        bin auto[8]</td> <td>0</td> <td>1 ZERO</td> </tr> <tr> <td>        bin auto[9]</td> <td>2</td> <td>1 Covered</td> </tr> <tr> <td>        bin auto[10]</td> <td>0</td> <td>1 ZERO</td> </tr> <tr> <td>        bin auto[11]</td> <td>1</td> <td>1 Covered</td> </tr> <tr> <td>        bin auto[12]</td> <td>0</td> <td>1 ZERO</td> </tr> <tr> <td>        bin auto[13]</td> <td>0</td> <td>1 ZERO</td> </tr> <tr> <td>        bin auto[14]</td> <td>0</td> <td>1 ZERO</td> </tr> <tr> <td>        bin auto[15]</td> <td>0</td> <td>1 ZERO</td> </tr> </tbody> </table>			TYPE /test_covergroup_7/gc	33.3%	100 Uncovered	Coverpoint gc::ac	33.3%	100 Uncovered	<b>illegal_bin ilvalues0</b>	<b>4</b>	<b>Occurred</b>	bin auto[1]	2	1 Covered	bin auto[2]	2	1 Covered	bin auto[3]	2	1 Covered	bin auto[4]	0	1 ZERO	bin auto[5]	0	1 ZERO	bin auto[6]	0	1 ZERO	bin auto[7]	0	1 ZERO	bin auto[8]	0	1 ZERO	bin auto[9]	2	1 Covered	bin auto[10]	0	1 ZERO	bin auto[11]	1	1 Covered	bin auto[12]	0	1 ZERO	bin auto[13]	0	1 ZERO	bin auto[14]	0	1 ZERO	bin auto[15]	0	1 ZERO
TYPE /test_covergroup_7/gc	33.3%	100 Uncovered																																																						
Coverpoint gc::ac	33.3%	100 Uncovered																																																						
<b>illegal_bin ilvalues0</b>	<b>4</b>	<b>Occurred</b>																																																						
bin auto[1]	2	1 Covered																																																						
bin auto[2]	2	1 Covered																																																						
bin auto[3]	2	1 Covered																																																						
bin auto[4]	0	1 ZERO																																																						
bin auto[5]	0	1 ZERO																																																						
bin auto[6]	0	1 ZERO																																																						
bin auto[7]	0	1 ZERO																																																						
bin auto[8]	0	1 ZERO																																																						
bin auto[9]	2	1 Covered																																																						
bin auto[10]	0	1 ZERO																																																						
bin auto[11]	1	1 Covered																																																						
bin auto[12]	0	1 ZERO																																																						
bin auto[13]	0	1 ZERO																																																						
bin auto[14]	0	1 ZERO																																																						
bin auto[15]	0	1 ZERO																																																						

Fig. 15.16 “illegal\_bins”

and cover only adr1=0, 1, 2, and 3. This is reflected in the simulation log at the bottom of Fig. 15.15.

Some examples of ignore\_bins.

**Example 1:** If “ignore\_bins” is specified with a guard expression (iff), then it is effective only if the guard expression is true at the time of sampling. Consider the following code:

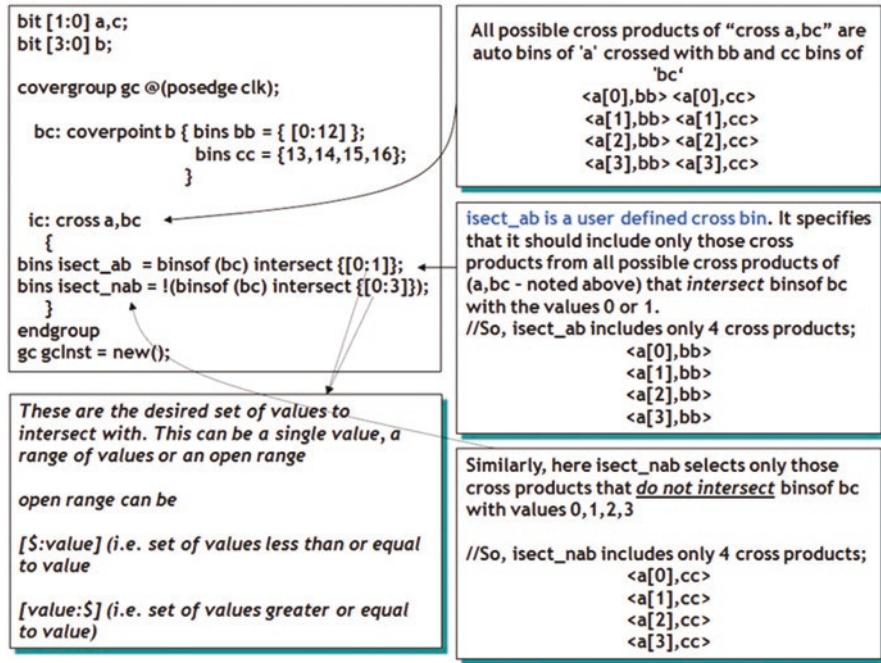


Fig. 15.17 “binsof” and “intersect”

```

covergroup cg1 @(posedge clk);
coverpoint a {
    bins b1[ ] = {0, 1, 2, 3};
    ignore_bins ignore_vals = {0, 3} iff c + d ;
}
endgroup

```

If during a simulation run, coverpoint “a” takes values 0, 1, 2, and 3, values 0 and 3 are ignored if and only if  $c+d$  is true at the point of sampling. If  $c+d$  is true throughout the simulation, the coverage percentage is reported as 2/4, which is 50%. This is what the simulators that I have tried have reported. My personal take is that if certain bins are ignored, then they should not be part of the coverage report, and the report in this case should be 100%. But then, I do not design simulators!!!

If the value of expression  $c+d$  is false at the time of sampling and the sampled values are 0 or 3, then these values are not ignored, and the count corresponding to bins  $b1[0]$  and  $b1[3]$  is incremented. In this situation, coverage percentage is reported as 4/4, which is 100%.

**Example 2:** The ignore\_bins specification in the bin declaration of a coverpoint may result in “empty” bins for a coverpoint. An empty bin is essentially a user-defined/automatic bin of a coverpoint for which all values are ignored using ignore\_

bins. If ignore\_bins specification results in empty bins for a coverpoint, then empty bins will not be dumped in the coverage database. For example:

```
covergroup cg1 @ (posedge clk);
coverpoint a {
  bins b1[ ] = {0, 1, 2, 3};
  bins b2 = {[5:8], 9};
  ignore_bins ignore_vals = {{0, 3, 1, 2}};
}
endgroup
```

In the above example, bin “b1” is an empty bin and will not be dumped because all of the values associated with the bin “b1” are specified as ignore\_bins.

#### Example 3:

```
covergroup cg1 @ (posedge clk);
coverpoint a {
  bins b1[ ] = {0, 1, 2, 3};
  ignore_bins ignore_vals = {0, 3};
}
endgroup
```

In the above example, if the sampled value of coverpoint “a” is 3, it is ignored, and the count for bin b1[3] is not incremented. If during a simulation run, coverpoint “a” takes values 0, 1, 2, and 3, the coverage percentage is reported as 2/2, which is 100%. The values specified with ignore\_bins have been removed from the total bin count.

**Example 4:** For transition bins, even if a transition occurs, it cannot be considered hit if it is specified with ignore\_bins. Consider the following code:

```
A: coverpoint a {
  bins b1[ ] = (2=>5=>1), (1=>4=>3);
  ignore_bins ignore_trans = (2=>5);
}
```

In this case, even if the transition  $2 \Rightarrow 5 \Rightarrow 1$  takes place, it will not be considered a hit, since transition  $2 \Rightarrow 5$  is ignored.

**Example 5:** A simple example shows the use of “with” clause and ignore\_bins and how they interact:

```

module top;
reg[15:0] CC;
covergroup cg@(posedge clk);
withC : coverpoint CC {
    bins bin_b1[ ] = {[0:20]} with (item % 2 == 0);
    ignore_bins gCC = CC with (item % 3 == 0);
}
endgroup
.....
endmodule

```

The “bins\_b1” of coverpoint “CC” will create the following bins:

```
bins bin_b1[ ] = {0,2,4,6,8,10,12,14,16,18,20};
```

and ignore\_bins “gCC” will ignore following bins:

```
ignore_bins gCC = {0,3,6,9,12,15,18,21,24,27 ...};
```

So, after ignore\_bins is applied to coverpoint “CC,” the following bins will remain and will be used in coverage report:

```

cg (covergroup)
    CC (coverpoint)
        bins of 'CC' (after ignore_bins) = bins_b1[2],
        bins_b1[4], bins_b1[8], bins_b1[10], bins_b1[14], bins_b1[16],
        bins_b1[20]

```

**Example 6:** This is an example simply to illustrate that “ignore\_bins” is very useful for weeding out bins from a “cross.” As you know “cross” can create a large number of bins (depending on what you are crossing, of course), and ignore\_bins will help you keep only the bins of interest. For example, you declare two coverpoints for variables x and y. Their cross will create a large number of bins. But you do not care for those bins where  $x > y$ . Here is how you do it:

```

int x, y;
covergroup xyCG;
    x_c : coverpoint x;
    y_c : coverpoint y;

    x_y_cross: cross x_c, y_c {
        ignore_bins ignore_x_GT_y = x_y_cross with (x_c > y_c);
    }
endgroup

```

## 15.12 “illegal\_bins”

“illegal\_bins” is interesting in that it will give an error, if you *do* cover a given scenario. In Fig. 15.16, we refer to the same old “adr1.” Sixteen auto-bins are created for coverpoint “adr1” since we do not explicitly declare any bins for it. And we say that if the testbench ever hits adr1 == 0, it should be considered illegal. Coverage of adr1 == 0 should not occur. As seen from the simulation log, coverage of adr1==0 results in an error.

Illegal bins take precedence over any other bins, that is, they will result in a runtime error even if they are included in another bin. Illegal transition bins cannot specify a sequence of unbounded or undetermined varying length.

Note that “illegal\_bins” takes precedence of “ignore\_bins.” Here is an example:

```

covergroup cg1 @(posedge clk);
    coverpoint a {
        bins b1[ ] = {0, 1, 2, 3};
        illegal_bins ill = {1, 2};
        ignore_bins ign = {2,3};
    }
endgroup

```

In this example, value 2 is specified both as ignore\_bins and illegal\_bins. When simulation hits the value 2, an error will be reported in the coverage report, since illegal\_bins takes precedence over “ignore\_bins.”

## 15.13 “binsof” and “intersect”

Figure 15.17 shows that “coverpoint b” has two bins; one is called “bb” and has value from 0 to 12 of “b.” bins cc on the other hand has values 13, 14, 15, and 16 of “b” that need to be covered. Then we declare a “cross” of a and bc. So, let us first see what this cross looks like.

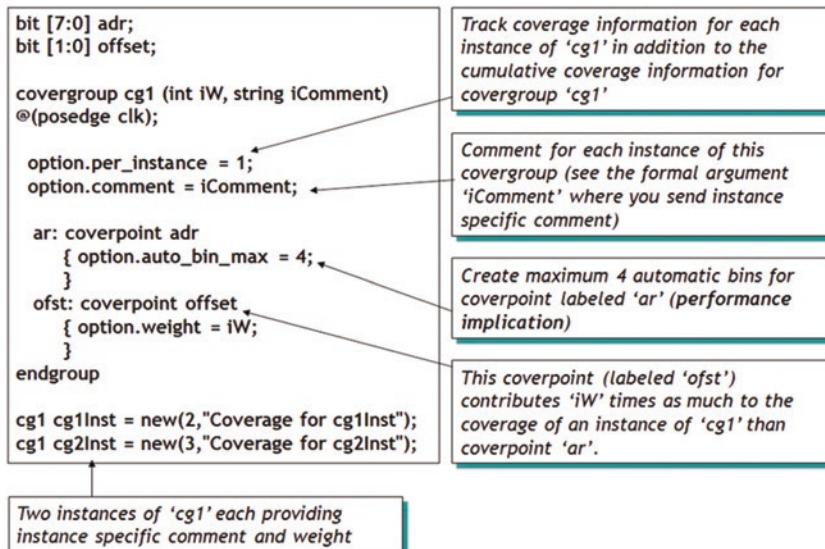


Fig. 15.18 Coverage options: instance-specific example

“a” has four implicit bins a[0], a[1], a[2], and a[3] for four values 0, 1, 2, and 3, and “bc” has two bins “bb” and “cc.” So, the “cross a, bc” produces:

```

<a[0],bb> <a[0],cc>
<a[1],bb> <a[1],cc>
<a[2],bb> <a[2],cc>
<a[3],bb> <a[3],cc>

```

We are moving along just fine – right? But now it gets interesting. We do not want to cover all “cross” bins into our “cross” of “a” and “bc.” We want to “intersect” them and derive a new subset of the total “cross” set of bins:

```
bins insect_ab = binsof (bc) intersect {[0:1]};
```

says that take all the bins of “bc” (which are “bb” and “cc”) and intersect them with the “values” 0 and 1. Now note carefully that “bb” carries values {[0:12]} which includes the values 0 and 1, while “cc” does not cover 0 or 1. Hence, only those cross products of “bc” that cover “bb” are included in this intersect. Note that the below-mentioned subset is selected from the “cross” product shown above:

```

<a[0],bb>,
<a[1],bb>,
<a[2],bb>,
<a[3],bb>

```

Now onto the next intersect:

```
bins isect_nab = ! (binsof (bc) intersect {[0:3]})
```

Similarly, first, note that here we are using negation of binsof. So, this “bins” statement says take the intersect of binsof (bc) and {[0:3]} and discard them from the “cross” of a and bc. Keep only those that do not intersect. Note that {[0:3]} again falls into the bins “bb” and would have resulted in exactly the same set that we saw for the non-negated intersect. Since this one is negated, it will ignore cross with “bb” and only pick the “bins cc” from the original “cross” of a and bc:

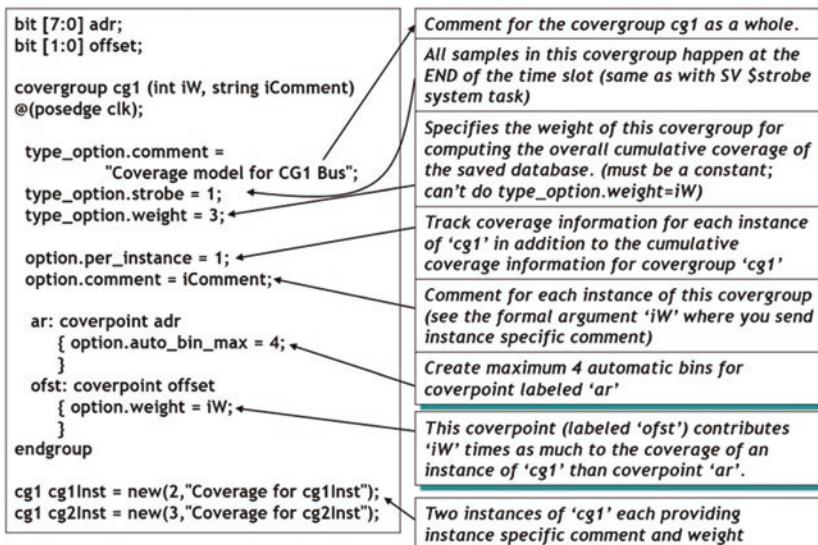


Fig. 15.19 Coverage options for “covergroup” type

**Table 15.1** Coverage options: instance-specific per-syntactic level

The following table summarizes the syntactical level (*covergroup*, *coverpoint*, or *cross*) at which instance options can be specified. All instance options can be specified at the *covergroup* level. Except for the *weight*, *goal*, *comment*, and *per\_instance* options, all other options set at the *covergroup* syntactic level act as a default value for the corresponding option of all *coverpoints* and *crosses* in the *covergroup*. Individual *coverpoint* or *crosses* can overwrite these default values. When set at the *covergroup* level, the *weight*, *goal*, *comment*, and *per\_instance* options do not act as default values to the lower syntactic levels.

Option Name	Allowed in Syntactic Level		
	<i>covergroup</i>	<i>coverpoint</i>	<i>cross</i>
<i>name</i>	Yes	No	No
<i>weight</i>	Yes	Yes	Yes
<i>goal</i>	Yes	Yes	Yes
<i>comment</i>	Yes	Yes	Yes
<i>at_least</i>	Yes (default for <i>coverpoints</i> & <i>crosses</i> )	Yes	Yes
<i>detect_overlap</i>	Yes (default for <i>coverpoints</i> )	Yes	No
<i>auto_bin_max</i>	Yes (default for <i>coverpoints</i> )	Yes	No
<i>cross_auto_bin_max</i>	Yes (default for <i>crosses</i> )	No	Yes
<i>cross_num_print_missing</i>	Yes (default for <i>crosses</i> )	No	Yes
<i>per_instance</i>	Yes	No	No

LRM: SystemVerilog 3.1a,  
Table 20-2

```
<a[0],cc>,
<a[1],cc>,
<a[2],cc>,
<a[3],cc>
```

## 15.14 User-Defined “sample( )” Method

**Table 15.2** Coverage group-type (static) options

Option name	Default	Description
<b>weight = constant_number</b>	1	If set at the covergroup syntactic level, it specifies the weight of this covergroup for computing the overall cumulative (or type) coverage of the saved database. If set at the coverpoint (or cross) syntactic level, it specifies the weight of a coverpoint (or cross) for computing the cumulative (or type) coverage of the enclosing covergroup. The specified weight shall be a non-negative integral value.
<b>goal = constant_number</b>	100	Specifies the target goal for a covergroup type or for a coverpoint or cross of a covergroup type.
<b>comment = string_literal</b>	”””	A comment that appears with the covergroup type or with a coverpoint or cross of the covergroup type. The comment is saved in the coverage database and included in the coverage report.
<b>strobe = boolean</b>	0	When true, all samples happen at the end of the time slot, like the \$strobe system task.
<b>merge_instances = boolean</b>	0	When true, cumulative (or type) coverage is computed by merging instances together as the union of coverage of all instances. When false, type coverage is computed as the weighted average of instances.
<b>distribute_first = boolean</b>	0	When true, instructs the tool to perform value distribution to the bins prior to application of the <i>with_covergroup_expression</i> .

As we saw in the previous sections, you can explicitly call a sample( ) method in lieu of providing a clocking event directly when a covergroup is declared. But what if you want to parameterize the built-in sample( ) method and pass it exactly the data that you want sampled? In other words, you want a way to sample coverage data from contexts other than the scope enclosing the covergroup declaration.

For example, an overridden sample method can be called with different arguments to pass directly to covergroup the data to be sampled from within a task or function or from within a sequence or property of a concurrent assertion.

As we have seen, concurrent assertions have special sampling semantics, i.e., data values are sampled in the preponed region. This helps in passing the sampled data from a concurrent assertion as arguments to an overridden sample method which in turn facilitates managing various aspects of assertion coverage, such as sampling of multiple covergroups by one property, sampling of multiple properties by the same covergroup, or sampling different branches of a sequence or property (including local variables) by any arbitrary covergroup.

The syntax for using your own sample( ) method is:

```
coverage_event ::=  
clocking_event  
| with function sample ([ tf_port_list ])
```

**Table 15.3** Instance-specific coverage options

Option name	Default	Description
<b>name = string</b>	Unique name	Specifies a name for the covergroup instance. If unspecified, a unique name for each instance is automatically generated by the tool.
<b>weight = number</b>	1	If set at the covergroup syntactic level, it specifies the weight of this covergroup instance for computing the overall instance coverage of the simulation. If set at the coverpoint (or cross) syntactic level, it specifies the weight of a coverpoint (or cross) for computing the instance coverage of the enclosing covergroup. The specified weight shall be a non-negative integral value.
<b>goal = number</b>	100	Specifies the target goal for a covergroup instance or for a coverpoint or a cross of an instance.
<b>comment = string</b>	“”	A comment that appears with a covergroup instance or with a coverpoint or cross of the covergroup instance. The comment is saved in the coverage database and included in the coverage report.
<b>at_least = number</b>	1	Minimum number of hits for each bin. A bin with a hit count that is less than <i>number</i> is not considered covered.
<b>auto_bin_max = number</b>	64	Maximum number of automatically created bins when no bins are explicitly defined for a coverpoint.
<b>cross_num_print_missing = number</b>	0	Number of missing (not covered) cross product bins that shall be saved to the coverage database and printed in the coverage report.
<b>detect_overlap = boolean</b>	0	When true, a warning is issued if there is an overlap between the range list (or transition list) of two bins of a coverpoint.
<b>per_instance = boolean</b>	0	Each instance contributes to the overall coverage information for the covergroup type. When true, coverage information for this covergroup instance shall be saved in the coverage database and included in the coverage report. When false, implementations are not required to save instance-specific information.
<b>get_inst_coverage = boolean</b>	0	Only applies when the <code>merge_instances</code> type option is set. Enables the tracking of per instance coverage with the <code>get_inst_coverage</code> build-in method. When false, the value returned by <code>get_inst_coverage</code> shall equal the value returned by <code>get_coverage</code> .

Predefined coverage system tasks and functions				
\$set_coverage_db_name (<name>);		Sets the filename of the coverage database into which coverage info. is saved at the <i>end</i> of a simulation run		
\$load_coverage_db (<name>);		Load from a given filename the cumulative coverage information for all the coverage group types.		
\$get_coverage();		Returns a real number in the range of 0 to 100 the overall coverage of <i>all</i> covergroups.		
<i>Pre-defined coverage methods used in procedural code</i>				
Method	Can be called on			Description
	covergroup	coverpoint	cross	
void sample()	Yes	No	No	Triggers sampling of the covergroup
real get_coverage()	Yes	Yes	Yes	Calculates type coverage number (0...100)
real get_inst_coverage()	Yes	Yes	Yes	Calculates coverage number (0...100)
void set_inst_name(string)	Yes	No	No	Sets the instance name to the given string
void start()	Yes	Yes	Yes	Starts collecting coverage information
void stop()	Yes	Yes	Yes	Stops collecting coverage information
real query()	Yes	Yes	Yes	Returns the cumulative coverage information (for the coverage group type as a whole)
real inst_query()	Yes	Yes	Yes	Returns the per-instance coverage information for this instance.

**Fig. 15.20** Predefined tasks, functions and methods for functional coverage

Let us see a simple example of this. Note the use of *with function* semantics in the example:

```

covergroup coverSample with function sample (int X);
    coverpoint X;
endgroup : coverSample

coverSample cs = new ( );

property propertySample;
    int pX;      //local variable

    //iSig1, iSig2, iSig3 are some internal signals
    //of your design
    @(posedge clk) (iSig1, pX = iSig2) |=> (iSig3,
cs.sample(pX));

endproperty : propertySample

```

In this example, we first declare a covergroup called *coverSample* and declare a function called *sample()* using the keywords *with function sample*. This means that the function *sample* is user defined and has formal parameter “int X.” With such a declaration, you can now call the *sample()* function from within a task or function or from within a sequence or property of a concurrent assertion.

The example further defines a property named *propertySample* where we declare a local variable called “int pX.” This is the variable we want to sample in the cover-group *coverSample*. So, we call the function *sample()* from *propertySample* on some condition and pass the variable pX (actual) to the formal “X” of the cover-group *coverSample*. This way, we can cover the variable data that we want to cover at the time we want to cover. The user-defined *sample()* method can have any kind of procedural code to manipulate the data we pass to *sample()* in both combinatorial and temporal domains.

Here is an example of calling *sample()* from a SystemVerilog function:

```

function fSample;
    int fS;
    ...
    cs.sample (fS);
endfunction

```

Note that you cannot use a formal argument both as the formal of a *covergroup* and the formal argument of the *sample()* function. Here is an example:

```

covergroup X1 (int cV) with function sample (int cV);
    coverpoint cV; //Compile ERROR
endgroup

```

Compile error: As you notice, “int cV” is declared both as the formal argument of “*covergroup X1*” and the *sample* method. This will give a compiler error.

## 15.15 Querying for Coverage

There are a few ways you can query for coverage. In other words, you can “sample” for coverage, get coverage information, and then stop coverage. This helps in not taking up valuable simulation time once you have reached your coverage goals. Here is an example of how you can do that for a covergroup:

```

covergroup rg;
    pc: coverpoint pendreq;
    gc: coverpoint numMasters;
endgroup

rg my_rg = new;

real cov;

always @ (posedge req)
    my_rg.start();

always @ (posedge grant)
begin
    my_rg.sample();

//Coverage information on my_rg instance of
//'covergroup rg'
cov = my_rg.get_inst_coverage();
    if (cov > 90) my_rg.stop;
end

```

Built-in to all covergroups, coverpoints, and “cross”es is a function called *get\_coverage()* (or *get\_inst\_coverage*). This function returns a real number of the percentage coverage. In the above example, we “start” coverage for the covergroup instance “my\_rg.” After that, we “sample” the coverage of that *instance* at every posedge of grant. If the coverage is greater than 90%, we “stop” collecting

coverage. This helps tremendously with unnecessary coverage collection, thereby reducing simulation overhead.

## 15.16 Strobe ( ) Method

Note that optionally, there is also a “strobe” option that can be used to modify the sampling behavior. When the strobe option is not set (the default), a coverage point is sampled the *instant* clocking event takes place. If the clocking event occurs multiple times in a time step, the coverage point will also be sampled multiple times. The “strobe” option can be used to specify that coverage points are sampled in the postponed region, thereby filtering multiple clocking events so that only one sample per time slot is taken. The strobe option only applies to the scheduling of samples triggered by a clocking event.

## 15.17 Coverage Options: Instance-Specific Example (Fig. 15.18)

Here is a simple example on how you can *exclude* coverage of a coverpoint from total coverage using the option.weight option:

```

logic [15:0] addr, data;
covergroup cov1;
cov1_covpoint: coverpoint addr;
{
    bins zero = {0}; //1 bin for value 0
    bins low = {1:3}; //1 bin for values 1,2,3
    bins high [ ] = {4:$}; //explicit number of bins for
all remaining values
    option.weight = 5; //coverpoint 'cov1_covpoint' has a
weight of 5
}
cov2_covpoint: coverpoint data;
{
    bins all = {0:$}; //1 bin for all values
//No weight; Exclude coverage of this coverpoint towards
//total
    option.weight = 0;
}
cross cov1_copoint, cov2_covpoint;
{
    //Higher weight for this 'cross' towards total
    option.weight = 10;
}
endgroup

```

### 15.18 Coverage Options for “covergroup” Type: Example (Fig. 15.19)

### 15.19 Coverage Options: Instance-Specific Per-syntactic Level

Tables 15.1, 15.2, and 15.3 are taken from SystemVerilog LRM.

### 15.20 Coverage System Tasks, Functions, and Methods

Here is a formal description of available tasks, functions, and methods (SystemVerilog\_LRM\_1800-2012) (Fig. 15.20).

We saw usage of some of these methods in previous examples. Here is further description of some of the methods.

### **sample () method**

Explicit “sample” of a covergroup. You call this method exactly when you want to sample a covergroup. In other words, you can design conditional logic to trigger the sampling of a covergroup. This does not apply to a coverpoint or a “cross.”

### **real get\_coverage( ref int, ref int )**

The get\_coverage ( ) method returns the cumulative coverage, which considers the contribution of all instances of a particular coverage item, and it is a static method that is available on both types (via the :: operator) and instances (using the .” operator). The get\_coverage ( ) method accept both an optional set of arguments and a pair of int values passed by reference. When the optional arguments are specified, the get\_coverage( ) method assigns to the first argument the value of the covered bins and to the second argument the number of bins for the given coverage item.

### **real get\_inst\_coverage( ref int, ref int )**

get\_inst\_coverage ( ) method returns the coverage of the specific instance on which it is invoked; thus, it can only be invoked via the .” operator. The get\_inst\_coverage ( ) method accept both an optional set of arguments and a pair of int values passed by reference. When the optional arguments are specified, the get\_inst\_coverage ( ) method assigns to the first argument the value of the covered bins and to the second argument the number of bins for the given coverage item.

Here is an example:

```

covergroup cg (int xb, yb) ;
    coverpoint x {bins xbins[ ] = { [0:xb] }; }
    coverpoint y {bins ybins[ ] = { [0:yb] }; }
endgroup
cg cv1 = new (1,2); // cv1.x has 2 bins, cv1.y has 3 bins
cg cv2 = new (3,6); // cv2.x has 4 bins, cv2.y has 7 bins
initial begin
    cv1.x.get_inst_coverage(covered,total); // total = 2 :: Coverage for coverpoint instance 'x'
    cv1.get_inst_coverage(covered,total); // total = 5 :: Coverage for covergroup instance 'cv1'
    cg::x::get_coverage(covered,total); // total = 6 :: Coverage for all instances of Coverpoint 'x'
    cg::get_coverage(covered,total); // total = 16 :: Coverage for all instances of covergroup 'cg'
end

void set_inst_name ( string )

```

Sets the instance name to the given “string.” Note that you cannot set \_inst\_name of a coverpoint (obviously). This applies only to a covergroup instance name, since covergroup is the only thing you instantiate.

# Chapter 16

## SystemVerilog Processes



**Introduction** This chapter discusses SystemVerilog processes such as “initial,” “always,” “always\_ff,” “always\_latch,” “always\_comb,” and “always @\*.” Also discusses fork-join, fork-join\_any, fork-join\_none, level-sensitive time control, named event time control, etc.

In this section we will discuss procedural blocks and timing control of system-verilog processes.

### 16.1 “initial” and “always” Procedural Blocks

The “initial” procedural block starts execution at time 0 and finishes when all its executable statements complete. It is executed only once. In contrast, the “always” block executes continually through simulation and finishes only when the simulation finishes. Initial blocks do not have a trigger point. They simply start executing at time 0. “always” blocks have a trigger point that triggers the execution of the block and keep executing whenever the trigger point fires. Note that you can have an “always” block without a trigger point. But that will only result in a zero time execution of the block, and you will end up in a zero-delay infinite loop and your simulation will get stuck. So, do not do that!

SystemVerilog is a concurrent language. From that point of view, *all the “initial” and “always” blocks fire at time 0 in parallel*. There is no order of execution.

Note that you can have as many “initial” blocks as you want and you can have as many “always” blocks as you want. You need to be careful though, when you assign various variables in these blocks. If you are assigning the same variable from

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_16](https://doi.org/10.1007/978-3-030-71319-5_16)) contains supplementary material, which is available to authorized users.

different blocks (and god forbid with blocking assignments), you may end up creating a race condition since these blocks all execute in parallel and one block may clobber the assignment to the same variable in another block. We will see more on this in coming sections.

### 16.1.1 “initial” Procedural Block

As noted above, the “initial” procedure will execute only once and cease when the statements in the blocks have finished. Even though the “initial” procedure is manly sequential, you can have a “forever” loop within the “initial” procedure.

Here is an example:

```
module init;

    int memIndex;
    bit [31:0] mem[0:3]; //Memory
    parameter Memsize = 4;
    logic a, b, c;
    logic clk;

    //first initial block
    initial begin
        a = 0; b = 1; c = 0; //initialize
        #10 a = 1; b = 0; c = 1; //timestep - change values
    end

    //second initial block
    initial begin //memory initialization
        for (memIndex = 0; memIndex < Memsize; memIndex++) begin
            mem[memIndex] = $urandom();
            $display($stime,,, "mem=%d", mem[memIndex]);
        end
    end

    //third initial block
    initial begin
        clk = 0;
        forever #10 clk = !clk; //forever block - clk generator
    end

    //fourth initial block
    initial begin
        $monitor($stime,,, "clk=%d a=%d b=%d c=%d", clk, a, b, c);
        #40 $finish(2);
    end

```

```
    end
endmodule
```

*Simulation log:*

```
0 mem= 303379748
0 mem=3230228097
0 mem=2223298057
0 mem=2985317987
0 clk=0 a=0 b=1 c=0
10 clk=1 a=1 b=0 c=1
20 clk=0 a=1 b=0 c=1
30 clk=1 a=1 b=0 c=1
$finish called from file "testbench.sv", line 28.
$finish at simulation time 40
```

### V C S   S i m u l a t i o n   R e p o r t

In this example, we have four initial blocks, each one fires at time 0. This is simply to show that you can have many “initial” blocks each doing a different function. As noted above, be careful to not use the same variable in multiple “initial” blocks. You may end up writing over each other, i.e., create a race condition. In all our four blocks, we use different variables.

First “initial” block simply shows that you can initialize variables and also change them in time steps.

Second “initial” block shows the use of a “for loop.” This is very useful when you want to initialize arrays such as a memory array. Here we initialize a memory with random values. The simulation log shows the values assigned to memory.

Third “initial” block generates a clock. This is very useful to generate clocks for a design. So, yes you can have a continually executing temporal domain construct within an “initial” block.

Fourth “initial” block monitors the changes in signals of the design. It also makes the simulation finish at a given time. This is useful when you have a hard limit on finish time.

An initial block executes sequentially from top to bottom. All the initial blocks execute in parallel. They are concurrent and all start at time 0 of simulation.

## 16.1.2 “always” Procedural Block: General

“always” blocks execute always, in perpetuity. It is a repetitive behavior based on either edge sensitivity or fixed time sensitivity, i.e., some form of timing control. It cannot be zero-delay infinite repetition, which is obvious. For example:

```
always clk = !clk; //infinite zero delay loop. Simulation
will get stuck at time 0
always #10 clk = !clk; //correct time control
```

The “always” keyword is for general-purpose “always” block execution. We will see other flavors of “always” block in the coming sections.

Here is an example:

```
module alws;

    bit [31:0] dmaData;
    logic dma_intr;
    logic a, b, c, clk;

    //first always block
    always @ (posedge dma_intr) begin //edge sensitive
        dmaData = $urandom;
        $display ($stime,,, "dmaData=%h", dmaData);
    end

    //second always block
    always #10 clk = !clk; //fixed time sensitive

    //third always block
    always @ (posedge clk) begin
        a = b; //race condition
    end

    //fourth always block
    always @ (posedge clk) begin
        a = c; //race condition
        //c = a; //race condition
    end

    initial begin
        dma_intr = 0; clk = 0; b = 1; c = 0;
        forever #10 dma_intr = !dma_intr; //forever loop
    end

    initial begin
        $monitor($stime,,, "a=%d b=%d c=%d", a, b, c);
        #60 $finish(2);
    end
endmodule
```

*Simulation log:*

0 a=x b=1 c=0

```

10  dmaData=12153524
10  a=0 b=1 c=0
30  dmaData=c0895e81
50  dmaData=8484d609
$finish called from file "testbench.sv", line 30.
$finish at simulation time          60
    V C S   S i m u l a t i o n   R e p o r t

```

In this example, we have multiple “always” blocks, each of which will independently run in parallel with others. The first “always” block is sensitive to a signal edge (posedge dma\_intr). Whenever dma\_intr goes high, the block executes – regardless of what is going on in other “initial” or “always” blocks.

The second “always” block is controlled by fixed time (# 10). This means the loop will execute at # 10 interval continually until the end of the simulation.

The third and fourth “always” blocks are to showcase how a race condition is created. This is a very simple example, but if you have complex functionality with hundreds of variable assignments, you may end up making such a mistake. So, the third block makes the assignment “a = b;,” while the fourth block makes the assignment “a = c;.” Since both blocks are executing in parallel, we do not know whether a = b will execute first or a = c. So, that is a race condition. From the simulation log, we see that a = c executed last. So, a = b effect was nullified. Note also that “c = a;” will also cause a race condition because we do not know if “a = b” will execute first or “c = a.” Will the new value of “a” go to “c” or the old value?

In all the examples throughout the book, we have used ““always @(posedge <signal>)” or ““always @(negedge <signal>)” in event control of procedural blocks. But what does posedge and negedge mean?

posedge means 0->1, 0->x, 0->z, x->1, z->1

negedge means 1->0, x->0, z->0, 1->x, 1->z

Note that if you want an edge-sensitive “always” block to be sensitive to both the posedge and negedge of clock, you can do it two ways. One is to use “edge” instead of “posedge” or “negedge”:

```
always @ (edge clk) begin...end
```

Or simply:

```
always @ (clk) begin...end
```

This block will trigger both at the posedge and the negedge of “clk.”

### 16.1.3 “*always\_comb*”

“*always\_comb*” works off an implicit sensitivity list (as opposed to the general “*always*” block that works off an explicit sensitivity list). This implicit sensitivity list is derived from the RHS variables of the expressions within the “*always\_comb*” block and/or from any function called within the block (the inputs and the RHS variables within the function become part of the implicit sensitivity list).

A common coding mistake with traditional Verilog is to have an incomplete sensitivity list. This is not a syntax error and results in RTL simulation behaving differently than the post-synthesis gate-level behavior. Inadvertently leaving out one signal in the sensitivity list is an easy mistake to make in large, complex decoding logic. It is an error that will likely be caught during synthesis, but that only comes after many hours of simulation time have been invested in the project.

References to class objects and method calls of class objects do not add anything to the sensitivity list of an *always\_comb*, except for any contributions from argument expressions passed to these method calls.

Task calls are allowed in an *always\_comb*, but the contents of the tasks do not add anything to the sensitivity list.

Here is an example:

```
module alws;
    bit [3:0] a, b, d, e, g, h, z;
    byte c, f;
    bit [7:0] m;
    bit clk;

    function bit [7:0] sub (bit [3:0] x, bit [3:0] z);
        sub = x - z;
        m = g * h;
    endfunction

    always_comb begin //sensitive to RHS variables
        //always @(a or b) //same as always_comb
        c = a + b;
        //##10; //ERROR - blocking timing statement not allowed.
        //@ (z); //ERROR - Event control not allowed.
    end

    always_comb begin //sensitive to function inputs
        //also sensitive to logic within the function
        //always @(d or e or g or h) //same as always_comb
        f = sub (d, e);
    end

```

```

    always #10 clk = !clk;

    always @(posedge clk) begin
        a = $urandom; b = $urandom; d = $urandom;
        e = $urandom; g = $urandom; h = $urandom;
    end

    initial begin
        clk = 0;
        $monitor($stime,,, "a=%d b=%d c=%0d\td=%d e=%d f=%0d\t
g=%d h=%d m=%d", a, b, c, d, e, f, g, h, m);
        #60 $finish(2);
    end
endmodule

```

*Simulation log:*

0 a= 0 b= 0 c=0	d= 0 e= 0 f=0	g= 0 h= 0 m= 0
10 a= 4 b= 1 c=5	d= 9 e= 3 f=6	g=13 h=13 m=169
30 a= 5 b= 2 c=7	d= 1 e=13 f=-12	g= 6 h=13 m= 78
50 a=13 b=12 c=25	d= 9 e= 6 f=3	g= 5 h=10 m= 50

\$finish called from file "testbench.sv", line 33.

\$finish at simulation time 60

V C S S i m u l a t i o n R e p o r t

In this example, I am showing two different ways that an implicit sensitivity list is derived by “always\_comb.” Let us look at the first way. Consider the following block:

```

always_comb begin //sensitive to RHS variables
    c = a + b;
end

```

In this block, the expression  $c = a + b;$  has “a” and “b” as the RHS variables. The “always\_comb” block will see these RHS variables and make it part of its sensitivity list. So, you do not need to explicitly provide these variables for the sensitivity list. In real life, you may have many expressions with hundreds of RHS variables. That is when “always\_comb” really gets useful. But you get the point with this example.

As mentioned before, you cannot have temporal domain blocking statements in an “always\_comb” block. If you did the following, you would get a compile time error:

```
//#10; //ERROR - blocking statement not allowed.
```

Here is the error given by Synopsys – VCS:

Error-[SV-BCACF] Blocking construct in always\_comb/ff  
testbench.sv, 15

"#(10);"

Statements in an always\_comb shall not include those that block, have blocking timing or event controls, or forkjoin statements. The always\_ff procedure imposes the restriction that it contains one and only one event control and no blocking timing controls.

Try using simple always blocks.

So, the above always\_comb block is equivalent to:

```
always @(a or b) begin //explicit sensitivity list
    c = a + b;
end
```

Now let us look at the following block:

```
always_comb begin //sensitive to function inputs
    //also sensitive to logic within the function
    f = sub (d, e);
end
```

In this block, we are calling a function called “sub.” This function has two inputs “x” and “z.” Since these are inputs to the function, they will become part of the implicit sensitivity list of the “always\_comb” block. In addition, the function also has an expression ( $m = g * h$ ) which is *not* based on the inputs to the function. These variables (“g” and “h”) will also become part of the implicit sensitivity list. This is an important point and should be noted.

So, the above “always\_comb” block is equivalent to:

```
always @(d or e or g or h) begin
    f = sub (d, e);
end
```

The simulation log shows that the “always\_comb” blocks trigger every time their implicit sensitivity list changes.

To recap:

- “always\_comb” works off an implicit sensitivity list.
- This implicit sensitivity list is derived from the RHS variables of the expressions within the “always\_comb” block.

- The implicit sensitivity list is also derived from any function called within the block (the inputs and the RHS variables within the function become part of the implicit sensitivity list – meaning it is sensitive to changes within the function).
- The “always\_comb” block automatically executes at time zero.
- Variables on the left-hand side of assignments within an “always\_comb” procedure, including variables from the contents of a called function, *will not be written to by any other processes*.
- Statements in an “always\_comb” will not include those that block, have blocking timing or event controls, or fork-join statements. In other words, the logic must be combinational.

### **16.1.4 always @ (\*)**

always @ (\*) is similar to “always\_comb” but with subtle differences. The implicit event\_expression, @\*, is a convenient shorthand that eliminates the problems of incomplete sensitivity list by adding all nets and variables that are read by the statement of a procedural\_timing\_control\_statement to the event\_expression.

All net and variable identifiers that appear in the statement will be automatically added to the event expression, except for identifiers that appear only in the “wait” or event expressions. Nets and variables that appear on the right-hand side of assignments, in subroutine calls, in case and conditional expressions, as an index variable on the left-hand side of assignments, or as variables in case item expressions will all be included by these rules.

Some examples (SystemVerilog\_LRM\_1800-2012).

#### **Example 1:**

```
always @ ( * )
z = a && b || d & e || myfunct(f);
```

Is equivalent to:

```
always @ (a or b or d or e or f)
z = a && b || d & e || myfunct(f);
```

#### **Example 2:**

```
//following equivalent to @(a or b or c or d or tmp1 or tmp2)
always @* begin
    tmp1 = a & b;
    tmp2 = c & d;
    y = tmp1 | tmp2;
end
```

**Example 3:**

```
always @* begin // equivalent to @(b)
    @(i) kid = b; // i is not added to @*
end
```

**Example 4:**

```
always @* begin // equivalent to @(a or b or c or d)
    x = a ^ b;
    @* // equivalent to @(c or d)
    x = c ^ d;
end
```

So, what is the difference between always\_comb and always @ (\*)?

- always\_comb automatically executes once at time 0, while @ (\*) waits until a change occurs on a signal in its sensitivity list.
- always\_comb is sensitive to changes within the contents of a function, whereas always @ (\*) is only sensitive to changes to the arguments of a function.
- Variables on the LHS of assignments within an always\_comb block, including variables from the contents of a called function, will not be written to by any other process. Always @ (\*) permits multiple processes to write to the same variable.
- Blocking timing control is not allowed in always\_comb but is allowed in always @ (\*).

**16.1.5 “always\_latch”**

“always\_latch” procedure allows for modeling latched behavior logic. This construct is identical to the “always\_comb” construct, except that the simulator performs additional checks and warns if the behavior in the “always\_latch” block does not represent latched logic (as opposed to “always\_comb” that will complain if the logic within the block does not represent combinational logic). Here is a simple example:

```
module alws;
    logic clk;
    logic q, d, a, c;

    always_latch begin
        q <= d;
```

```
//#10; //ERROR
end

always #10 clk = !clk;
always @(posedge clk) d <= ~d;

initial begin
    clk = 0; d = 1;
    $monitor($stime, "clk = %d d = %d q = %d",clk,d,q);
    #60 $finish(2);
end
endmodule
```

*Simulation log:*

```
0  clk = 0 d = 1 q = 1
10  clk = 1 d = 0 q = 0
20  clk = 0 d = 0 q = 0
30  clk = 1 d = 1 q = 1
40  clk = 0 d = 1 q = 1
50  clk = 1 d = 0 q = 0
```

\$finish called from file "testbench.sv", line 17.

\$finish at simulation time 60

V C S   S i m u l a t i o n   R e p o r t

The “always\_latch” models the latch behavior ( $q \leq d$ ), and since “ $d$ ” is on the RHS of the expression, it becomes the sensitivity list of the “always\_latch” procedure. The simulation log shows the transparent latch behavior.

And just as with “always\_comb” if you place a timing blocking statement, you will get a compile error.

Also, just as in always\_comb, variables on the left-hand side of assignments within an “always\_latch” procedure, including variables from the contents of a called function, *will not be written to by any other processes*. For example, following is not allowed:

```
always_latch begin
    latch1 <= a;
end

//following will give an Error since latch1 is driven
//from two processes.
always @*
    latch1 = b;
```

Here is the error you’ll get (Mentor – Questa):

\*\* Error (suppressible): testbench.sv(46): (vlog-7033) Variable 'latch1' driven in a combinational block, may not be driven by any other process.

### 16.1.6 “always\_ff”

As you might have guessed, “always\_ff” is for flip-flop behavior. It is useful for modeling synthesizable sequential logic behavior. As opposed to “always\_latch,” the “always\_ff” must contain an event control – and one and only one event control – else you will get a compile error. It cannot contain blocking timing controls. Variables on the left-hand side of assignments within an always\_ff procedure, including variables from the contents of a called function, will not be written to by any other process.

Here is an example showing which constructs would cause a compile error:

```
module alws;
  logic clk;
  logic q, d, reset;

  //always_ff begin //COMPILE ERROR
  //always_ff @ (posedge clk or posedge reset) //OK
  always_ff @ (posedge clk) begin
    //@ (posedge reset); //COMPILE ERROR
    //#1 q <= d; //COMPILE ERROR
    q <= d;
  end

  always #10 clk = !clk;
  always @ (posedge clk) d <= ~d;

  initial begin
    clk = 0; d = 1;
    $monitor($stime,,,"clk = %d d = %d q = %d", clk, d, q);
    #60 $finish(2);
  end
endmodule
```

*Simulation log:*

```
0  clk = 0 d = 1 q = x
10 clk = 1 d = 0 q = 1
20 clk = 0 d = 0 q = 1
30 clk = 1 d = 1 q = 0
40 clk = 0 d = 1 q = 0
```

```

50  clk = 1 d = 0 q = 1
$finish called from file "testbench.sv", line 17.
$finish at simulation time          60
V C S   S i m u l a t i o n   R e p o r t

```

The “always\_ff” procedure models the flip-flop behavior, and its results are shown in the simulation log.

The following will cause a compile error:

```

always_ff begin //COMPILE ERROR
    q <= d;
end

```

That is because there is no event control specified with the “always\_ff” construct. The error will be as follows (Synopsys – VCS):

Error-[ECNIAF] Event control needed in always\_ff  
testbench.sv, 5

There must always be exactly one event control in always\_ff blocks.

Similarly, the following block will also give a compile error:

```

always_ff @ (posedge clk) begin
    @ (posedge reset); //COMPILE ERROR
    q <= d;
end

```

Here the error emanates from the fact that there is a blocking timing control within the block. The error will be as follows (Synopsys – VCS):

Error-[OOECAIAB] Too many event controls  
testbench.sv, 7

Only one event control is allowed in an always\_ff block.

The following will also give a compile error and cannot have any type of blocking timing control:

```
#1 q <= d; //COMPILE ERROR
```

Here is the compile error you’ll get (Synopsys – VCS):

Error-[SV-BCACF] Blocking construct in always\_comb/ff  
testbench.sv, 8  
"#(1) q <= d;"

Statements in an always\_comb shall not include those that block, have blocking timing or event controls, or forkjoin statements. The always\_ff procedure imposes the restriction that it contains one and only one event control and no blocking timing controls.

Notice the simulation log file, it shows true flip-flop behavior.

## 16.2 “final” procedure

The “final” procedure is like an “initial” procedure, defining a procedural block of statements, except that it occurs at the *end of simulation* time and executes without delays. A final procedure is typically used to display statistical information about the simulation.

The only statements allowed inside a “final” procedure are those that can execute within a single simulation cycle. It executes in zero time (as opposed to an “initial” block that can have blocking timing controls).

You can have multiple “final” blocks and they will execute in any order. But the simulator will maintain a deterministic order every time the simulation is run. This is to allow consistent results in simulation log.

No remaining scheduled events execute after all “final” procedures have executed. Here is an example:

```
module alws;
  logic clk;
  int cycleCount, sum;

  function void sumIt ( );
    sum = cycleCount + 100;
  endfunction

  always #10 clk = !clk;

  always @(posedge clk)
    cycleCount = cycleCount + 1;

  initial begin
    clk = 0;
    #600 $finish(2);
  end

  final begin
    $display("Final - 1: cycleCount = %0d", cycleCount);
  end

  final begin
    sumIt ( );
    $display("Final - 2: sum = %0d", sum);
  end
endmodule
```

```

    end
endmodule

```

*Simulation log:*

```

Final - 1: cycleCount = 30
Final - 2: sum = 130
$finish at simulation time          600
V C S   S i m u l a t i o n   R e p o r t

```

This simple example shows that the “final” block is executed *after* the \$finish task is invoked. The \$finish executes at time 600, and that is when the “final” procedure is executed. There are two “final” procedures, and their execution order is random. But the simulator will (should) maintain a deterministic order from one regression to another. The first “final” block has a simple \$display, while the second “final” block calls a function which executes in zero time and then the \$display takes place. The simulation log shows the output from the two “final” procedures.

## 16.3 Parallel Blocks: fork-join

We have seen and will continue to see the sequential blocks, i.e., begin-end. So, I will not go into its detail. Instead focus on the parallel/concurrent block, namely, fork-join blocks.

As the name suggests, the statements between fork-join (and its variations) statements execute concurrently. Control passes out of the block when the last time-ordered statement executes based on the type of “join” keyword.

There are three variations of fork-join:

```

fork-join
fork-join_any
fork-join_none

```

Let us discuss each one.

### 16.3.1 *fork-join*

In such a block, each statement executes as a concurrent process. The timing control does not have to be ordered sequentially in time. In this block, the parent process *blocks* until all the processes spawned by the “fork” competes. Here is an example:

```

module forkJoin;
    int a, b, c, d;

    initial
        fork : forkBlock
            begin //first process
                #50 a = 5;
                $display($stime,,, "a = %0d",a);
            end

            begin //second process
                #100 b = 10;
                $display($stime,,, "b = %0d",b);
            end

            begin //third process
                #100 c = 20;
                $display($stime,,, "c = %0d",c);
            end

            begin //fourth process
                #50 d = 15;
                $display($stime,,, "d = %0d",d);
            end

        //first, second, third and fourth processes execute in
        parallel.

        join

endmodule

```

*Simulation log:*

```

50  a=5
50  d=15
100 b=10
100 c=20
V C S  S i m u l a t i o n  R e p o r t

```

There are four processes running in parallel in this example. Each begin-end is an independent process with no relation to the other begin-end processes. Note that you do not have to have a begin-end around each statement. I am making the point that you can indeed have begin-end processes within fork-join. The begin-end block

causes the entire block to execute as a single process. Within the begin-end block, the statements will execute sequentially (as with any begin-end block).

The time within each process (begin-end) is independent of time in other processes. In other words, time is not incremented from one begin-end process to another. The first process executes at time 50, and the second process executes at time 100. This does not mean that the second process will execute at time 150. The first process will execute at time 50 and the second process will execute at time 100. The time-delay control in each process is independent of time-delay control in other processes.

Hence, in simulation log, you see that the first and fourth processes execute concurrently since they are both scheduled to execute at time 50. Similarly, the second and third processes execute in parallel since they are both scheduled at time 100. And all the four processes execute in parallel. This is evident from the simulation log.

The following two blocks will produce identical results, further proving that fork-join processes execute in parallel:

```
fork
#10 a = 1;
#5 b=1;
#20 c = 1;
#40 d = 1;
join
```

Will produce identical results as with the following block where we have reversed the order of execution of statements:

```
fork
#40 d = 1;
#20 c = 1;
#5 b = 1;
#10 a = 1;
join
```

Note that a “return” statement within the context of a fork-join block is illegal and will result in a compile error.

### 16.3.2 *fork-join\_any*

fork-join\_any works such that when *any* of the parallel process finishes, you will jump out of the fork-join\_any block. Other concurrent processes will continue to execute. In other words, the parent process blocks only until any of the processes spawned by the fork completes.

Here is an example similar to the one discussed above:

```
module forkJoin;
    int a, b, c, d;

    initial
        begin

            fork : forkBlock
                begin //first process
                    #50 a = 5;
                    $display($stime,,, "a = %0d",a);
                end

                begin //second process
                    #100 b = 10;
                    $display($stime,,, "b = %0d",b);
                end

                begin //third process
                    #100 c = 20;
                    $display($stime,,, "c = %0d",c);
                end

                begin //fourth process
                    #40 d = 15;
                    $display($stime,,, "d = %0d",d);
                end
            join_any
                $display($stime,,, "out of the loop");
        end
    endmodule
```

*Simulation log:*

```
40  d = 15
40  out of the loop
50  a = 5
100 b = 10
100 c = 20
V C S  Simulation  Report
```

Again, four processes are forked off. But this time, we use fork-join\_any structure. The fourth process is the first one to complete at time 40. And as soon as one of the processes (fourth in our case) is complete, the control jumps out of the fork

block. We have a display statement outside of the fork block which executes as soon as the fourth process (which is the earliest of all four processes) completes. This is evident from the simulation log (you jump out of the fork block at time 40 which is when the first of the four processes completes). Note that other concurrent processes will continue to execute to their completion.

### 16.3.3 *fork-join\_none*

fork-join\_none is interesting in that the control jumps out of the fork block even when none of the processes have completed. Useful in cases, when you want to fork off multiple parallel processes but do not want to wait for any of them to complete.

The following example is identical to the one discussed above except that I have replaced join\_any with join\_none:

```
module forkJoin;
    int a, b, c, d;

    initial
        begin

            fork : forkBlock
                begin //first process
                    #50 a = 5;
                    $display($stime,,, "a = %0d",a);
                end

                begin //second process
                    #100 b = 10;
                    $display($stime,,, "b = %0d",b);
                end

                begin //third process
                    #100 c = 20;
                    $display($stime,,, "c = %0d",c);
                end

                begin //fourth process
                    #40 d = 15;
                    $display($stime,,, "d = %0d",d);
                end
            join_none

            $display($stime,,, "out of the loop");
```

```
    end
endmodule
```

*Simulation log:*

```
0  out of the loop
40  d = 15
50  a = 5
100 b = 10
100 c = 20
```

V C S   S i m u l a t i o n   R e p o r t

Time: 100 ns

Again, four processes are forked off in parallel. Since there is join\_none construct used with the fork block, the control will jump out of the fork block as soon as the fork block starts executing. You do not wait for any of the processes to complete. That is why the display statement “out of the loop” executes at time 0, which is when the fork block starts executing. You jump out at time 0.

## 16.4 Level-Sensitive Time Control

So far, we have seen edge-sensitive time control, as in @(posedge clk) or @(negedge event). There is also level-sensitive time control denoted by the keyword “wait.” This time control does not wait for an edge but simply waits for a condition to “occur.” For example, “wait a == 1.” The “wait” statement will evaluate a condition, and if it is not true, it will block the procedural statements that follow that “wait” statement.

Here is an example:

```
module level;
  int a, b, c, d;

  initial
    begin

      # 10; b = c * d;

      wait (a == 1);
      $display($stime,,, "wait complete");

    end
```

```

initial begin
    c = 5; d = 10; a = 1;
end
endmodule

```

*Simulation log:*

```
wait complete
```

### V C S   S i m u l a t i o n   R e p o r t

In this example, we use the “wait” statement to level-sensitive wait for a condition to occur (“`wait ( a == 1 ) ; .`.” This means that as soon as `a == 1`, the control will move on to the next statement in the block. We are not waiting for an edge to occur on “`a`.” So, after time 10, the “`wait`” looks to see if “`a == 1`,” and since it does find `a` to be equal to 1, it passes the control to the next statement which is a display statement in our case. The simulation log shows this \$display coming out at time 10.

What if you changed “`wait (a == 1)`” to “`@(posedge a)`” in the above example? Try it out. Hint: you will not get the \$display statement to execute.

Here’s another example. Study it carefully to see how you can mix “`wait`” and “`@`” controls within a block:

```

module level;
int a, b, c, d;

initial begin
    wait (a == 1);
    $display($stime,,, "wait a == 1 Complete");

    @(posedge b);
    $display($stime,,, "@(posedge b) Complete");

    wait (c == 1)
    $display($stime,,, "wait c == 1 Complete");
end

initial begin
    b=0; c = 0; a = 1;
    #10; b = 1;
    #10; c = 1;
end
endmodule

```

*Simulation log:*

```
# run -all
```

```
#      0  wait a == 1 Complete
#      10 @ (posedge b) Complete
#      20  wait c == 1 Complete
# exit
```

### 16.4.1 Wait Fork

This is an interesting “wait.” It allows a program block to wait for the completion of all its concurrent threads before existing. It blocks process execution flow until all immediate child subprocesses have completed. Note that it blocks until processes created by current processes, excluding their descendants, are completed. Best explained via an example derived from (SystemVerilog – LRM):

```
module top;

initial begin
    fork
        child1( );
        child2( );
    join_none

    do_more( );
end

task do_more( );
    fork
        child3( );
        child4( );
    fork : descend
        descendant1( );
        descendant2( );
    join_none
    join_none

    do_further();

wait fork; //block until child1 ... child6 complete
                //will not wait for descendant1 and descendant2
$display($stime,,, "wait fork DONE");

endtask
```

```
function void do_further();
    fork
        child5( );
        child6( );
    join_none
endfunction

task child1;
    #1; $display($stime,,, "In child1");
endtask

task child2;
    #1; $display($stime,,, "In child2");
endtask

task child3;
    #2; $display($stime,,, "In child3");
endtask

task child4;
    #2; $display($stime,,, "In child4");
endtask

task child5;
    #3; $display($stime,,, "In child5");
endtask

task child6;
    #3; $display($stime,,, "In child6");
endtask

task descendant1;
    #3; $display($stime,,, "In descendant1");
endtask

task descendant2;
    #3; $display($stime,,, "In descendant2");
endtask
endmodule
```

*Simulation log:*

```
# run -all
#      1  In child1
#      1  In child2
```

```

#      2 In child3
#      2 In child4
#      3 In child5
#      3 In child6
#      3 wait fork DONE
#      3 In descendant1
#      3 In descendant2
# exit

```

We fork off many concurrent processes in the example. First, immediate child processes child1 and child2 are forked off after which we call the task “do\_more.” This task forks off two more immediate child processes, child3 and child4, and also creates a sub-fork which forks off “descendant1” and “descendant2” processes. Note that “descendant1” and “descendant2” are not immediate child processes. Two more immediate child processes child5 and child6 are forked off by the function “do\_further.” After calling do\_further, we wait for all the immediate concurrent processes to complete using “wait fork.”

“wait fork” completes (stops blocking) after all the immediate concurrent processes are completed but does not wait for the descendant processes. The simulation log shows how the processes are executed and when “wait fork” completes.

## 16.5 Named Event Time Control

Named events are also discussed in Sect. 2.12.

We have seen edge-sensitive and level-sensitive time control. But what if you want to have explicit time control? You want to create an explicit edge that then triggers edge-based timing control. This is done via a so-called named event. It is declared by the keyword *event* and triggered by the construct “->.” Here is an example:

```

module level;
    bit dma_interrupt;
    event process_dma; //named event

    initial begin
        dma_interrupt = 0;
        #10;
        dma_interrupt = 1;
    end

    always @(posedge dma_interrupt)
    begin
        //do something

```

```

#10;
-> process_dma; //trigger named event
$display($stime,,, "trigger process_dma event");
end

always @process_dma begin //detect event edge
$display($stime,,, "process_dma event edge detected");
end
endmodule

```

*Simulation log:*

```

20 trigger process_dma event
20 process_dma event edge detected
VCS Simulation Report

```

In this example, we declare a named event called “process\_dma.” On posedge of `dma_interrupt`, we want to process the interrupt. So, we fire/trigger the named event “process\_dma” using the `->` syntax (`-> process_dma`). This triggers the block that is waiting for an edge on the event “process\_dma.” So, we explicitly fire the event `process_dma` and pass control to the block waiting for that event to trigger. This allows for very good timing control over executing concurrent processes. This is useful when you want explicit control over triggering various processes. Note that once you fire the event, the control passes to the very next statement. The simulation log shows when the `process_dma` event was fired and when it was detected.

### 16.5.1 Merging of Named Events

When one event variable is assigned to another, the two become merged. Thus, executing `->` on either event variable affects processes waiting on either event variable. Here is an example:

```

module mevent;

event a, b, c;

initial begin
a = b;
-> c;
#1; -> a; // also triggers b
#1; -> b; // also triggers a
a = c;

```

```

b = a;
#1; -> a; // also triggers b and c
#1; -> b; // also triggers a and c
end

always @ (a)
$display($stime,,, "event a triggered");

always @ (b)
$display($stime,,, "event b triggered");

always @ (c)
$display($stime,,, "event c triggered");
endmodule

```

*Simulation log:*

```

0 event c triggered
1 event a triggered
1 event b triggered
2 event a triggered
2 event b triggered
3 event c triggered
3 event a triggered
3 event b triggered
4 event c triggered
4 event a triggered
4 event b triggered

```

In this example, we are equating events with each other. First we assign event  $a = \text{event } b$  ( $a = b;$ ). This means that when we trigger “a,” we also trigger “b.” Hence, when we trigger “a,” we get the following in the simulation log:

```

1 event a triggered
1 event b triggered

```

Similarly, when we trigger “b,” both “a” and “b” will be triggered as shown in the simulation log:

```

2 event a triggered
2 event b triggered

```

Then, we assign “ $a = c$ ” which means when we trigger “a” both “a” and “c” will be triggered. Then we also assign “ $b = a;$ ” So, in this case when “b” is triggered, both “a” and “b” will be triggered. So, the net effect is when “a” is triggered, all three “a,” “c,” and “b” will be triggered. This is evident from the simulation log:

```

3 event c triggered

```

- 3 event a triggered
- 3 event b triggered

Same logic applies when we trigger “b.” All three “a,” “b,” and “c” will be triggered:

- 4 event c triggered
- 4 event a triggered
- 4 event b triggered

### **16.5.2 Event Comparison**

Event variables can be compared against other event variables (or the “null” value). The following operators are allowed:

- Equality ( == )
- Inequality ( != )
- Case equality ( === )
- Case inequality ( !== )

For example:

```
module mevent;
  event a, b;

  initial begin
    a = b;
    if ( a != null) begin
      if ( a == b)
        $display($stime,,,"event a and b are equal");
    end
  end
endmodule
```

*Simulation log:*

```
0 event a and b are equal
V C S S i m u l a t i o n R e p o r t
```

## **16.6 Conditional Event Control**

You can have a conditional control over the “@” edge-sensitive event control. For example:

```

module myMod (output logic y, input a, input enable);
...
always @(a iff enable == 1)
y <= a;
...
endmodule

```

In this example, the event control “`always @(a)`” triggers, if and only if, `enable` is true. Note that the “`iff`” condition is evaluated when “`a`” changes and not when “`enable`” changes.

Another example:

```

always_ff @ (posedge clock iff reset == 0)
busGnt = busReq && busAck;

```

## 16.7 Disable Statement

The disable statement allows you to have explicit control over when to disable (stop from executing) a process. The disable statement gives a mechanism for terminating a task before it executes all its statements, breaking from a looping statement, or skipping statements in order to continue with another iteration of a looping statement. For example, this is useful when you want to have explicit control over processing a non-maskable machine exception. You want to disable the executing process as soon as you detect such an exception and pass control over to another process (the process that services the exception).

The disable statement terminates the activity of a task or a named block. Execution resumes at the statement following the disabled block. All activities enabled within the named block or task will be terminated as well. If task enable statements are nested (i.e., one task enables another, and that one enables yet another), then disabling a task within the chain shall disable all tasks downward on the chain. If a task is enabled more than once, then disabling such a task will disable all activations of the task.

Here is an example:

```

module dsabl;
bit a, b;

initial begin : init_block
a = 0;
#10;
a = 1;
disable init_block;
$display($stime,,,"after disable init_block");

```

```

        //will never execute
    end

    always @(posedge a)
        begin : outer_block

            begin : inner_block
                #20 b = 1;
                if (a == 1) disable inner_block;
            end

            $display($stime,,, "From outer_block");
        end
    endmodule

```

*Simulation log:*

```

20  From outer_block
V C S   S i m u l a t i o n   R e p o r t

```

In this example, we are showing two different usages of “disable” block. The first one is in the “initial” block. The first thing to note here is that we must name a block (“init,” “always,” etc.) so that we can disable it with that name. The “initial” block in our case is named “init\_block.” Inside the “initial” block, we perform some logic on signal “a” and then disable the block “init\_block.” This will stop execution of the block and quit the block. Hence, the display statement after the disable statement will never execute.

The second “disable” statement is in the “always” block. There are two blocks within the “always” block: one the “outer\_block” and another the “inner\_block.” The point to note here is that when we disable the “inner\_block,” the control passes to the “outer\_block.” In other words, control passes to any statement or process that is executing right after the block that is being disabled. In our case, the block that is executing after we disable the “inner\_block” is the “outer\_block,” and hence the \$display from “outer\_block” is executed.

Here is an example of how “disable” affects an executing loop:

```

...
begin : top_block
    for (I = 0; I <= n; I=I+1) begin : inner_block
        @ (event)
            if (data <= 'hff) disable inner_block;
        ...
        ...
        @ (event)
            if (data == 'haa) disable top_block;
        ...

```

```
....  
    end  
end
```

When “disable inner\_block” is executed, the execution of “inner\_block” terminates, and the execution passes to the next iteration of the for loop. When “disable top\_block” executes the entire top, block is terminated which includes the for loop.

# Chapter 17

## Procedural Programming Statements



**Introduction** These chapter discusses procedural programming statements, such as if-else-if, case/casex/casez, unique-if, priority-if, unique-case, priority-case, loop statements, jump statements, etc.

### 17.1 if-else-if Statements

We are quite familiar with if-else-if constructs which we use often in our code. SystemVerilog also supports this construct as in many other languages. The “if” statement checks to see if a condition is true or false (meaning the condition is evaluated for a Boolean outcome). If the result of the expression is “0” or “x,” the expression condition is said to fail. For example:

```
reg f, g, a, b, s1, s2;
always @ (a or b or s1 or s2)
begin
  if (s1 == 1)
    begin
      f = a;
      if (s2 == 1)
        g = a | b;
      else g = a & b;
    end
  end
```

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_17](https://doi.org/10.1007/978-3-030-71319-5_17)) contains supplementary material, which is available to authorized users.

```
    else
        f = b;
    end
```

These are nested ifs. Note the placement of begin-end to correctly associate if and else branches.

There is also the if-else-if construct that allows chaining of nested “if” statements. They allow for a flexible way of writing multiway decision. If any of the expression is true, the statement associated with it is executed, and the whole chain is then terminated. For example:

```
module unq;

bit [7:0] intr, vector, vector1;
bit [15:0] intrVector, index;

initial begin

#10;
if (intr < vector) begin
    intrVector = index + vector;
    index = index - 1;
    $display($stime,,, "intr < vector");
end

else if (intr > vector1) begin
    intrVector = index;
    index = index + 1;
    $display($stime,,, "intr > vector1");
end

else if (intr == vector) begin
    intrVector = index -- ;
    $display($stime,,, "intr == vector");
end

else
    intrVector = intrVector + 1;

end

initial begin
    intr = 12; vector=123; vector1=2;
```

```
    end
endmodule
```

*Simulation log:*

```
10  intr < vector
VCS Simulation Report
```

Note that in this example, both conditions, namely, (intr < vector) and (intr > vector1), are true. But as soon as the first condition (intr < vector) is satisfied, the statement associated with it is executed, and the entire if-else-if chain is considered to be over.

The final “else” is the default statement. If the entire chain does not satisfy its conditions, the final “else” will be executed as the default catch all statement.

### 17.1.1 *unique-if* and *unique0-if*

SystemVerilog adds two new keywords called “unique-if” and “unique0-if.” They are useful not only for functional modeling but also for logic synthesis and formal verification. These are “if” statement modifiers. These keywords are to be used before an “if” to perform certain violation checks.

The unique keyword tells all software tools that support SystemVerilog, including those for simulation, synthesis, lint-checking, and formal verification, that each selection item in a series of decisions is unique from any other selection item in that series and that *all legal cases have been listed*.

“unique-if” and “unique0-if” specify (assert) that *there is no overlap in a series of if-else-if conditions*. In other words, the *conditions are mutually exclusive* (unique). This allows the conditions to be evaluated in parallel, since there is no overlap. This also means that the conditions may be evaluated and compared in any order. So, a “unique-if” and “unique0-if” is violated if more than one condition is found to be true. In that case, the simulator will issue a warning.

Also, if there is no match on any of the “if-else-if” conditions, and no default “else” clause, you will get a violation warning.

Let us look at an example. This example is identical to the one above, except that I have replaced “if” with “unique-if”:

```
module unq;
  bit [7:0] intr, vector, vector1;
  bit [15:0] intrVector, index;

  initial begin
#10;

  unique if (intr < vector) begin
```

```

intrVector = index + vector;
index = index - 1;
$display($stime,,, "intr < vector");
end

else if (intr > vector1) begin
    intrVector = index;
    index = index + 1;
    $display($stime,,, "intr > vector1");
end

end

initial begin
    intr = 12; vector=123; vector1=2;
end
endmodule

```

In this example, both conditions (intr < vector) and (intr > vector1) are satisfied. This violates the rule for unique-if. Unique-if requires that all conditions are unique and that they do not overlap. So, you will get the following warning (Synopsys – VCS):

Warning-[RT-MTOCMUIF] More than one condition match in statement  
testbench.sv, 10

More than one condition matches are found in 'unique if' statement inside  
unq, at time 10ns.

Line number 10 and 17 are overlapping.

Now, I take the same example but change the initial values of “intr,” “vector,” and “vector1” as shown here:

```

initial begin
    intr = 12; vector=1; vector1=123;
end

```

These values are such that none of the if-else-if conditions will be satisfied (and there is no default “else” clause). In this case, the “unique-if” will issue a warning as follows (Synopsys – VCS):

Warning-[RT-NCMUIF] No condition matches in statement  
testbench.sv, 10

No condition matches in 'unique if' statement. 'else' statement is missing

for the last 'else if' block, inside unq, at time 10ns.

### V C S   S i m u l a t i o n   R e p o r t

If we had not used “unique-if” and used only “if” in the above example, we will not get such a warning.

Now, let us look at “unique0-if.” “unique0-if” is identical to “unique-if,” except that if no conditions match, there will *not* be a violation. In the following example, none of the conditions match and still there will not be a violation warning. If we had used “unique-if” instead, we would get a warning as we saw in the previous example:

```
module unq;
    bit [7:0] intr, vector, vector1;
    bit [15:0] intrVector, index;

    initial begin

        #10;

        unique0 if (intr < vector) begin

            intrVector = index + vector;
            index = index - 1;
            $display($stime,,, "intr < vector");
        end

        else if (intr > vector1) begin
            intrVector = index;
            index = index + 1;
            $display($stime,,, "intr > vector1");
        end
    end

    initial begin
        intr = 12; vector=1; vector1=123;
    end
endmodule
```

If you carefully examine the “initial” values of all the variables, none of the if-else-if conditions match. But since we used “unique0-if,” no violation will be reported.

### 17.1.2 priority-if

Just as in “unique” and “unique0,” a “priority” keyword can be used before an “if.” It also performs violation checks. It is identical in all respects to its “unique” counterpart, except that a series of if-else-if conditions will be evaluated *in the order in which these conditions are listed*. With “unique” or “unique0,” the statements are evaluated in parallel, in any order. Just as with “unique” cases, the “priority” keyword instructs all tools that support SystemVerilog that each selection item in a series of decisions must be evaluated in the order in which they are listed and that all legal cases have been listed. The syntax is:

```
priority if
```

In the preceding example (on unique0), if we change the values of intr, vector, and vector1 such that both branches of if-else-if are satisfied, then a “priority-if” logic would be required.

Note that the unique, unique0, and priority keywords apply to the entire series of if-else-if conditions. In other words, in any of the preceding examples, it would have been illegal to insert any of these keywords after any of the occurrences of “else.” The following example is identical to the one on “unique-if,” except that the “unique” keyword is replaced with “priority” keyword:

```
module unq;
    bit [7:0] intr, vector, vector1;
    bit [15:0] intrVector, index;

    initial begin

        #10;

        priority if (intr < vector) begin

            intrVector = index + vector;
            index = index - 1;
            $display($stime,,, "intr < vector");
        end

        else if (intr > vector1) begin
            intrVector = index;
            index = index + 1;
            $display($stime,,, "intr > vector1");
        end
    end
endmodule
```

```

    end

    else if (intr == vector) begin
        intrVector = index -- ;
        $display($stime,,, "intr == vector");
    end

    else
        intrVector = intrVector + 1;

    end

initial begin
    intr = 123; vector=1234; vector1=12;
end
endmodule

```

*Simulation log:*

```
# run -all
#      10  intr < vector
# exit
```

Even though both “intr < vector” and “intr > vector1” conditions match, the “intr < vector” branch is executed because of priority-if. No warnings.

Note: An excellent paper titled ‘SystemVerilog’s priority & unique - A Solution to Verilog’s “full\_case” & “parallel\_case” Evil Twins!’ written by Cliff Cummings (Cummings, [www.sunburst-design.com](http://www.sunburst-design.com)) is available on [www.sunburst-design.com](http://www.sunburst-design.com). If you are interested in evaluating “unique” and “priority” cases for logic synthesis, this paper is a must read.

## 17.2 Case Statements

We have all used a “case” statement in many different languages. It is basically a multiway decision statement that tests whether an expression matches one of several branches according to the selecting expression. Here is an example:

```

module cas;
    logic [2:0] index;

    always @(index) begin
        case (index)
            0 : $display($stime,,, "Case 0 is executed");

```

```

1 : $display($stime,,, "Case 1 is executed");
2 : $display($stime,,, "Case 2 is executed");
3 : $display($stime,,, "Case 3 is executed");
      default : $display($stime,,, "Default Case is
executed");
endcase
end

initial begin
#5 index = 0;
$monitor($stime,,, "index = %0d",index);
#40 $finish(2);
end

always begin
#10; index = index++;
end
endmodule

```

*Simulation log:*

```

5 Case 0 is executed
5 index = 0
10 Case 1 is executed
10 index = 1
20 Case 2 is executed
20 index = 2
30 Case 3 is executed
30 index = 3
40 Default Case is executed
40 index = 4

```

\$finish called from file "testbench.sv", line 19.

\$finish at simulation time 45

V C S   S i m u l a t i o n   R e p o r t

The “case” statement in the example is based on the selection of “index.” “index” is called a *case\_expression*, and a branch selection item (in our case 0, 1, 2, 3 default) is called *case\_item\_expression (or case\_item)*. As “index” changes, it selects a branch of the “case” statement. Simulation log shows how each branch is selected based on the value of “index.” Note the use of “default” case. “default” statement is optional.

The *case\_item\_expressions* will be evaluated and then compared in the exact order in which they appear. During this linear search, if one of the *case\_item\_expressions* matches the *case\_expression*, then the statement associated with that *case\_item*

is executed, and the linear search will terminate. If all comparisons fail and the default item is given, then the default item statement is executed. If the default statement is not given and all of the comparisons fail, then none of the case\_item statements will be executed.

Whenever “index” value is *not* one of 0, 1, 2, or 3, the default case statement will be executed. Note that the case\_expressions and the case\_item\_expressions do not necessarily have to be constant expressions.

In a case\_expression comparison, the comparison succeeds only when each bit matches exactly with respect to values 0, 1, x, and z. From that, it is derived that the bit length of all the expressions needs to be equal, so that exact bitwise comparison can be performed. Therefore, the length of all the case\_item\_expressions, as well as the case\_expression, will be made equal to the length of the longest case\_expression and case\_item\_expressions. If any of these expressions is unsigned, then all of them will be treated as unsigned. If all of these expressions are signed, then they will be treated as signed.

Note that you can have multiple case\_items in a case statement. In the above example, you could do the following:

```
always @(index) begin
    case (index)
        //multiple case_items
        0,1 : $display($stime,,, "Case 0 or 1 is executed");

        2 : $display($stime,,, "Case 2 is executed");
        3 : $display($stime,,, "Case 3 is executed");
        default : $display($stime,,, "Default Case is
executed");
    endcase
end
```

And the simulation log will be as follows:

```
5  Case 0 or 1 is executed
5  index = 0
10 Case 0 or 1 is executed
10 index = 1
20 Case 2 is executed
20 index = 2
30 Case 3 is executed
30 index = 3
40 Default Case is executed
40 index = 4
```

### 17.2.1 *casex, casez, and do not care*

In addition to normal “case,” the language adds two other types of case statements. These are provided to handle “do not care” conditions during case comparison. One of these treats high impedance (“z”) as a “do not care.” That is “casez.” The other treats both “z” and “x” as “do not cares.” That is “casex.” Both are used the same way as a normal case, but they begin with keywords “casez” or “casex.” “Do not care” values “z” or “x” in any bit of either the case\_expression or the case\_item is treated as a “do not care” condition during case comparison.

There is also the “?” “do not care” symbol. Whenever a “?” appears in a case item, it will be considered a “do not care.”

Let us look at an example. I will cover “casez” and “casex” both with the same example. Also, we will see how “?” works:

```
module cas;
  logic [7:0] index;

  always @(index) begin
    casez (index)

      'b 1??????? : $display($stime,,, "Case 1?????? is
executed");
      'b 01?????? : $display($stime,,, "Case 01?????? is
executed");
      'b 001zzzzz : $display($stime,,, "Case 001zzzzz is
executed");
      'b 0001xxxx : $display($stime,,, "Case 0001xxxx is
executed");

      default : $display($stime,,, "Default Case is
executed");
    endcase
  end

  initial begin
    #10; index = 8'b1010_1010;
    #10; index = 8'b0101_0101;
    #10; index = 8'b0010_1010;
    #10; index = 8'b0001_0101;
    #10; index = 8'b0000_1111;

    #40 $finish(2);
  end
```

```

initial begin
    $monitor($stime,,, "index = %b",index);
end
endmodule

```

*Simulation log with “casez”:*

```

10 Case 1??????? is executed
10 index = 10101010
20 Case 01?????? is executed
20 index = 01010101
30 Case 001zzzzz is executed
30 index = 00101010
40 Default Case is executed
40 index = 00010101
50 Default Case is executed
50 index = 00001111
$finish called from file "testbench.sv", line 23.
$finish at simulation time 90

```

#### V C S   S i m u l a t i o n   R e p o r t

In this example, we are using “casez” (“casez (index)”). So, whenever there is a “z” in the case\_item, it will be considered a “do not care.” Also, whenever there is a “?,” that will also be considered a “do not care.”

Let us look at simulation log and see what is going on. The first result from the log is:

```

10 Case 1??????? is executed
10 index = 10101010

```

Here the “index” is 10101010 and the case item is 1???????. This means that only the first bit of index needs to be “1.” The rest of the bits of index can be anything. In our case, the first bit of 10101010 is “1,” and since others are “do not care,” the case item 1??????? matches as shown in the log. Note that we have used “?” as a “do not care” and not a “z” or an “x” (which we will see in a while). This is to show that a “?” is also a “do not care” for “casez.”

The next case works on the same line of thought:

```

20 Case 01?????? is executed
20 index = 01010101

```

Let us see and look at the following:

```

30 Case 001zzzzz is executed
30 index = 00101010

```

In this case, the case item is 001zzzzz which means as long as the first three bits of index are “001,” the rest of the bits are “do not care.” So, when index is equal to 00101010, the first 3-bits are “001” and the case item matches the index.

The next log item is interesting. Here we are using “x” as a bit value instead of “z.” Since this is “casez,” “x” bit is *not* a “do not care.” So, when case item is 0001xxxx and index value is 00010101, even though the first four bits match, the “x” bits do not match. Hence, the default case is executed:

```
40 Default Case is executed  
40 index = 00010101
```

And the last log item simply does not match any of the items, and hence the default case is executed:

```
50 Default Case is executed  
50 index = 00001111
```

Now, I am keeping the example identical, except that I change “casez” to “casex.” Let us see how the simulation log looks like.

*Simulation log with “casex”:*

```
10 Case 1??????? is executed  
10 index = 10101010  
20 Case 01?????? is executed  
20 index = 01010101  
30 Case 001zzzzz is executed  
30 index = 00101010  
40 Case 0001xxxx is executed  
40 index = 00010101  
50 Default Case is executed  
50 index = 00001111  
$finish called from file "testbench.sv", line 23.  
$finish at simulation time 90
```

#### V C S   S i m u l a t i o n   R e p o r t

As we discussed, in the case of “casex,” both “z” and “x” are “do not care.” In the first three cases, a “z” as bit value in “index” is a “do not care.” So, the explanation for the first three cases is the same as what we just saw above. But the following case differs:

```
40 Case 0001xxxx is executed  
40 index = 00010101
```

Since this is a “casex,” “x” is a “do not care.” So, when index is 00010101, the first 4-bits match the case\_item and the rest are “do not care.” Hence, 00010101 matches 0001xxxx. In the case of “casez,” this case did not match.

So, what happens if I change either “casez” or “casex” (in the preceding example) to a regular “case”? Here is the simulation log. See if you can figure it out. Hint: Neither “x” nor “z” are “do not care” for a “case.”

*Simulation log with “case”:*

```
0 index = xxxxxxxx
10 Default Case is executed
10 index = 10101010
20 Default Case is executed
20 index = 01010101
30 Default Case is executed
30 index = 00101010
40 Default Case is executed
40 index = 00010101
50 Default Case is executed
50 index = 00001111
```

\$finish called from file "testbench.sv", line 23.

\$finish at simulation time 90

V C S   S i m u l a t i o n   R e p o r t

Here are the truth tables for casex and casez. “1” is a match and “0” is not a match. caseZ truth table is shown in Table 17.1, and caseX truth table is shown in Table 17.2.

**Table 17.1** caseZ truth table

casez	0	1	X	Z
0	1	0	0	1
1	0	1	0	1
X	0	0	1	1
Z	1	1	1	1

**Table 17.2** caseX truth table

casex	0	1	X	Z
0	1	0	1	1
1	0	1	1	1
X	1	1	1	1
Z	1	1	1	1

### 17.2.2 Constant Expression in “case” Statement

A constant expression can be used for the case\_expression (so far, we have seen case\_expression to be a variable). The value of the constant case\_expression is compared against the case\_item\_expression. Here is an example:

```
module cas;
    logic [1:0] index;

    always @(index) begin
        case (1'b1)

            index [1] : $display($stime,,, "Case index[1] is
executed");
            index [0] : $display($stime,,, "Case index[0] is
executed");

        endcase
    end

    initial begin
        #10; index[1:0] = 10;
        #10; index[1:0] = 01;

        #40 $finish(2);
    end

    initial begin
        $monitor($stime,,, "index = %b", index );
    end
endmodule
```

*Simulation log:*

```
0  index = xx
10 Case index[1] is executed
10  index = 10
20 Case index[0] is executed
20  index = 01
$finish called from file "testbench.sv", line 17.
$finish at simulation time          60
V C S  S i m u l a t i o n  R e p o r t
```

A constant is used as the case\_expression (“`case (1'b1)`”). What this means is that the case\_item\_expressions (`index[1]` and `index[0]`) will be compared against

the constant 1'b1. If index[1] is a 1'b1, the case item matches, and if index[0] is a 1'b1, the case item matches. This is evident from the simulation log.

What if you have 1'bx or 1'bz as the constant case expression? There are three scenarios: (1) case (1'bx), (2) casex(1'bx), and (3) casez(1'bz). How will these three cases work? Let us look at an example:

```

module top;
  reg [3:0] a;

  initial begin
    #1;
    case (1'bx) //CASE : none of the case item will execute -
  except default
    1'b1 : $display($stime,,, "CASE:EXECUTED");
    1'b0 : $display($stime,,, "CASE:NOT EXECUTED");
    default: $display($stime,,, "CASE:DEFAULT EXECUTED");
  endcase
  casex (1'bx) //CASEX: always the first item will execute
    1'b1 : $display($stime,,, "CASEX:FIRST EXECUTED");
    1'b0 : $display($stime,,, "CASEX:NOT EXECUTED");
  endcase
  casez (1'bz) //CASEZ: always the first item will execute
    1'b0 : $display($stime,,, "CASEZ:FIRST EXECUTED");
    1'b1 : $display($stime,,, "CASEZ:NOT EXECUTED");
  endcase
  end
  initial begin
    #10;
    a = 4'b1x1x;
    if (a == 4'b1x1x) $display($stime,,, "X detected");
    else $display($stime,,, "FAIL");

    #10;
    if (1'bx) $display($stime,,, "1'bx: X detected");
    else $display($stime,,, "1'bx: FAIL");
  end
endmodule

```

*Simulation log:*

```

# 1 CASE:DEFAULT EXECUTED
# 1 CASEX:FIRST EXECUTED
# 1 CASEZ:FIRST EXECUTED
# 10 FAIL
# 20 1'bx: FAIL

```

With case (1'bx), none of the case items match 1'bx. Hence the default case statement will be executed as shown in the simulation log.

However, with casex(1'bx), it's always the first case item that will be executed. Similarly, with casez(1'bz), it's always the first case item that will be executed.

Also, when you compare an unknown expression in an if statement, the "if" will always fail as you notice in the simulation log.

### 17.2.3 One-Hot State Machine Using Constant Case Expression

Let us now look at a practical example of where constant case expression is useful in designing a one-hot state machine. Consider the following state machine (Fig. 17.1).

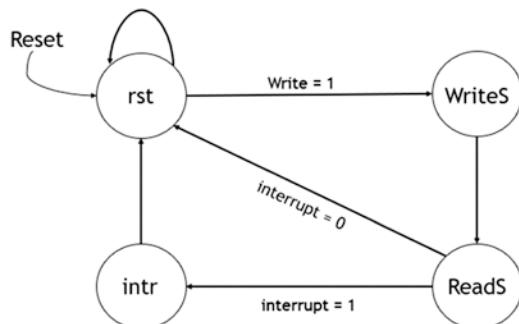
The state transitions are as shown in the figure. When reset is asserted, the FSM goes to the "rst" state. From "rst" if write = 1, state transitions to "WriteS" state. From "WriteS" state it unconditionally transitions to "ReadS" state at the next posedge clock. If interrupt = 1 when in "ReadS" state, the state transitions to "intr" state; if not, the state transitions to "rst" state. And the state transitions unconditionally from state "intr" to "rst" at the next posedge clock.

Here's the code that models the above state machine as a synchronous Moore-style one-hot encoded state machine. We'll see how constant case expression helps with the decoding of one-hot state register:

```
module top;

localparam rst = 0,
          WriteS = 1,
          ReadS = 2,
          intr = 3;
```

**Figure 17.1** Finite-state machine



```
logic [3:0] State;
logic clock, Write, Reset, interrupt;

always_ff @(posedge clock or posedge Reset)
    if (Reset)
begin
    State <= 3'b000;
    State[rst] <= 1'b1;
end
else
begin
    State <= 4'b0000;

    case (1'b1) // synopsys parallel_case
        State[rst] : if (Write)
                        State[WriteS] <= 1'b1;
                    else
                        State[rst] <= 1'b1;
        State[WriteS] : State[ReadS] <= 1'b1;
        State[ReadS] : if (interrupt)
                        State [intr] <= 1'b1;
                    else
                        State[rst] <= 1'b1;
        State[intr] : State[rst] <= 1'b1;
    default: $display($stime,,, "Default CASE executed");
endcase

end

initial begin
    Reset =0; Write = 0; clock=0; interrupt=0;
    #1 Reset = 1; #1; Reset = 0;
    #10; Write = 1;
    #10; interrupt = 1;
    #60 $finish;
end

always #5 clock = !clock;

initial $monitor($stime,,, "State = %b", State);
endmodule
```

*Simulation log:*

```
# run -all
#      0  State = xxxx
#      1  State = 0001
#     15  State = 0010
#     25  State = 0100
#     35  State = 1000
#     45  State = 0001
#     55  State = 0010
#     65  State = 0100
#     75  State = 1000
# ** Note: $finish : testbench.sv(42)
```

The “state” register is 4-bits wide since we need to encode the FSM with one-hot encoding and since there are four states, namely, “rst,” “WriteS,” “ReadS,” and “intr.” The key in one-hot encoded machine is that only 1-bit of the FSM is asserted (=1) at any given time and the rest of the bits remain zero. For example, when we do `State[rst]=1`, we are setting bit 0 of the state register to 1, and the rest of the bits remain zero. So, the four states determine the position of the state in the “state” register. Once you have encoded the FSM with one-hot state register, the decode also needs to be one-hot. That is where the constant case expression comes into picture. As you notice in the case statement, we are using a constant case expression (`case (1'b1)`), meaning compare the case items to be 1-bit true with constant case expression. Only 1-bit of the case item will be true at any given time, and the constant case expression will compare it with `1'b1` (true). We have created a one hot encoded machine.

The simulation log shows that the state transitions are one-hot encoded and transition in one-hot fashion (only 1-bit asserted at a time). This is where constant case expression really gets used.

### 17.2.4 *unique-case, unique0-case, and priority-case*

Just as with unique and priority “if” cases, there are also unique and priority variations of “case” statements. The case, casez, and casex can be qualified with unique, unique0, and priority keywords. Just as with their “if” counterparts, these perform certain violation checks.

A priority-case acts on the first match only. Unique-case and unique0-case assert that there are no overlapping case\_items and hence it is safe for the case\_items to be evaluated in parallel. Since unique and unique0 do not have overlapping case\_items, the case\_expression will be evaluated exactly once and before any of the case\_item\_expressions. The case\_item\_expressions may be evaluated in any order and compared in any order. Unique-case and unique0-case are violated if more than one case\_item is found to match the case\_expression. A violation report is generated.

But note that it is not a violation of uniqueness for a single case\_item to contain more than one case\_item\_expression that matches the case\_expression. Note that unique and unique0 will issue a violation report and execute the statement associated with the matching case\_item that appears first in the “case” statement.

Let us look at an example. I will use this one example to showcase unique, unique0, and priority cases. Let us start with “unique.” I am using “casex,” but the idea equally applies to “case” or “casez”:

```

module cas;
    logic [7:0] index;

    always @(index) begin
        unique casex (index)

            'b 1xxxxxxxx : $display($stime,,, "Case 1xxxxxxxx is
executed");
            'b 10xxxxxx : $display($stime,,, "Case 10xxxxxx is
executed");
            default : $display($stime,,, "Default Case is
executed");

        endcase
    end

    initial begin
        #10; index = 8'b1010_1010;
        #10; index = 8'b0101_0101;
        #40 $finish(2);
    end

    initial begin
        $monitor($stime,,, "index = %b", index);
    end
endmodule

```

The first thing you notice in this example is that the following two cases overlap:

```

'b 1xxxxxxxx : $display($stime,,, "Case 1xxxxxxxx is executed");
'b 10xxxxxx : $display($stime,,, "Case 10xxxxxx is executed");

```

“1xxxxxxxx” has first bit as “1” and the rest are “do not care.” The second case\_item is “10xxxxxx” where the second bit “0” overlaps with the “do not care” second bit of “1xxxxxxxx.” Hence, in this case, you get the following warning (Synopsys – VCS):

Warning-[RT-MTOCMUCS] More than one condition match in statement

testbench.sv, 5

More than one condition matches are found in 'unique case' statement inside cas, at time 10ns.

Line number 7 and 8 are overlapping.

Similarly, if you change “unique” to “unique0” in the preceding example, you will get the same warning.

Let us now look at the “unique” example but without the “default” case. As we saw with “if,” if there is no default case with “unique,” you will get a warning. Here is the example without the “default” case:

```
module cas;
    logic [7:0] index;

    always @ (index) begin
        unique casex (index)

            'b 1xxxxxxx : $display($stime,,, "Case 1xxxxxxx is
executed");
            'b 10xxxxxx : $display($stime,,, "Case 10xxxxxx is
executed");
            //default : $display($stime,,, "Default Case is
executed");

        endcase
    end

    initial begin
        #10; index = 8'b1010_1010;
        #10; index = 8'b0101_0101;
        #40 $finish(2);
    end

    initial begin
        $monitor($stime,,, "index = %b",index);
    end
endmodule
```

With this example, you will get two warnings: one for the overlap and one for the missing “default” case.

*Simulation log:*

10 Case 1xxxxxxx is executed

Warning-[RT-MTOCMUCS] More than one condition match in statement

testbench.sv, 5

More than one condition matches are found in 'unique case' statement inside cas, at time 10ns.

Line number 7 and 8 are overlapping.

```
10 index = 10101010
```

Warning-[RT-NCMUCS] No condition matches in statement  
testbench.sv, 5

No condition matches in 'unique case' statement. 'default' specification is missing, inside cas, at time 20ns.

```
20 index = 01010101
```

\$finish called from file "testbench.sv", line 17.

\$finish at simulation time 60

V C S   S i m u l a t i o n   R e p o r t

And just with the "if" case, if we replace "unique" with "unique0" in the example above, we will not get a warning for the missing "default" case.

Let us look at the same example but with "priority" keyword:

```
module cas;
    logic [7:0] index;

    always @ (index) begin
        priority casex (index)

            'b 1xxxxxxxx : $display($stime,,, "Case 1xxxxxxxx is
executed");
            'b 10xxxxxxxx : $display($stime,,, "Case 10xxxxxxxx is
executed");
            default : $display($stime,,, "Default Case is
executed");

        endcase
    end

    initial begin
        #10; index = 8'b1010_1010;
        #10; index = 8'b0101_0101;
        #40 $finish(2);
    end

    initial begin
        $monitor($stime,,, "index = %b", index);
    end
endmodule
```

With priority case, the case statements are executed in the order in which they are listed (not executed in parallel in any order as with unique cases). Hence, when it encountered the first case\_item, it executed it and was done with the case. So, even though the second case\_item overlaps with the first one, we will not get a warning. Hence, the simulation log looks as shown below.

*Simulation log:*

```
0 index = xxxxxxxx
10 Case 1xxxxxxxx is executed
10 index = 10101010
20 Default Case is executed
20 index = 01010101
$finish called from file "testbench.sv", line 17.
$finish at simulation time           60
V C S   S i m u l a t i o n   R e p o r t
```

## 17.3 Loop Statements

We are all familiar with looping statements of C/C++ languages. SystemVerilog provides six different types of looping constructs:

- for
- while
- do - while
- foreach
- forever
- repeat

Let us look at each.

### 17.3.1 *The “for” Loop*

Here is a simple example:

```
module forloop;
    int i;
    int mem[0:3];

    initial begin
        //control variable declared within the loop
        outerLoop: for (int j = 0; j < 2; j++)
            begin
```

```

#10;
//control variable declared outside the loop
innerLoop: for (i = 0; i < 3; i++)
begin
    mem[i] = i+1;
    $display($stime,,, "mem[%0d] = %0d", i, mem[i]);
end

end
end
endmodule

```

*Simulation log:*

```

10 mem[0] = 1
10 mem[1] = 2
10 mem[2] = 3
20 mem[0] = 1
20 mem[1] = 2
20 mem[2] = 3

```

V C S   S i m u l a t i o n   R e p o r t

A couple of points.

The variable used to control the for loop can be declared prior to the loop (“i” in our example). Need to be careful though that if loops in two or more parallel processes use the same loop control variable, there is a potential that one loop may clobber the variable, while the other loop is still executing on it. Rather obvious, but easy to miss.

In contrast, the variable used to control the for loop can be declared within the loop itself (“j” in our example). This creates an implicit begin-end block around the loop. Such an implicit begin-end block creates a new hierarchical scope, making the variables local to the loop scope.

Note that you can have more than one comma-separated variables in a for loop. For example:

```
for (int i = 0, j = 0; i*j < 3; i++, j++)
```

In such a case, either all or none of the control variables need to be locally declared. So, the following would be illegal (“i” is not local but “j” is local):

```
for (i = 0, int j = 0, ...)
```

Note the labels used for the for loops in our example (“outerLoop” and “innerLoop”). These are optional but are highly recommended, since other parallel loops cannot inadvertently affect the loop control variable of the named loop blocks.

### 17.3.2 The “repeat” Loop

The repeat loop executes a statement of a fixed number of times (repeat expression). Here is an example to show how repeat loop works:

```

module forloop;
    int i;
    int mem[0:7];
    bit clk;
    parameter repeatNum = 3;

    initial begin

        outerLoop: for (int j = 0; j < 2; j++)
            begin

                innerLoop: repeat (repeatNum) //REPEAT LOOP
                begin
                    @ (posedge clk);
                    i++;
                    storeMem ( i ); //task call
                    $display ($stime, , "mem[%0d] = %0d", i, mem[i]);
                end

            end
        end

        task storeMem ( int i );
            begin
                mem[i] = i + 1;
            end
        endtask

        initial begin
            clk = 0;
            #120 $finish(2);
        end

        always #10 clk = !clk;
    endmodule

```

*Simulation log:*

10 mem[1] = 2

```

30  mem[2] = 3
50  mem[3] = 4
70  mem[4] = 5
90  mem[5] = 6
110 mem[6] = 7
$finish called from file "testbench.sv", line 31.
$finish at simulation time          120
V C S   S i m u l a t i o n   R e p o r t

```

In this example, I have mixed a for loop and a repeat loop. The repeat loop (“innerLoop”) repeats itself “repeatNum” times (“repeatNum” is a parameter). Inside the begin-end of the “repeat” loop, you can code any procedural statements that you would normally use in a begin-end block. The entire begin-end block will be executed “repeatNum” times. Inside this begin-end, we are waiting for @ (posedge clk) and then calling a task “storeMem” to store value of “i” in the memory “mem.” The idea behind this code is to show that you can have complex code within the repeat loop.

Note that if the repeat expression (“repeatNum” is our example) is “x” or “z,” the repeat count will be considered zero, and no statements associated with “repeat” will be executed.

### 17.3.3 *The “foreach” Loop*

The foreach loop construct specifies iteration over the elements of an array. Its argument is an identifier that designates any type of array enclosed in square brackets. You may have multi-dimensional array, in which case you may have a comma-separated list of loop variables. Each loop variable corresponds to one of the dimensions of the array.

Here is an example of a multi-dimensional packed array:

```

module tb;
  bit [2:0][1:0][7:0] m_data; //3D packed array

  initial begin

    //Assign the entire array with a single value
    m_data = 48'hcafe_face_0708;

    foreach (m_data[i]) begin
      $display ("m_data[%0d] = 0x%h", i, m_data[i]);
      foreach (m_data[i, j]) begin
        //comma separated list of loop variables

```

```

        $display ("m_data[%0d] [%0d] = 0x%h", i, j, m_
data[i][j]);
    end
end
end
endmodule

```

The example has a 3D packed array “m\_data.” Since this is a packed array, the single largest contiguous accessible data is 48-bit wide ( $[2:0][1:0][7:0] = 3*2*8 = 48$ ).

In order to access each element of the array m\_data, we use a foreach loop iterating over the dimensions [2:0][1:0] (the last dimension [7:0] is the smallest contiguous data accessible and not iterated).

First, we iterate over the dimension [2:0] and display the 16-bit ( $[1:0][7:0] = 2*8 = 16$ ) content of each of the elements of [2:0]. In other words, when we pick m\_data[i], it iterates over the first dimension [2:0]:

```

foreach (m_data[i]) begin
    $display ("m_data[%0d] = 0x%h", i, m_data[i]);

```

Next, we iterate over the entire array [2:0][1:0] using the following:

```

foreach (m_data[i, j]) begin
    $display ("m_data[%0d] [%0d] = 0x%h", i, j, m_
data[i][j]);

```

Here, m[i, j] means iterate over [2:0] followed by iteration over [1:0]. The accessible data is the byte [7:0].

*Simulation log: It shows how the array elements are looped to access array elements:*

```

m_data[2] = 0xcafe
m_data[2][1] = 0xca
m_data[2][0] = 0xfe
m_data[1][1] = 0xfa
m_data[1][0] = 0xce
m_data[0][1] = 0x07
m_data[0][0] = 0x08
m_data[1] = 0xface
m_data[2][1] = 0xca
m_data[2][0] = 0xfe
m_data[1][1] = 0xfa
m_data[1][0] = 0xce
m_data[0][1] = 0x07
m_data[0][0] = 0x08
m_data[0] = 0x0708

```

```
m_data[2][1] = 0xca
m_data[2][0] = 0xfe
m_data[1][1] = 0xfa
m_data[1][0] = 0xce
m_data[0][1] = 0x07
m_data[0][0] = 0x08
```

### V C S   S i m u l a t i o n   R e p o r t

Now, what if you accessed the elements of the array the following way?

```
foreach (m_data[i]) begin
    $display ("m_data[%0d] = 0x%h", i, m_data[i]);
    foreach (m_data[i][j]) begin
        //NOT a comma separated list of loop variables
        $display ("m_data[%0d][%0d] = 0x%h", i, j, m_
data[i][j]);
    end
end
```

You will get the following simulation results. *Exercise:* Can you figure out why the results are different from the ones we got when used a comma separated list of loop variables?

```
m_data[2] = 0xcafe
m_data[2][1] = 0xca
m_data[2][0] = 0xfe
m_data[1] = 0xface
m_data[1][1] = 0xfa
m_data[1][0] = 0xce
m_data[0] = 0x0708
m_data[0][1] = 0x07
m_data[0][0] = 0x08
```

### V C S   S i m u l a t i o n   R e p o r t

Here is an example of how mapping of loop variables to array indices are determined:

```
//      1   2   3
int intA [2][3][4];
foreach (intA [i, j, k]) ....
```

This foreach loop causes “i” to iterate from 0 to 1, “j” from 0 to 2, and “k” from 0 to 3:

```
//      3      4      1      2
bit [3:0][2:1] B [5:1][4]
foreach ( B [q, r, , s] )...
```

This foreach loop causes “q” to iterate from 5 to 1, “r” to iterate from 0 to 3, and “s” to iterate from 2 to 1. Iteration over the third index is skipped.

Some more points:

- The number of loop variables cannot be greater than the number of dimensions of the array variable.
- Loop variables may be omitted to indicate no iteration over that dimension of the array.
- Just as in for loop, the foreach loop creates an implicit begin-end block. Such implicit begin-end block creates a new hierarchical scope, making the variables local to the foreach loop scope. As we saw, such a block is unnamed by default but can be named adding a statement label.
- Fforeach loop variables are read-only.
- The type of each loop variable is implicitly declared to be consistent with the type of the array index.

### 17.3.4 The “while” Loop

The while loop repeatedly executes a statement as long as a control expression is true. Here is an example:

```
module tb;
    logic [2:0] count;
    logic [3:0] mem[0:7];

    initial begin
        count = 4;
        while (count)
            begin
                mem[count] = $urandom;
                $display($stime,,, "mem[%0d] = %0h", count,
mem[count]);
                count--;
            end
        end
    endmodule
```

The control expression in this example is “count.” While “count” is true, the loop will execute. We start with “count = 4” and decrement it through each iteration. Hence, the loop will execute four times, as shown in the simulation log.

*Simulation log:*

```
0 mem[4] = 4
0 mem[3] = 1
0 mem[2] = 9
0 mem[1] = 3
V C S S i m u l a t i o n R e p o r t
```

### 17.3.5 The “do – while” Loop

The do – while loop differs from the while loop in that a do – while-loop tests its control expression at the

*end* of the loop. I have simply taken the preceding example and changed it to check for the control expression at the end of the loop+

```
module tb;
    logic [2:0] count;
    logic [3:0] mem[0:7];

    initial begin
        count = 4;
        do begin
            mem[count] = $urandom;
            $display($stime,,,"mem[%0d] = %0h",count,mem[count]);
            count--;
        end
        while (count);
    end

```

The “do begin” starts the loop. It decrements “count” through each iteration of the loop. At the end of the “do” block, we check for the control expression “count” and continue executing the “do” block as long as ‘count’ is true.

*Simulation log:*

```
0 mem[4] = 4
0 mem[3] = 1
0 mem[2] = 9
0 mem[1] = 3
V C S S i m u l a t i o n R e p o r t
```

### 17.3.6 The “forever” Loop

As the name suggests, the “forever” loop repeatedly executes a statement forever (till the end of simulation time). Since this is a forever loop, you should have some sort of timing control within the loop; else you will end up in a 0-delay loop, which will hang your simulation (0-delay stuck in time). The “forever” loop is most useful in the “initial” block because you can’t have an “always” block within the “initial” block. Here is an example of clock generator:

```
module tb;
    bit clk;
    bit [7:0] count;

    initial begin
        clk = 0;
        forever begin
            #10 clk = !clk;
            $display ($stime,,, "clk = %0d",clk);
        end
    end

    initial begin
        #60; $finish(2);
    end
endmodule
```

*Simulation log:*

```
10  clk = 1
20  clk = 0
30  clk = 1
40  clk = 0
50  clk = 1
```

\$finish called from file "testbench.sv", line 25.

\$finish at simulation time 60  
V C S S i m u l a t i o n R e p o r t

Here, within the “initial” block, we create a “forever” loop to generate a clock. This “forever” will execute till the end of simulation.

Note that “forever” is not restricted to an “initial” block. You can use it in an “always” block as well. *But I do not recommend it.* Embedding a “forever” executing block within an “always” executing block may result in unintended results. Avoid it. But for the sake of completeness, here is an example of it:

```
module tb;
    bit clk;
    bit [7:0] count;

    initial begin
        clk = 0;
        forever begin
            #10 clk = !clk;
        end
    end

    always @ (posedge clk) begin
        $display ($stime,,, "clk = %0d",clk);
        forever begin //Allowed but -not- recommended
            #10;
            count = count + 1;
            $display($stime,,, "count = %0d", count);
        end
    end

    initial begin
        #60; $finish(2);
    end
endmodule
```

Here we embed a “forever” loop within an “always” loop. So, in @ (posedge clk) we enter the “always” block and then execute the “forever” block forever. Once a (posedge clk) arrives, we will remain in the “forever” loop forever! This way of embedding two “always” executing blocks within each other is not recommended, unless you have no other choice.

*Simulation log:*

```
10  clk = 1
20  count = 1
30  count = 2
40  count = 3
50  count = 4
```

\$finish called from file "testbench.sv", line 23.

\$finish at simulation time 60

V C S   S i m u l a t i o n   R e p o r t

## 17.4 Jump Statements

### 17.4.1 “break” and “continue”

“break” breaks out of a loop and “continue” skips to the end of the loop. These controls are exactly like they are in “C” language. Here is an example of how to break out of a loop:

```
module tb;
    logic [2:0] count;
    logic [3:0] mem[0:7];

    initial begin
        count = 4;
        while (count)
            begin
                #10;
                mem[count] = $urandom;
                $display($stime,,, "mem[%0d] = %0h", count, mem[count]);
                count--;
                break;
            end
        $display($stime,,, "After 'while' loop");
    end

    initial #60 $finish(2);
endmodule
```

This example is similar to preceding example on while loop. But here, we break out of the loop after the first iteration. Hence, “mem[count] = \$urandom” will be executed only once (as evident from the simulation log). Once you “break” out of the loop, the control passes to the statement following the loop block.

*Simulation log:*

```
10 mem[4] = 4
10 After 'while' loop
$finish at simulation time          60
                                V C S  Simulation Report
```

Now let us look at how “continue” works. Same example as above, but we simply “continue” once we enter the loop. So, the statement “mem[count] = \$urandom” will never get executed (and the \$display will never be executed). The loop will complete (because we decrement the “count” before we “continue”), and the control will pass to statements after the looping block:

```
module tb;
    logic [2:0] count;
    logic [3:0] mem[0:7];

    initial begin
        count = 4;
        while (count)
            begin
                #10;
                count--;
                continue;
                mem[count] = $urandom;
                $display($stime,,, "mem[%0d] = %0h", count, mem[count]);
            end

        $display($stime,,, "After 'while' loop");
    end

    initial #60 $finish(2);
endmodule
```

*Simulation log:*

```
40  After 'while' loop
$finish at simulation time          60
V C S   S i m u l a t i o n   R e p o r t
```

In this example, we “continue” with the loop before we assign “mem[count] = \$urandom.” So, this statement will never be executed since we skip the loop (“continue”) just before we make the assignment. Hence, simulation log shows at time 40 the statement after the “while” loop (time 40 because the loop repeats four times each with timestep of 10). To reiterate, “continue” skips the current loop execution and continues with the next iteration of the loop; it does not break out of the loop.

# Chapter 18

## Inter-process Synchronization.

### Semaphores and Mailboxes



**Introduction** This chapter discusses inter-process synchronization mechanisms such as semaphores and mailboxes, including semaphore and mailbox methods, parameterized mailbox, etc.

### 18.1 Semaphores

Semaphore is a mechanism that allows you to synchronize two or more processes trying to access a shared resource. High-level and easy-to-use synchronization and communication mechanisms are essential to control the kinds of interactions that occur between dynamic processes used to model a complex system or a highly reactive testbench.

For example, in an SoC, both CPU and DMA would want to write and read from the same memory. What happens if both CPU and DMA start to write to the same memory location? CPU data will be overwritten by DMA or vice versa. Essentially, you have data corruption. If CPU then wants to read the data it had written, it may end up reading the data that DMA had written. This is a dangerous situation. That is where a semaphore comes into picture. When CPU wants to write, it will first “lock” the memory access with a “key” (i.e., use a semaphore to “get” the key); write the data to memory. It will not “release” (“put”) this key until it is done reading (or doing anything else) with the data that it has written to memory. As long as CPU has locked the memory with a key, DMA cannot access the memory. Once the CPU is done with its accesses to the memory, it will “release” the key to memory (“put” the

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_18](https://doi.org/10.1007/978-3-030-71319-5_18)) contains supplementary material, which is available to authorized users.

key back for someone else to grab). DMA has been waiting to “get” that key. It “gets” the key that is “put” by CPU and starts accessing the memory.

So, semaphores work off a key. Conceptually, a semaphore is like a bucket. When a semaphore is allocated, a bucket that contains a fixed number of keys is created. Processes using semaphores must first procure a key from the bucket before they can continue to execute. If a specific process requires a key, only a fixed number of occurrences of that process can be in progress simultaneously. All others must wait until a sufficient number of keys are returned to the bucket. Semaphores are typically used for mutual exclusion, access control to shared resources, and basic synchronization.

Here is the syntax:

```
semaphore semaphore_name;
```

It is a *built-in class* that provides certain methods to manage the keys. These methods are:

**new ( )**: Create a semaphore with a specified number of keys:

```
function new (int keyCount = 0);
```

The default keyCount is 0. The keyCount specifies the number of keys initially allocated to the semaphore bucket. The default value for keyCount is 0. The new( ) function returns the semaphore handle.

**get ( )**: Get one or more keys from the bucket (blocking task):

```
task get (int keyCount = 1);
```

The default is 1. The keyCount specifies the required number of keys to obtain (get) from the semaphore. If the specified number of keys is available, the method returns, and execution continues. If the specified number of keys is not available, the process *blocks* until the keys become available. *The semaphore's waiting queue is first-in, first-out (FIFO)*. This does not guarantee the order in which processes arrive at the queue, only that their arrival order will be preserved by the semaphore.

**put ( )**: Return one or more keys to the bucket (non-blocking function):

```
function void put (int keyCount = 1);
```

The default is 1. The keyCount specifies the number of keys returned (put) to the semaphore. If the specified number of keys is available, the method returns a positive integer and execution continues. If the specified number of keys is not available, the method returns 0.

**try\_get ( )**: Try to obtain one or more keys (non-blocking function):

```
function int try_get (int keyCount = 1);
```

Same as get ( ) but non-blocking. It is used to procure a specified number of keys from a semaphore. The default is 1. If the specified number of keys is available, the method returns a positive integer and execution continues. If the specified number of keys is not available, the method returns 0.

Here is an example:

```
module sema;

bit [7:0] mem [0:3];
int i, data;

semaphore s1;

initial begin
    s1 = new (1); //Create semaphore with 1 key
    fork
        DMA_write;
        CPU_read;
    join
end

task DMA_write;
    if (s1.try_get(1)) //non-blocking. Locks (gets)
        $display($stime,,, "DMA gets a KEY from semaphore");
    else
        wait (s1.try_get(1));

    //DMA writes data
    for (i=0; i < 4; i++) begin
        mem[i] = $urandom;
        $display($stime,,, "DMA WRITE[%0d] = %0d",i,mem[i]);
    end

    #5; //do something else

    s1.put(1); //DMA releases (puts) the key
    $display($stime,,, "DMA puts the KEY into semaphore");
endtask

task CPU_read;
    #0; s1.get (1); //WAIT to get the key - blocking
    $display($stime,,, "CPU gets the KEY from semaphore");

    //CPU reads data
    for (i=0; i < 4; i++) begin
        data = mem[i];
        $display($stime,,, "CPU READ[%0d] = %0d",i,data);
    end
```

```
endtask
endmodule
```

There are two processes fired concurrently, the DMA\_write and CPU\_read. DMA writes to the memory, and CPU reads from the memory (what is written by DMA).

We need to make sure that CPU reads only – after – DMA has completed writing (else it will get garbage data). Both processes try to get a key right away at time 0. This is a race condition. Hence, you have to delay the CPU\_read at time 0. We do that by scheduling CPU\_read at the end of the time step using #0. DMA gets the key first (because of try\_get). This means that DMA has now locked the memory. At the same time, CPU also has requested a key (s1.get(1)). But since DMA got the key and locked the memory, CPU’s “get” will be blocked and will wait until DMA releases the key.

DMA then writes the data to memory, waits for five time units, and then releases (s1.put(1)) the key. This means that CPU which had been waiting for the one key that was locked by DMA and now can move forward since its “get” now gets the key put by DMA. CPU then starts to read the memory that DMA had written and gets the correct data as shown in the simulation log.

So, one has to be careful with semaphores in how you put and get keys when concurrent processes try to access the same key.

*Simulation log:*

```
0 DMA gets a KEY from semaphore
0 DMA WRITE[0] = 36
0 DMA WRITE[1] = 129
0 DMA WRITE[2] = 9
0 DMA WRITE[3] = 99
5 DMA puts the KEY into semaphore
5 CPU gets the KEY from semaphore
5 CPU READ[0] = 36
5 CPU READ[1] = 129
5 CPU READ[2] = 9
5 CPU READ[3] = 99
```

V C S   S i m u l a t i o n   R e p o r t

## 18.2 Mailboxes

Mailboxes are another way of inter-process communication, just as semaphores. But, in my opinion, a lot more powerful and easier to use. It is a communication mechanism that allows messages to be exchanged between two processes. Data can be sent by one process and retrieved by another process, just as in your regular mailbox that you use to receive letters from your loved ones (or junk!!) and retrieve once

the mail is delivered to the mailbox. When you check for your letters, if the letter has not yet been delivered, you can do one of two things. One, wait for the letter to arrive or come back later to re-check for the arrival of letters. Similarly, SystemVerilog's mailboxes provide processes to send and retrieve data in a controlled manner.

There are two types of mailboxes: bounded and unbounded. Bounded mailbox allows only those many messages to be stored as its bound allows. If a process attempts to place a message into a full mailbox, the process will be suspended until enough space is available in the mailbox. Unbounded mailboxes are unlimited in size and never suspend a process in a send operation.

By default, the mailbox is typeless, which means that the mailbox can send and receive any type of data.

Syntax for mailbox declaration is:

```
mailbox mbox;
```

Mailbox is a built-in class with provision for many methods:

```
new()
```

The new() method creates a mailbox. Its syntax is:

```
function new() (int bound = 0);
```

Just as with class, when you create/instantiate a mailbox, it returns a mailbox handle. If the bound argument is zero, then it is an unbounded mailbox. Unbounded is the default. If the bound is non-zero, it defines the size of the mailbox. Obviously, the bound needs to be positive.

```
num()
```

The number of messages available in a mailbox can be obtained via the num() method. The syntax is:

```
function int num();
```

```
put()
```

The put() method puts a message in a mailbox. The syntax is:

```
task put(singular message)
```

This method stores a message in a mailbox (in a FIFO order). As noted above, if the mailbox is bounded and full, the process will block until there is enough room to store new messages. The message is any singular expression, including object handles.

```
try_put()
```

In contrast to put(), the try\_put() method *attempts* (non-blocking) to put a message in the mailbox. This can be viewed as non-blocking put(). It stores the message in the mailbox in a FIFO order. It makes sense to use this method only with bounded mailboxes. If the mailbox is full, the method returns a 0; else the message is placed in the mailbox and a positive integer is returned. The syntax is:

```
function try_put (singular message);
get()
```

The get( ) method retrieves a message from the mailbox. The syntax is:

```
task get ( ref singular message);
```

The get( ) method removes one message from the mailbox queue. If the mailbox is empty, then the current process blocks until a message is placed in the mailbox.

```
try_get()
```

Non-blocking get( ). Analogous to try\_put( ), the try\_get( ) method *attempts* (non-blocking) to retrieve a message from the mailbox without blocking. The syntax is:

```
function int try_get( ) (ref singular expression)
```

The message can be any singular expression (it will be a valid left-hand expression). The functionality is similar to get( ), except that this method is a function and non-blocking. If the mailbox is empty, the method returns a 0; else if a message is available, it returns a positive integer.

```
peek()
```

As the name suggests, the method allows you to copy a message from the mailbox without removing it which is in contrast to get( ), which removes a message from the mailbox. The syntax is:

```
task peek (ref singular message);
```

If the mailbox is empty, the current process blocks until a message is available.

As long as a message is available in the mailbox queue, any process blocked in a peek( ) (or get( )) operation will become unblocked.

```
try_peek()
```

The try\_peek method *attempts* to copy a message from the mailbox without blocking (it's a function). The syntax is:

```
function int try_peek ( ref singular expression)
```

It does not remove the message from the mailbox. If the mailbox is empty, the method returns a 0. If the mailbox is not empty, and the type of message is equivalent to the type of message variable, the message is copied, and a positive integer is returned.

Let us look at an example. I am taking the same example as the one we saw under semaphores. The point is that it is much easier to pass transactions between two processes using a mailbox than a semaphore. No need to put a key, get a key, make sure that there is no race condition between two get( ), etc. Compare this example with the one under semaphore section to see the difference:

```
module mB;

bit [7:0] mem [0:3];
int i, j, data;

mailbox mbox; //declare a mailbox


initial begin
    mbox = new (4); //create a bounded mailbox
    fork
        DMA_write;
        CPU_read;
    join
end

task DMA_write;
    $display($stime,,, "DMA puts Mem Data into mbox");
    for (i=0; i < 4; i++) begin
        mem[i] = $urandom;
        $display($stime,,, "DMA WRITE[%0d] = %0d",i,mem[i]);
        mbox.put(mem[i]); //put data into the mailbox
    end
endtask

task CPU_read;
    $display($stime,,, "CPU retrieves Mem Data from mbox");
    for (j=0; j < 4; j++) begin

        mbox.get(data); //retrieve data from the mailbox
        $display($stime,,, "CPU READ[%0d] = %0d",i,data);
    end
endtask
endmodule
```

*Simulation log:*

```
0 DMA puts Mem Data into mbox
0 DMA WRITE[0] = 36
```

```

0 DMA WRITE[1] = 129
0 DMA WRITE[2] = 9
0 DMA WRITE[3] = 99
0 CPU retrieves Mem Data from mbox
0 CPU READ[0] = 36
0 CPU READ[1] = 129
0 CPU READ[2] = 9
0 CPU READ[3] = 99

```

### V C S   S i m u l a t i o n   R e p o r t

The example first declares a mailbox called “mbox” and instantiates/creates it with an upper bound of 4. We fork of two processes, “DMA\_write” and “CPU\_read.” These are concurrent processes. DMA\_write puts data into the mailbox, and “CPU\_read” retrieves data from the mailbox. Note that the processes are concurrent, but the “CPU\_read” waits until “DMA\_write” has written (put) data into the mailbox. “put( )” is a blocking method. Only when “DMA\_write” puts a transaction (data) into the mailbox that the “CPU\_read” can receive it. And the two methods put( ) and get( ) are automatically synchronized by the mailbox.

Now, I am converting this example into a class-based example. Such SystemVerilog class-based methodology is where the mailbox will most likely get used:

```

class transaction;
    rand bit [7:0] addr;
    rand bit [7:0] data;

    function void disp;
        $display("Transaction :: Transaction Generated");
        $display("Transaction :: Addr=%0d,Data=%0d",addr,data);
    endfunction

endclass

```

```
//-----
```

```

class DMA_write;
    transaction tr;
    mailbox DMA_m_box;

//constructor, getting mailbox handle
function new(mailbox m_box);
    DMA_m_box = m_box;
endfunction

```

```
task run;
  repeat(2) begin
    tr = new( );
    tr.randomize( ); //generating transaction
    tr.disp( );
    DMA_m_box.put(tr); //putting transaction into mailbox
    $display("DMA_write::Transaction Put into Mailbox");
    #5;
  end
endtask
endclass

//-----


class CPU_read;
  transaction tr;
  mailbox CPU_m_box;

  //constructor, getting mailbox handle
  function new(mailbox m_box);
    CPU_m_box = m_box;
  endfunction

  task run;
    repeat(2) begin
      CPU_m_box.get(tr); //getting transaction from mailbox
      $display("CPU_read::Transactions Received");
      $display("CPU_read::Addr=%0d,Data=%0d\n",tr.addr, tr.data);
    end
  endtask
endclass

//-----


module mailbox_top;
  DMA_write dma;
  CPU_read cpu;
  mailbox m_box; //declaring mailbox m_box

  initial begin
    m_box = new( ); //creating mailbox

  //create 'dma' and pass mailbox handle m_box
```

```

dma = new(m_box);

//create 'cpu' and pass mailbox handle m_box
cpu = new(m_box);

fork
    cpu.run();
    dma.run();
join
end
endmodule

```

*Simulation log:*

```

Transaction :: Transaction Generated
Transaction :: Addr=40,Data=169
DMA_write::Transaction Put into Mailbox
CPU_read::Transaction Received
CPU_read::Addr=40,Data=169

```

```

Transaction :: Transaction Generated
Transaction :: Addr=223,Data=184
DMA_write::Transaction Put into Mailbox
CPU_read::Transaction Received
CPU_read::Addr=223,Data=184

```

V C S   S i m u l a t i o n   R e p o r t

We declare three classes: “transaction,” “DMA\_write,” and “CPU\_read.”

The “transaction” class is a simple class that defines a bus transaction with “rand” variables, so that when this class is randomized, we get random “addr” and “data.”

The “DMA\_write” class declares a mailbox called “DMA\_m\_box.” In its constructor function “new,” we parameterize it with a handle of type mailbox. The class instantiates the “transaction” class (with the name “tr”), randomizes it, and puts the resulting transaction into “m\_box” (DMA\_m\_box.put(tr); ).

Similarly, the “CPU\_read” class declares a mailbox called CPU\_“m\_box.” In its constructor function “new,” we parameterize it with a handle of type mailbox. The class gets a transaction (that was “put” by DMA\_write) (CPU\_m\_box.get(tr); ).

In the module “mailbox\_top,” we instantiate a mailbox with the handle name m\_box. We pass this handle to both the DMA\_write and CPU\_read classes. This is important because we are putting and retrieving transactions from the same mailbox. Same mailbox should be shared in order to communicate. We then fork off two processes, dma.run and cpu.run, to exchange transactions between “dma” and “cpu.” As you see from the simulation log, the first DMA\_write puts the transaction into the m\_box mailbox and CPU\_read and then retrieves this transaction. Note that we have put a delay of #5 in the class DMA\_write. This makes sure that once DMA\_write “put’s a transaction, there is time for CPU\_read to “get” the

transaction. Also, note that we fork off “DMA\_write” and “CPU\_read” from a “fork” statement in “mailbox\_top.” This is an asynchronous way of communicating via a mailbox.

**Exercise:** What happens if you remove the delay “#5” from the class “DMA\_write”?

### 18.2.1 Parameterized Mailbox

The mailbox that we have seen so far is called a generic mailbox. In other words, it is a typeless mailbox. It can transmit/receive data of any type. That is very powerful. But it can cause a run-time problem if the transmit data type is not the same as the receiver data type. So, the language provides a parameterized mailbox, so that type mismatch can be caught at compile time.

The syntax is:

```
mailbox #(type) mailbox_name;
```

Here is a simple example ((SystemVerilog\_LRM\_1800-2012)):

```
module pMailbox;

    typedef mailbox #(string) string_mbox;
    string s;

    initial begin

        static string_mbox SMbox = new;

        s = "hi";

        SMbox.put( s );
        $display("String 'put' is %s", s);

        SMbox.get( s );
        $display("String 'get' is %s", s);

    end
endmodule
```

*Simulation log:*

```
# run -all  
# String 'put' is hi  
# String 'get' is hi  
# exit
```

Finally, named events also help with inter-process synchronization. This is discussed in Sect. 2.12.

# Chapter 19

## Clocking Blocks



**Introduction** This chapter discusses finer nuances of clocking blocks, including clocking blocks with interfaces, global clocking, etc.

A clocking block can be viewed as a synchronizer block between a testbench and a DUT. One of the problems in applying stimulus and evaluating response is that the testbench could end up applying stimulus to the DUT the same time that the DUT samples this stimulus. Will the DUT get the latest/new value of the stimulus or the older values? Same issue occurs when the DUT drives signals that are sampled by the testbench. What you really need is that the testbench should drive signals, a setup time (as in hardware) before the clock edge, and then sample the outputs from DUT, a hold time after the clock edge.

That is where a clocking block comes into picture. It assembles signals that are synchronous to a particular clock and makes their drive/sense timing explicit. Clocking blocks have been introduced in SystemVerilog to *address the problem of specifying the timing and synchronization requirements* of a design in a testbench.

*A clocking block is a set of signals synchronized on a particular clock.* It basically separates the time-related detail from the functional and procedural elements of a testbench. It helps the designer to develop testbenches in terms of transactions and cycles. Rather than focusing on signals and transitions in time, the test can be defined in terms of cycles and transactions. A clocking block provides the timing, relative to a clock event, that the testbench uses to drive and sample signals to/from a DUT. The clocking block construct provides testbenches with a method of easily defining the timing of testbench/DUT interfaces with a defined clock, built-in skew, and constructs that allow stimulus in testbenches to be defined by the clock in a better cohesive way.

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_19](https://doi.org/10.1007/978-3-030-71319-5_19)) contains supplementary material, which is available to authorized users.

A testbench can contain one or more clocking blocks, each containing its own clock plus an arbitrary number of signals.

In its simplest form, the syntax is:

```
clocking clocking_block_name @(clocking_event);
    default input input_skew output output_skew;
    input <input_skew> signal_name;
    output <output_skew> signal_name;
endclocking
```

The `clocking_event` is the synchronizing event – such as `@(posedge clk)`. The `clocking` skews (input and output) provide the input setup (when the inputs are to be sampled) and output hold (when the outputs should be driven) skews with respect to the clocking event. You can specify the skew either with the `input/output` signal declaration or as default skew.

Here is the graphical representation of how signals are sampled/driven by a clocking block (Fig. 19.1).

For example:

```
clocking clock1 @(posedge clk);
    default input #1 output #2;
    input from_dut;
    output to_dut;
endclocking
```

The `clocking event` is `@(posedge clk)` which is the synchronizing event. The example defines default skews for inputs and outputs. The skews are relative to the synchronizing event (`posedge clk`). The input skew is one time step and output skew is two time steps. If you have a time precision of 1ns, this means that you will sample the inputs

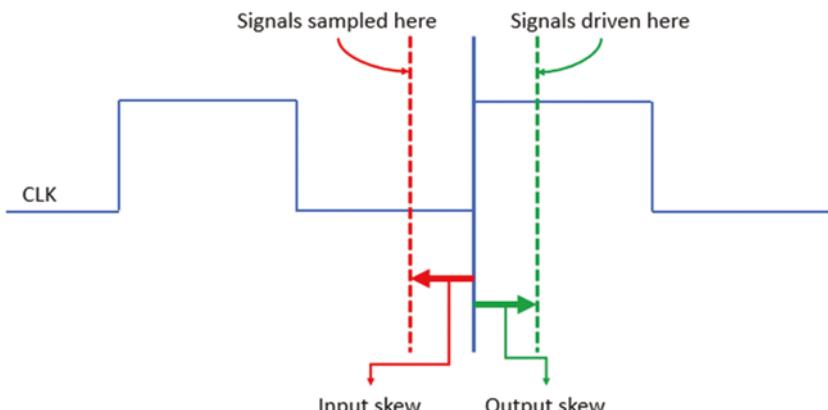


Fig. 19.1 Clocking block: sample and drive signals

1ns prior to the clock edge (setup) and drive signals 2ns after the clock edge (hold). The signal from\_dut, as the name suggests, is the output of the DUT (input to the clocking block) and the signal to\_dut is the input to the DUT (output from the clocking block).

Note that you can also have an edge as skew. For example:

```
clocking ck1 @ (posedge clk);
    default input #1 output negedge; //edge as output skew
    ...
endclocking
```

You can also have hierarchical names and multiple names as inputs and outputs. You can also override default input and output skews. For example:

```
clocking ck1 @ (posedge clk);
    default input #1ns output #2ns;
        input addr, data, enb = PCIe.control.enb; //hierarchical signal
        output posedge CME; //override default output skew
        input #1step ctrl; //override default input skew
endclocking
```

Now, if you think about it, the input skews are implicitly negative, i.e., they always refer to a time *before* the clock. Output skews always refer to a time *after* the clock.

*Note that the default input skew is #1step and the default output skew is #0.* If you do not specify any input or output skews, these default skews will take effect. #1step means that the signal will be sampled just 1 delta before the clocking event. A #1step input skew allows input signals to sample their steady-state values in the time step immediately before the clock event (i.e., in the preceding postponed region). #0 for output skews means that the output will be driven at the *end* of the time unit.

Clocking blocks can be declared inside a module, an interface, a checker, or a program.

You can also declare a signal as “inout.” It signifies two clocking declarations, one input and one output, with the same signal. Such a signal must meet the requirements for both a clocking input and a clocking output.

Let us look at a complete example. I am showing a few different aspects of a clocking block in this example. The example shows how to override default input and output skews, apply “cycle delay” using a default clocking block, use of hierarchical names, etc.:

```
module testBench;
    wire [3:0] address;
    wire [31:0] data;
    wire [3:0] bytes;
    bit clk;
```

```

//Clocking block outputs are DUT (PCI) inputs and vice-versa
default clocking ck1 @ (posedge clk);
    default input #1step output #2;
    input data;
    output bytes;
input #5 irdy = testBench.PCI1.irdy; //override input default
    output negedge address; //override output default
endclocking

initial begin
    //##1 = 1 cycle delay of @ (posedge clk)
    ##1; ck1.bytes <= 10; ck1.address <= 12;

    ##1; ck1.bytes <= 14; ck1.address <= 15; //1 cycle delay
    ##1; ck1.bytes <= 11; ck1.address <= 13; //1 cycle delay
end

initial begin
    $monitor($stime,,,"clk = %b bytes = %0d address = %0d data
= %h irdy = %0d", clk, bytes, address, data, testBench.PCI1.irdy);
    #70 $finish(2);
end

PCI PCI1 (clk, address, data, bytes, irdy); //instantiate PCI
endmodule

module PCI (clk, address, data, bytes, irdy);
    output bit clk;
    output logic [31:0] data;
    input [3:0] bytes;
    input [3:0] address;
    output bit irdy;
    logic [31:0] mem[0:15];

    initial begin
        clk = 0;
        irdy = 0;
    end

    always #10 clk = !clk;

    always @ (posedge clk)
    begin
        mem[address] = $urandom;
        data = mem[address];
    end

```

```

    irdy = !irdy;
end
endmodule

```

*Simulation log:*

```

0  clk = 0 bytes = z address = z data = xxxxxxxx irdy = 0
10 clk = 1 bytes = z address = z data = xxxxxxxx irdy = 1
12 clk = 1 bytes = 10 address = z data = xxxxxxxx irdy = 1
20 clk = 0 bytes = 10 address = 12 data = xxxxxxxx irdy = 1
30 clk = 1 bytes = 10 address = 12 data = c0895e81 irdy = 0
32 clk = 1 bytes = 14 address = 12 data = c0895e81 irdy = 0
40 clk = 0 bytes = 14 address = 15 data = c0895e81 irdy = 0
50 clk = 1 bytes = 14 address = 15 data = 8484d609 irdy = 1
52 clk = 1 bytes = 11 address = 15 data = 8484d609 irdy = 1
60 clk = 0 bytes = 11 address = 13 data = 8484d609 irdy = 1
$finish called from file "testbench.sv", line 24.
$finish at simulation time 70

```

V C S   S i m u l a t i o n   R e p o r t

The module testBench instantiates module “PCI” and drives the inputs and outputs of the system. Module testBench declares a clocking block called “ck1” with the clocking event @ (posedge clk). Here is the clocking block:

```

default clocking ck1 @ (posedge clk);
  default input #1step output #2;
  input data;
  output bytes;
  input #5 irdy = testBench.PCI1.irdy; //override input default
  output negedge address; //override output default
endclocking

```

A few things to note in this clocking block.

First, we have declared it as a “default” clocking block, so that we can use “cycle” delay to drive outputs of the clocking block. A cycle delay is “##n” where “n” is a positive integer (or can be an expression). The “cycle” is the clocking event – which is @ (posedge clk) in our example. This allows an intuitive way of driving signals on a cycle boundary. You do not – have – to declare a clocking block as default, if you do not care to drive outputs with a cycle delay. We will discuss cycle delay in detail a bit later.

Next, the default input skew is “#1step.” This is the default input skew. It means that the signal will be sampled just 1 delta before the clocking event. A #1step input skew allows input signals to sample their steady-state values in the time step immediately before the clock event (i.e., in the preceding postponed region). The default output skew is #2.

Next, we define “irdy” as a hierarchical signal. Yes, you can do that, if the signal is not available in the module in which the clocking block is defined.

Also, we change the input skew of “irdy” to #5, overriding the default input skew of “#1step.” So, you can define skews as default but also override them during signal declaration.

Similarly, we change the default output skew of “address” to “negedge.” This means that “address” output will be delayed until the negedge of the clocking event (negedge clk, in our example).

Note that the signal direction in the clocking block within the module testBench is with respect to the module testBench. In other words, “data” is an output of module PCI and an input to the module testBench.

Next, we drive the clocking block outputs from an “initial” block of module testBench:

```
initial begin
  //##1 cycle delay of @ (posedge clk)
  ##1; ck1.bytes <= 10; ck1.address <= 12;
  ##1; ck1.bytes <= 14; ck1.address <= 15; //1 cycle delay
  ##1; ck1.bytes <= 11; ck1.address <= 13; //1 cycle delay
end
```

As mentioned before, we are using cycle delays to drive the outputs. Each cycle delay is the delay of the clocking event which is @ (posedge clk) in our example. So, the outputs will be driven with a delay of ##1 posedge clk (plus the delays specified with the output signal declaration).

Then “PCI” module drives “data,” “clk,” and “irdy” signals.

*Let us examine the simulation log:*

```
0  clk = 0 bytes = z address = z data = xxxxxxxx irdy = 0
10 clk = 1 bytes = z address = z data = xxxxxxxx irdy = 1
12 clk = 1 bytes = 10 address = z data = xxxxxxxx irdy = 1
20 clk = 0 bytes = 10 address = 12 data = xxxxxxxx irdy = 1
```

Two things to notice here. First, the “bytes” output skew is “#2,” which means that when “bytes” was driven at posedge clk (##1 cycle delay) at time 10, the “bytes” changed at time 12, as expected. Also, the “address” was driven out at the negedge of clk (time 20), since its delay in the clocking block is “negedge” (“output negedge address”).

This pattern continues in the rest of the simulation log. Study carefully to see how the clocking block delays apply to outputs.

One quick note that in our example we have shown only one clock and one clocking block that works off that clock. But most modern designs have multiple clock domains. For that, you can have multiple clocking blocks, one for each clock.

## 19.1 Clocking Blocks with Interfaces

A SystemVerilog interface can contain one or more clocking blocks which can help simplify the interface timing in simulation. The interface signals will be sampled or driven relative to the clocking event of the clocking block. A clocking block encapsulates a set of signals that share a common clock; therefore, specifying a clocking block using a SystemVerilog interface can reduce the amount of code needed to connect the testbench without race condition.

Here is an example of how to use clocking blocks in an interface. For interface discussion, please refer to (Chap. 11).

```
interface mIntf (input clk);
    logic a, b, c, d;
    bit clk;

    clocking mClocking @(posedge clk);
        default input #2 output #3;
        input a,b;
        output c,d;
    endclocking

    clocking sClocking @(posedge clk);
        default input #4 output #6;
        output a,b;
        input c,d;
    endclocking

    modport master (
        input a, b,
        output c, d
    );

    modport slave (
        output a, b,
        input c, d
    );
endinterface

module m1 (mIntf iM);

initial begin
```

```

@(iM.mClocking);
iM.mClocking.c <= 1; //drive clocking block output
iM.mClocking.d <= 0;
end

endmodule

module s1 (mIntf is);

initial begin
@(iS.sClocking);
iS.sClocking.a <= 0; //drive clocking block output
iS.sClocking.b <= 1;
end

endmodule

module top;

reg clk = 0;

always #5 clk=!clk;
initial #20 $finish;

mIntf mI(clk); //Instantiate interface

m1 u1(.iM(mI.master));
s1 u2(.iS(mI.slave));

initial $monitor($stime,,,"clk=%b u2.iS.a = %b",clk,u2.iS.a,
                  " u2.iS.b = %b",u2.iS.b,
                  " u1.iM.c = %b",u1.iM.c,
                  " u1.iM.d = %b",u1.iM.d);

endmodule

```

*Simulation log:*

```
0  clk=0 u2.iS.a = x u2.iS.b = x u1.iM.c = x u1.iM.d = x
```

```

5  clk=1 u2.iS.a = x u2.iS.b = x u1.iM.c = x u1.iM.d = x
8  clk=1 u2.iS.a = x u2.iS.b = x u1.iM.c = 1 u1.iM.d = 0
10 clk=0 u2.iS.a = x u2.iS.b = x u1.iM.c = 1 u1.iM.d = 0
11 clk=0 u2.iS.a = 0 u2.iS.b = 1 u1.iM.c = 1 u1.iM.d = 0
15 clk=1 u2.iS.a = 0 u2.iS.b = 1 u1.iM.c = 1 u1.iM.d = 0

```

In the interface “mIntf,” we define two modports (“master” and “slave”). We then define two clocking blocks. “mClocking” (to provide skews for I/O of modport “master”) and “sClocking” (to provide skews for I/O of modport “slave”). This allows us to embed clocking blocks directly where the directionality of signals is modeled using modports. This is a very modular and powerful way to embed timing for a testbench stimulus/drive signals. Embedding a clocking block in a SystemVerilog interface can significantly reduce the amount of code needed to connect the testbench without race condition.

There are two modules “m1” (that instantiates “mIntf iM”) and “s1” (that instantiates “mIntf iS”). The thing to note in these modules is that you can access the clocking block’s clocking event as well as clocking block’s outputs from these external modules.

One more feature showcased in this example is the clocking event used in modules “master” and “slave.” Here is the code from module “master”:

```

module m1 (mIntf iM);
    initial begin
        @ (iM.mClocking);

```

Notice that here the clocking event is directly using the clocking block name, regardless of the actual clocking event used in the declaration of the clocking block. So, @ (iM.mClocking) is equivalent to @(posedge clk).

Simulation log shows that the outputs “c” and “d” of instance “u1” of module “m1” change after a delay of 3 from the posedge of clk (at time 8). Similarly, outputs “a” and “b” of instance “u2” of module “s1” change after a delay of 6 after the posedge of clk (at time 11).

## 19.2 Global Clocking

In order to specify which clocking event in simulation corresponds to the primary system clock, you can use a global clock. You refer to a global clock using the system function \$global\_clock. For example:

```

module gClock (input logic clk1, clk2, i, output o);
    global clocking gC @ (clk1 or clk2); endclocking
    always @ ($global_clock) o <= i;
endmodule

```

“gC” is defined as the global clock event and will occur when either of the two signals clk1 or clk2 change. You could have referred to the global clock as “`always @ (gC) o <= i;`.” But this is optional since the global clocking event may be referenced simply by using `$global_clock`.

Such a specification may be done for a whole design or for different subsystems of a design. Although more than one global clocking declaration may appear in different parts of a design hierarchy, at most one global clocking declaration is effective at each point in the design hierarchy.

You can also use the global clocking in default clocking block. For example:

```
module m(input logic clk, in, output logic out);
    global clocking @ (posedge clk); endclocking
    default clocking @ ($global_clock); endclocking
    //.....
endmodule
```

Global clocking can be declared in a module, an interface, a program, or a checker. The module, interface, program, or checker can contain at most one global clocking declaration.

When multiple global clocks are defined (e.g., in different modules), the `$global_clock` system function refers to the global clock definition in the scope containing the call to `$global_clock`, and at most one global clocking declaration is effective at each point in the elaborated design hierarchy.

For example:

```
module PCI_master (...);
//...
global clocking @ (posedge master_clock); endclocking
//...
always @ ($global_clock)
//...
endmodule

module PCI_slave (...);
//...
global clocking @ (posedge slave_clock); endclocking
//...
property @ ($global_clock)
//...
endproperty
//...
endmodule
```

The following example is derived from SystemVerilog – LRM. This example shows how global clocks are derived/accessed in a given hierarchy. Note: The

example is not simulated, since the features were not supported by the simulator available to the author!

```

module top;
    subsystem1 sub1();
    subsystem2 sub2();
endmodule

module subsystem1;
    logic subclk1;

    global clocking sub_sys1 @ (subclk1); endclocking
    // ...

    common_sub common();
endmodule

module subsystem2;
    logic subclk2;

    global clocking sub_sys2 @ (subclk2); endclocking
    // ...

    common_sub common();
endmodule

module common_sub;
    always @ ($global_clock) begin
    // ...
end
endmodule

```

Two subsystems are declared: “subsystem1” and “subsystem2.” Each of the subsystem defines a global clock. Each of them also instantiates a “common\_sub” module. So, what happens when module “common\_sub” refers to \$global\_clock? The call to \$global\_clock in top.sub1.common resolves to the clocking event top.sub1.sub\_sys1. The call to \$global\_clock in top.sub2.common resolves to the clocking event top.sub2.sub\_sys2.

A note on global clocking and formal verification. Global clocking is mainly useful in formal verification, i.e., with SystemVerilog Assertions. The verification engineers write assertions for simulation and would like to use the same assertions for formal verification. However, one of the challenges in writing SVA assertions that is compatible across simulation and formal verification is the identification of a primary system clock. That is where the concept of system-level global clock comes into picture. Global clocking facilitates identifying a global system clock. With this,

the user can designate a clock as global clock for the entire system (or a subsystem). The SVA assertions can then refer to this global clock via the system function `$global_clock` instead of a local clock name. This will allow the SVA assertions to be used both for simulation and formal verification.

The past and future global clocking sampled value functions (part of SystemVerilog Assertions) are discussed at length in Sect. 14.24.2.

# Chapter 20

## Checkers



**Introduction** In this chapter we explore the construct of a “checker” which allows you to group several assertions in a bigger block with its well-defined functionality and interfaces providing modularity and reusability. The “checker” is a powerful way to design modular and reusable code. Nested checkers, formal/actual arguments, and checkers in a package are also discussed.

Checkers provide a way to group several assertions together into a bigger block which acts with its well-defined functionality and interfaces providing modularity and reusability. In addition to bundling assertions, you may also put modeling code in these blocks that the assertions or covergroups need. A checker allows you to place all such logic in a well-defined block. One of the intended uses of checkers is to serve as verification library unit.

But wait. Don’t we have “module” and “interface” that does the same thing? Sure, you can have a “module” or “interface” which can keep assertions separate from RTL code and bind them “externally.” But there are significant advantages to keeping assertions grouped into a checker:

1. A checker can be instantiated from a procedural block as well as from outside procedural code as with concurrent assertions. On the other hand, and as we are familiar with, a module cannot be instantiated in a procedural block. It can only be instantiated outside of a procedural block.
2. The formal arguments (ports) of a checker can be sequences, properties, or other edge-sensitive events. Module I/O ports do not allow this.
3. Synthesis tools normally ignore the entire checker block, while in a module, you have to use conditional compile if you have synthesizable code mixed with assertions.

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_20](https://doi.org/10.1007/978-3-030-71319-5_20)) contains supplementary material, which is available to authorized users.

Here is the syntax for a checker. A checker is declared using the keyword *checker* followed by a name and optional port list and ending with the keyword *endchecker*:

```
checker checker_identifier [([checker_port_list])];
  {checker_or_generate_item}
endchecker [: checker_identifier]
```

Let us start with a simple example where we show the advantages of grouping assertions in a *checker* vs. a *module*:

1. Module: First we will define a *module* which holds properties and sequences for a simple bus protocol.
2. Module testbench: Then we will define a testbench that instantiates this *module*.
3. Checker: Then we will see how to put all these properties in a *checker*.
4. Checker testbench: And finally, we will see how the testbench “instantiates” this *checker* from procedural code.

*One:* Assertions in a *module*:

```
module moduleM #(burstSize =4) (dack_, oe_, bMode, bMode_
in, clk, rst);
  input dack_, oe_, bMode , bMode_in, clk, rst;

  sequence data_transfer;
    #2 ((dack_==0) && (oe_==0)) [*burstSize];
  endsequence

  sequence checkbMode;
    (!bMode) throughout data_transfer;
  endsequence

  property pbrule1;
    @ (posedge clk) disable iff (rst) bMode_in |-> checkbMode;
  endproperty

  checkBurst: assert property(pbrule1) else
$display($stime,,, "Burst Rule Violated");
endmodule
```

Module “moduleM” is a simple bus protocol module that is fashioned on the PCI bus. “bMode\_in” needs to be driven low for two clocks (from the testbench) which will then drive bMode\_in of the assertion. Once the bMode\_in is asserted, we need to make sure that “bMode” remains low *throughout* data\_transfer which is four clocks long. We have seen a very similar model in (Sect. 14.18.10) while studying “throughout.” Please refer to the AC specs (timing diagrams) for this *module* in that section. Note that we have parameterized the “burstSize” which is a practical way to model a property that can be reused for different burst lengths.

Now let us see how we instantiate this “moduleM” *module* from our testbench.  
**Two:** Testbench for *module* “moduleM”:

```

module test_moduleM;
logic dataAck_, outputEn_;
logic bMode, bMode_send, rst, clk;

always @ (posedge clk or negedge rst) begin
    if (!rst) begin
        dataAck_=1'b0; outputEn_=0; bMode=0;
    end
end

/*Following block generates a 'bMode' that is Low for 2
clocks consecutively. If so, we send 'bMode_send' to the 'moduleM'
module. */

always @ (posedge clk && rst) begin
    if (!bMode) begin
        @ (posedge clk)
            if (!bMode) bMode_send=bMode;
    end
end

//Now let us instantiate module 'moduleM'

moduleM
    ck1 (.dack_(dataAck_), .oe_(outputEn_), .bMode(bMode), .
bMode_in(bMode_send), .clk(clk), .rst(rst));
endmodule

```

A few things to note here:

1. We had to explicitly create procedural code using an “always” block (to check that “bMode” is low for two consecutive clocks) in behavioral code since we cannot pass sequences to a module. We could have created a “sequence” (in test\_moduleM) to do the same but then you can’t pass a sequence to a module port.
2. We must explicitly pass clk and rst to the module “moduleM” since a module instance will not infer clk or rst from its context. In other words, clk and rst cannot be inferred from the module test\_moduleM.
3. You cannot pass an edge control to a module. Since an edge cannot be passed to a port, you have to make sure that you send the right polarity on these ports (clk for posedge clk) and (!clk for negedge clk).

Now let us model the same “moduleM” *module* as a *checker* “checkerM.”  
*Three:* Assertions in a *checker*:

```

checker checkerM #(burstSize =4) (dack_ , oe_ , bMode, bMode_
in, rst, event clk=$inferred_clock);

input dack_, oe_, bMode ,bMode_in, rst;

sequence data_transfer;
    ##2 ((dack_==0) && (oe_==0)) [*burstSize];
endsequence

sequence checkbMode;
    (!bMode) throughout data_transfer;
endsequence

property prule1;
    @ (posedge clk) disable iff (rst) bMode_in |->
checkbMode;
endproperty

checkBurst: assert property (prule1) else $display($stime,,,
"Burst Rule Violated");

endchecker
```

Note that ‘clk’ is now inferred from the context from which checkerM is instantiated (see the next module test\_checkerM). Also, a sequence ‘bMode\_Sequence’ will be explicitly assigned to port ‘bMode\_in’ from the test\_checkerM module. Neither of these two features are possible if we model our assertions in a Verilog *module*.

Here is the test\_checkerM module that calls the ‘checker checkerM’.

*Four:* Testbench for *checker* ‘checkerM’ (Step Three above)

```

module test_checkerM;
logic dataAck_, outputEn_, bMode;
logic rst, clk;
.....
/*Following block generates a bMode that is Low
for 2 consecutive clocks. */
sequence bMode_Sequence;
    @ (posedge clk) !bMode[*2];
endsequence
.....
```

```

//Now let us call the checker 'checkerM' from a procedural block
always @ (posedge clk or negedge rst) begin
  if (!rst) begin
    dataAck_=1'b0; outputEn_=0;bMode=0;
  end

  else
    checkerM (#8) ck1 (.dack_(dataAck_), .oe_(outputEn_),
.bMode(bMode), .bMode_in(bMode_Sequence) .rst(rst));
  end

endmodule

```

Note:

1. We did not explicitly pass “clk” to the *checker* checkerM. The clk was inferred from the context of the procedural block from which it was called (just as in concurrent assertion that is called from a procedural block). We could have done the same for “rst.”
2. We passed a sequence bMode\_Sequence to checkerM on bMode\_in port.
3. The mechanism for passing input arguments to a checker is similar to the mechanism for passing arguments to a property.
4. We instantiated “checkerM” from the procedural code based on the “rst” condition.

As you noticed, it is much more practical, modular, and easier to code and bundle assertions in a *checker* than in a *module*.

Now let us study further language features and nuances of a “checker.”

Once again, the clock and reset (disable iff) contexts are inherited from the scope of the checker instantiation. Here is another simple example:

```

module test;
  default clocking @ clk; endclocking
  default disable iff reset;

  checker test_bMode;
    //directly inherits @ clk and 'reset' from the higher-
level context of module test
  endchecker

  checker test_cMode; //Note this is a new checker
  //Redefines the default blocks. Point is that you can infer/
  inherit or redefine what is inherited
    default clocking @ clk1; endclocking //Note that the
  default clocking block is for @ clk1

```

```

        default disable iff reset_system; //The default disable
        iff condition is 'reset_system'
endchecker

endmodule

```

The example shows a testbench called “module test” which defines a clk and a reset at “module test” level. The first checker “test\_bMode” inherits the clk and reset from its higher-level scope (which is “module test”). The second checker “test\_cMode” defines its own clk1 and reset\_system. This enables it to have its own local definition of clk1 and reset\_system. It will not inherit the default clk and reset from the top-level module “module test.”

## 20.1 Nested Checkers

As mentioned earlier, a checker can embed another checker, thus making checkers nested. Here is an example that follows the examples above:

```

checker ck1(irdy, trdy, frame_, event clk=$inferred_clock,
event reset = $inferred_disable);

default clocking @ clk; endclocking
default disable iff reset;

property check1;
    irdy |-> ##2 trdy;
endproperty

property check2;
    $rose(irdy) |=> frame_;
endproperty

checker ck2; //nested checker
    property check1; //Redefinition of check1 within the
                      //local scope of checker ck2
        $rose(trdy) |-> irdy;
    endproperty

    property check3;
        $fell(irdy) |-> !frame_;
    endproperty

```

```

checkp1: assert property (check1); //local to checker ck2
checkp3: assert property (check3); //local to checker ck2
checkp2: assert property (check2); //declared in checker ck1
endchecker : ck2

ck2 ck2i ( ); //instantiate ck2

endchecker : ck1

```

Points to note:

1. Checker ck1 properties are visible to checker ck2. Hence checker ck2 is able to “assert” check2 of checker ck1.
2. Checker ck2 redefines property check1 for its local scope use. Since ck2 is instantiated in checker ck1, property check1 of checker ck2 is not directly visible to checker ck1.
3. The inferred clk and reset of checker ck1 are visible to checker ck2.

## 20.2 Checkers: Legal Conditions

A checker body may contain the following elements:

- Declarations of “let” constructs, sequences, properties, and functions.
- Deferred assertions.
- Concurrent assertions.
- Nested checker declarations.
- Other checker instantiations.
- Covergroup declarations and instances.
- “always” (“always\_comb,” “always\_latch,” “always\_ff”), “initial” and “final” procedures. An “initial” procedure in a checker body may contain “let” declarations, immediate, deferred immediate, and concurrent assertions. The procedural timing control statement can be using event control only (edge sensitive).
- The “always” block (and its variations) allows blocking assignments, non-blocking assignments, loop statements, timing even control, subroutine calls, immediate, deferred immediate, concurrent assertions, and “let” declarations.
- “generate” blocks.
- “default clocking” and “default disable iff” statements.
- A formal argument of a checker may be optionally preceded by a direction qualifier: input or output:
  - If no direction is specified explicitly, then the direction of the previous argument is inferred. If the direction of the first checker argument is omitted, it

will default to input. An input checker formal argument cannot be modified by a checker.

- The type of an output argument cannot be of untyped, sequence, or property.
- A checker declaration may also specify an initial value for each singular output port using the same syntax as the default value specification for input arguments.
- A checker declaration may specify a default value for each singular input port.

Checker variable declarations and assignments. Checker variables maybe assigned using blocking and non-blocking procedural assignments or non-procedural continuous assignment.

## 20.3 Checkers: Illegal Conditions

Following is – *not* – allowed in the checker body (this is as far as the author knowledge permits, since the simulators did not fully support checkers as of this writing to validate the following).

A checker body *cannot* contain the following elements (as of writing of this book – IEEE standard is still evolving):

- “if,” “case,” “for,” “continuous assignment,” etc. type of procedural conditional and loop statements are not allowed.
- “initial” block can only contain concurrent or deferred assertions and a procedural event control statement “@.” All other statements are forbidden in the “initial” block.
- Modules, interfaces, programs, and packages cannot be declared inside a checker.
- A checker *cannot* be instantiated in a concurrent procedural construct such as fork..join, fork...join\_any, or fork...join\_none.
- Continuous assignment.
- Declaring *nets* in a checker body is illegal.
- Referencing a checker variable using its hierarchical name in assignments is illegal.

Here are examples showing illegal conditions. (*Disclaimer* – These examples have not been simulated. The simulators available to the author at the time of writing did not fully support checkers):

```
checker myCheck (a, b, c);
  bit myBit;
  ....
endchecker

module myMod;
  ...

```

```

mycheck mck1(a, b, c);
    $display(mck1.myBit); //Hierarchical reference to checker
variable is ILLEGAL
endmodule

```

The following is illegal as well!!

```

checker myCheck(a ,b, c);
logic myBus[7:0], clk;

    always @ (posedge clk) begin
        myBus[1:0] = 2'b0;
        myBus[7:1] = 6'b1;
        // Multiple assignments to the same variable are ILLEGAL.
        // Bit myBus[1] is common and assigned twice.
    end
endchecker

```

BUT the following is legal.

```

checker myCheck(a, b, c);
logic myBus[7:0], clk;

    always @ (posedge clk) begin
        myBus[1:0] = 2'b0;
        myBus[7:2] = 6'b1;
        //Multiple assignments to bits of myBus is assigned
        //only once. LEGAL
    end
endchecker

```

So, with all these restrictions on checker variable assignments, what is one supposed to do? One of the solutions is to use functions, as in the example below:

```

checker myCheck(a, b);
bit a, b;

    initial begin
        @ (posedge clk);
        a = returnAvalue;
    end

    function returnAvalue;
        return a+1;
    endfunction
endchecker

```

In this example, since we cannot assign a value to “a” directly in the “initial” block, we called the function “returnAvalue” to accomplish the same. Note that “a” is visible in the function “returnAvalue.” Since function “returnAvalue” is within the scope of checker mycheck, all the variables available to “mycheck” are also visible to “returnAvalue.” As evident, procedural control statements are allowed in a function.

## 20.4 Checkers: Important Points

1. A checker can be declared in a:

- (a) Module
- (b) Interface
- (c) Program
- (d) Checker (nested checkers)
- (e) Package
- (f) Generate block
- (g) Compilation unit scope

How are variables/arguments in a checker evaluated (i.e., sampled or current)? Note that sampled value means evaluated in the prepended region. Now, this gets a bit tricky.

Except for the variables used in the event control, all other expressions in always\_ff procedures are sampled. This means that the expressions in immediate and deferred assertions instantiated in this procedure are also sampled. However, expressions in always\_comb and always\_latch procedures are *not* implicitly sampled, and the assignments appearing in these procedures use the *current* values of their expressions.

Some examples:

```
module myMod;
    assert #0 (rdy); //This is a deferred assert in a module.
                      //The current value of 'rdy' is used
endmodule

checker myCheck (rdy);
    assert #0 (rdy)  //In the checker, sampled value of 'rdy'
                      //is used.
endchecker

From LRM (LRM, 2002):
checker check(logic a, b, c, clk, rst);
logic x, y, z, v, t;
```

```

assign x = a; // current value of a

always_ff @(posedge clk or negedge rst)
    // current values of clk and rst
begin
    a1: assert (b); // sampled value of b
    if (rst) // current value of rst
        z <= b; // sampled value of b
    else z <= !c; // sampled value of c
end

always_comb begin
    a2: assert (b); // current value of b
    if (a) // current value of a
        v = b; // current value of b
    else v = !b; // current value of b
end

always_latch begin
    a3: assert (b); // current value of b
    if (clk) // current value of clk
        t <= b; // current value of b
end
// ...
endchecker : check

```

2. “type” and “data” declarations within the checker are local to the checker scope and are static.
3. Clock and “disable iff” contexts are inherited from the scope of the checker declaration.
4. You *can* modify/access DUT variables from a “checker”! But my suggestion is to not overdo it. Checker code will not be portable and may result in spaghetti code. Try to keep a checker modular and reusable.
5. Checker formal arguments cannot be of type “local.”
6. Checker formal argument cannot be an “interface.”
7. The connectivity between the actual arguments and formal arguments of a checker follows exactly the same rules as those for modules, namely:
  - (a) Positional association
  - (b) Explicit-named association
  - (c) Implicit-named association
  - (d) Wildcard-named associations
8. A checker body may contain the following elements:

- (a) Declarations of “let” constructs, sequences, properties, and functions.
  - (b) Deferred assertions.
  - (c) Concurrent assertions.
  - (d) Nested checkers are allowed.
  - (e) Covergroup declarations and assignments. One or more “covergroup” declarations or instances are permitted in a checker. These declarations and instances cannot appear in any procedural block in a checker. A “covergroup” may reference any variable visible in its scope, including checker formal arguments and checker variables.
  - (f) Default clocking and disable iff declarations are allowed.
  - (g) Checker output arguments must be typed, and their type cannot be sequence or property.
  - (h) Initial, always, and final procedural blocks are allowed in a “checker” body.
    - (i) An “initial” procedure in a checker body may contain *let* declarations; *immediate*, *deferred*, and *concurrent* assertions; and a procedural timing control statement using an event control only. Similarly, an “always” procedural block also may contain concurrent, deferred assertions, variable assignments, and event control “@.” Nothing else.
    - (ii) The following forms of “always” procedures are allowed in checkers: always\_comb, always\_latch, and always\_ff. Checker “always” procedures may contain the following statements:
      - 1. Blocking assignments
      - 2. Non-blocking assignments
      - 3. Immediate, deferred, and concurrent assertions
      - 4. Loop statements
      - 5. Timing event control
      - 6. Subroutine calls
      - 7. “let” declarations
  - 9. Generate blocks containing any of the above elements are allowed.
  - 10. Variables can be defined in a checker, but *you cannot define a net (wire, tri, etc.)*.
  - 11. Checker variables can be assigned using blocking, non-blocking, or continuous assignments.
  - 12. Checker variables can be “rand” (free variables). Note that the formal arguments and internal variables of a function used in checkers cannot declare “rand” variables. However, free variables are allowed to be passed in as actual arguments to a function.
  - 13. The semantics of “checker” formal arguments is similar to the semantics of “property” arguments. Almost all formal argument types allowed in properties are also allowed in “checkers.”
  - 14. Just as in properties, you can also use inference system functions such as \$inferred\_clock and \$inferred\_disable for checker argument initialization.
- For example:

```
checker checker_args
    (sequence start,
     property end,
     string message = " ",
     event clk = $inferred_clock,
     event rst = $inferred_disable
    );
```

15. The RHS of a checker variable assignment may contain the sequence method .triggered:

```
checker myCheck(a, b, c);
...
sequence busSeq ; ...; endsequence
always @ (posedge clk) begin a <= busSeq.triggered; end
endchecker
```

## 20.5 Checkers: Instantiation Rules

We have already covered nested checker rules. This section provides further guidelines on checker instantiation rules.

As noted, in previous section, a checker can be instantiated anywhere a concurrent assertion can be, except that:

- A checker cannot be instantiated in a procedural construct such as fork...join, fork...join\_any, or fork...join\_none. Checkers cannot contain declarations of modules, interfaces, programs, and packages. *Modules, interfaces, and programs cannot be instantiated inside checkers.*

There is a difference between a checker instantiation and a procedural block or outside. Let us study this via an example:

```
`define limit 256
checker cl1( logic[7:0] a, b);
logic [7:0] add;
always @ (posedge clk) begin
    add <= a + 1'b1;
end

p1: assert property (@ (posedge clk) add < `limit);
p2: assert property (@ (posedge clk) a != b);

endchecker
```

```

module m(input logic rst, clk, logic en, logic[7:0] in1, in2,
in_array [20:0]);

//Concurrent (static) instantiation of 'c1' checker
c1 check_outside(in1, in2);

always @ (posedge clk) begin
    automatic logic [7:0] v1=0;
    if (en) begin
        //Procedural instantiation of 'c1'
        c1 check_inside(in1, v1);
    end
    for (int i = 0; i < 4; i++) begin
        v1 = v1+5;
        if (i != 2) begin

            //Procedural (Loop) instantiation of 'c1'
            c1 check_loop (in1, in_array [v1]);
        end
    end
end
endmodule: m

```

Points to note in the above example:

1. *check\_outside* is a static instantiation, while *check\_inside* and *check\_loop* are procedural. Total of three instantiations of “c1.”
2. Each of the three instantiations of “c1” has its own copy of “add” – which is rather obvious because without it, one instance of “add” would clobber the “add” of another instance. This copy of “add” is updated at every positive clock edge, regardless of whether that instance was visited in procedural code. Even in the case of *check\_loop*, there is only one instance of “add,” and it will be updated using the sampled value of “in1.”
3. For checker instance “check\_outside,” “p1,” and “p2” are checked at every posedge clock. For checker instance “check\_inside,” “p1,” and “p2” are queued for evaluation anytime ‘en’ is true (on posedge clk).
4. For *check\_loop*, three procedural instances of “p1” and ‘p2’ are queued (for  $i = 0,1,3$ ), and they will evaluate at every posedge clk.
5. Since “c1” (*check\_outside*) instance is static (concurrent), the assertion statements in “checker c1” are continually monitored and begin execution on any time step when their sampling edge (clock event) occurs.

## 20.6 Checkers: Rules for “formal” and “actual” Arguments

The mechanism for passing actual arguments to the formal arguments of a checker is the same as that for passing actual arguments to a “property.” It is important to note that it’s the “sampled” value (i.e., the value in the prepended region) of an actual that is assigned to the formal of the checker (this rule is also the same as that for a property).

Here are the rules for “formal” and “actual” of a checker and checker instantiation. Again, they are similar to those applied to a “property” or a “sequence.” But some are repeated here for the sake of completeness:

- A “formal” argument of a checker can be optionally preceded by a direction qualifier: “input” or “output.”
- If no direction is specified explicitly, then the direction of the previous argument will be inferred.
- If the direction of the first checker argument is omitted, it will default to “input.”
- Obviously, an “input” checker formal argument cannot be modified by a checker.
- The legal data types of a checker formal argument are the same as those legal for a property.
- The type of an “output” argument cannot be of type “untyped,” “sequence,” or “property.”
- You cannot omit the type of a formal argument, if you have assigned an explicit direction qualifier.
- If you do omit the type of a checker formal argument and if it’s the first argument of the checker, then it will be assumed to be “input untyped.”
- If you do omit the type of a checker formal argument and it is *not* the first argument, then the type of the “previous” formal argument will be inferred.
- A checker declaration may specify a “default” value for each singular input.
- A checker declaration may also specify an initial value for each of its singular output using the same syntax as the default value specification for input arguments.
- As modules, checkers may access elements from their enclosing scope through their hierarchical names, except the following:
  - Automatic and dynamic variables
  - Elements of fork...join blocks (including join\_any and join\_none)

Please note: Checkers for formal verification are not covered in this book since it is beyond the scope of the book. Specifically, “assume property” is not explored beyond its context in simulation.

## 20.7 Checkers: In a Package

For modularity and reusability, it is good to keep checkers (or properties or sequences for that matter) in SystemVerilog “package.” In addition, this will keep the scope of the checker to local scope of the package and will avoid name conflicts if someone named their checker (or module for that matter) the same as your checker.

Let us look at a simple example:

```
package project_library;
    checker irdy_trdy (irdy, trdy, event clk =
$inferred_clock);
    a1: assert property (@clk) irdy |=> trdy;
endchecker : irdy_trdy
endpackage: project_library
```

As in SystemVerilog, if you want to instantiate this checker from a module, you will have to import this package:

```
module PCI(irdy, trdy, clk);
    import project_library::*;
        irdy_trdy (irdy, trdy, posedge clk)
    endmodule : PCI
```

Points to note in this example:

- The statement `project_library::*` makes the entire content of the package visible to module PCI.
- You could have imported only the checker of interest rather than importing the entire package, as in:

```
import project_library::irdy_trdy;
```

# Chapter 21

## “let” Declarations



**Introduction** This chapter delves into the detail of “let” declaration. “let” declarations have local scope in contrast with `define which has global scope. A ‘let’ declaration defines a template expression (a let body), customized by its ports (a.k.a. parameters). “let” with parameters and “let” in immediate and concurrent assertions are also discussed.

We have all used the compiler directive `define (as a global text substitution macro). Note the word *global* – it is truly global spanning across *all* scopes of your design modules and files. For example, `define intr 3'b111 will substitute `intr with 3'b111 wherever it sees `intr either within the local scope or global scope. This can be good and bad. Good is that you have to define it only once and it will span across module/file boundaries. Bad is you cannot redefine `intr (well actually you can but with consequences). For example, if you change the definition of `intr in a package, you will get a warning, and also the new definition will overwrite all the previous ones.

That is where “let” comes into picture. “let” allows local scope and allows parameterization (or as LRM puts it, it has “ports”) (as in a sequence or a property).

To reiterate, “let” declarations can be used for customization and can replace the text macros in many cases. The “let” construct is safer because it has a local scope, while the scope of compiler directives is global within the compilation unit. A “let” declaration defines a template expression (a let body), customized by its ports (a.k.a. parameters). A “let” construct may be instantiated in other expressions.

The syntax for “let” is:

```
let_declaration ::= let let_identifier [ ( [let_port_list] ) =  
expression;
```

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_21](https://doi.org/10.1007/978-3-030-71319-5_21)) contains supplementary material, which is available to authorized users.

## 21.1 “let”: Local Scope

First let us see an example using “let”:

```
module example;
  logic r1,r2, r3,r4,clk,clk1;
  let exDefLet = r1 || r2;
  always @ (posedge clk) begin: ablock
    let exDefLet = r1 & r2;
    //exDefLet has a local scope of 'ablock'
    r3=exDefLet;
  end
  always @ (posedge clk1) begin: bblock
    r4=exDefLet;
    // exDefLet will take the definition from the scope that is
    // visible to it. Here it is the outer most scope
    // definition of (r1 || r2);
  end
endmodule
```

We have defined “exDefLet” in two different scopes. One in the always block “ablock” and another at the outermost scope “module example.” Note that their definition (expression) is different in each block. Since “let” can have local scope, each of the definition of “let” will be preserved in its local block. The above code will look like the following after “let” substitutions take place:

```
module example;
  logic r1,r2, r3,r4,clk,clk1;
  always @ (posedge clk) begin :ablock
    r3=r1 & r2;
  end
  always @ (posedge clk1) begin: bblock
    r4=r1 || r2 ;
  end
endmodule
```

If the same design was modeled using `define, here’s how the code would look like:

```
module example;
  logic r1,r2, r3,r4,clk,clk1;
  `define exDefLet r1 || r2;
  always @ (posedge clk) begin :ablock
    `define exDefLet r1 & r2;
    r3=`exDefLet;
  end
  always @ (posedge clk1) begin: bblock
    r4=`exDefLet;
  end
endmodule
```

In this example, since there are two `define for the same variable, the compiler will complain right off the bat and use the second definition r1 & r2 (it is the latest in lexical order) as the global definition of exDefLet. The above code will look like the following after `define substitutions. Both r3 & r4 will take on the latest `define value of `exDefLet, which is r1 & r2:

```
module example;
  logic r1,r2, r3,r4,clk,clk1;
  always @ (posedge clk) begin: ablock
    r3 = r1 & r2;
  end
  always @ (posedge clk1) begin: bblock
    r4 = r1 & r2;
  end
endmodule
```

As you see “let” is very useful from scoping point of view. It follows the normal scoping rules. You can parameterize it (i.e., it can have “ports”) and reuse the “let” expression repeatedly with different parameters.

## 21.2 “let”: With Parameters

As mentioned before, “let” can be parameterized. Note that instantiation of “let” is quite different from a “parameterized function.” With “let” you replace the instance with *entire* “let” body. With “function” you simply pass the parameters, and the function executes using those parameters. Function does not replace the instance of function call.

Ok, let us see a simple example:

```

module abc;
  logic clk, x, y, j;
  logic [7:0] r1;
  let lxor (p, q=1'b0) = p^q;
  always @ (posedge clk) begin
    for (i = 0; i <= 256; i++) begin
      r1 = lxor( i );
      //After expanding the 'let' instance,
      //this will be r1 = i ^ 1'b0;
    end
  end
endmodule

```

For each value of “i” r1 will get the “xor” of “i” and “q = 1’b0.” Note that the formal parameter “q” is assigned a default value of “1’b0.” That being the case, when “r1 = lxor(i)” is executed, the actual “i” replaces the formal “p” in lxor, and “q” takes on its assigned default value of “1’b0.” You could have also specified “r1 = lxor(i, j),” and the formal “q” will now take the value of “j.”

Some rules apply to the formal arguments:

1. Note again that the “let” body gets expanded with the actual arguments (which replace the formal arguments) and the body (RHS of “let”) will replace the instance of “let.” That being the case, once the body of “let” replaces the instance of “let,” all required semantic checks will take place to see that the expanded “let” body with the actual arguments is legal.
2. The formal arguments can have a default value (as we saw in the example above).
3. The formal arguments can be typed or untyped. The typed arguments will force type compatibility between formal and actual (cast compatibility). In other words, the actual argument will be cast to the type of the formal argument before being substituted. Untyped formal in that case is more flexible.
4. If the formal argument is of “event” type, then the actual argument must be an event\_expression. Each reference to the formal argument will be in a place where an event\_expression may be written.
5. The self-determined result type of the actual argument must be cast compatible with the type of the formal argument. The actual argument must be cast to the type of the formal argument before being substituted for a reference to the formal argument.
6. If the variables used in “let” are not formal arguments to the “let” declaration, they will be resolved according to the scoping rules of the scope in which “let” is declared.

## 21.3 “let”: In Immediate and Concurrent Assertions

Yes, “let” can be used in an immediate (“assert”) as well as concurrent (“assert property”) assertions in a procedural block.

Let us start with a very simple example of “let” usage in a sequence. *Note that “let” expression can only be structural or with sampled value function (as in \$past).* We will see “let” with sampled value function in the next section:

```
module abc;
  logic req, gnt;
  let reqack = !req && gnt;
  sequence reqGnt;
    reqack;
  endsequence
endmodule
```

After expanding the “let” instance:

```
module abc;
  logic req, gnt;
  sequence reqGnt;
    !req && gnt;
  endsequence
endmodule
```

Here’s another example:

```
module abc;
  logic clk, r1,r2,req,gnt;
  let xxory (x, y) = x ^ y; //bit wise xor
  let rorg = req || gnt;
  ....
  //concurrent assertion
  P1: assert property (@ (posedge clk) (rorg));
  always_comb begin
    a1: assert (xxory (r1,r2));      //immediate assertion
    a2: assert (rorg);
  end
endmodule
```

After expansion:

```
module abc;
  logic clk, r1,r2,req,gnt;
  P1: assert property (@ (posedge clk) (req || gnt));
  always_comb begin
    a1: assert (r1 ^ r2);      //immediate assertion
    a2: assert (req || gnt);
  end
endmodule
```

Now, here is an example that uses the sampled value functions \$(rose) and \$(fell):

```
module abc;
  logic clk,r1,r2,req,gnt,ack,start;
  let arose(x) = $rose( x );
  let afell(y) = $fell ( y );
  always_comb begin
    if (ack) s1: assert(arose(gnt));
    if (start) s2: assert(afell(req));
  end
```

Another intended use of let is to provide shortcuts for identifiers or subexpressions. For example:

```
task write_value;
  input logic [31:0] addr;
  input logic [31:0] value;
  ...
endtask
...
let addr = top.block1.unit1.base + top.block1.unit2.displ;
...
write_value(addr, 0);
```

But note that hierarchical references to “let” expressions are *not* allowed. For example, the following is illegal. Assuming “my\_let” to be a *let* declaration, the following is illegal:

```
assign e = Top.CPU.my_let(a)); // ILLEGAL
```

Also, Recursive ‘let’ instantiations are not permitted.

Here is an example of how the “let” arguments bind in the declarative context:

```

module sys;
logic req = 1'b1;
logic a, b;
let y = req;
...
always_comb begin
    req = 1'b0;
    b = a | y;
end
endmodule: sys

```

The effective code after expansion:

```

module sys;
logic req = 1'b1;
logic a, b;
...
always_comb begin
    req = 1'b0;
    b = a | (sys.req);
    //NOTE: y binds to preceding definition of 'req' in the
    //declarative context of 'let'
end
endmodule : top

```

Similarly, here's another example:

```

module CPU;
logic snoop, cache;
let data = snoop || cache;
sequence s;
    data ##1 cache
endsequence : s
...
endmodule : top

```

After expansion:

```

module CPU;
logic snoop, cache;
sequence s;
    (CPU.snoop || CPU.cache) ##1 cache;
endsequence : s
...
endmodule : top

```

The following is an example of “let” with typed formal arguments. The example shows how type conversion works when the type of a formal is different from the type of an actual.

First, module “m” with “let” declarations. Note the typed formals in “let” declarations and the mismatch between “let” instance “actuals” and “formals”:

```
module m(input clock);
logic [15:0] a, b;
logic c, d;
typedef bit [15:0] bits;
...
let ones_match(bits x, y) = x == y;
let same(logic x, y) = x === y;

always_comb
a1: assert(ones_match(a, b));
//Note: the actuals 'a' and 'b' are of type 'logic', while the
//formals 'x', 'y' are of type 'bits'

property toggles(bit x, y);
    same(x, y) |=> !same(x, y);
//Note: the actuals 'x', 'y' are of type 'bit', while the
//formals 'x', 'y' are of type 'logic'
endproperty

a2: assert property (@(posedge clock) toggles(c, d));
endmodule : m
```

After expansion:

```
module m(input clock);
logic [15:0] a, b;
logic c, d;
typedef bit [15:0] bits;
...
// let ones_match(bits x, y) = x == y;
// let same(logic x, y) = x === y;

always_comb
a1: assert((bits'(a) == bits'(b)));

property toggles(bit x, y);
    (logic'(x) === logic'(y)) |=> !(logic'(x) == logic'(y));
endproperty
```

```
a2: assert property (@(posedge clock) toggles(c, d));  
endmodule : m
```

Finally, here's where a “let” can be declared:

- A module
- An interface
- A program
- A checker
- A clocking block
- A package
- A compilation unit scope
- A generate block
- A sequential or parallel block
- A subroutine

# Chapter 22

## Tasks and Functions



**Introduction** This chapter discusses SystemVerilog “tasks” and “functions,” including static/automatic tasks and functions, parameterized tasks/functions, etc. Argument passing and argument binding are also discussed.

Tasks and functions are building blocks of design and verification logic. They allow for modular and reusable development of code. They provide the ability to execute common procedures from several places in your logic. You can parameterize tasks and functions so that they can be invoked from different places in your code with different parameters.

The fundamental difference between a task and a function is that a function is zero time executing block, while a task may contain time-consuming statements. Also, a function cannot enable a task, but a task can enable other tasks or functions. As you know, a non-void function returns a value, while a void function does not return a value. A task can have outputs that can be assigned within the task, but the task itself does not return a value. And a non-void function can be used in an expression as an operand, where the value of the operand is the value returned by the function.

As mentioned above, a function returns a value, while a task can have an output that will return a value. For example:

```
//task that returns an output value  
valueT (input int iT, output int oT);  
  
//function that returns a value  
oT = valueT (iT);
```

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_22](https://doi.org/10.1007/978-3-030-71319-5_22)) contains supplementary material, which is available to authorized users.

## 22.1 Tasks

Since a task can consume time, once you invoke (call) it, the control (end of task) will be passed back from where it was called, *after* the task has completed. That can be the same time when the task was called or a later time.

Multiple statements can be written between the task declaration and endtask. Statements are executed sequentially, the same as if they were enclosed in a begin...end group.

A task can enable other tasks, which in turn can enable furthermore tasks.

Variables declared inside a task are local to that task. They can drive global variables external to the task.

Tasks are static by default. They can be declared automatic (discussed below).

But note that a task is automatic by default within a “class.” Lifetime of “class” methods are always automatic.

Tasks can use blocking and non-blocking assignments.

You can declare a “module” also as automatic, in which case all the tasks in the module would be considered automatic by default so that you do not have to add the automatic keyword after each function or task. Syntax for declaring a module automatic is “module *automatic* <module\_name>.”

The syntax of a task in its simplest form is:

```
task [static | automatic] (<task port list>
    <task body>
endtask
```

For example:

```
task PCI_read (inout int data, output int address, input logic
bMode, iReady);
    ...
endtask
```

OR

```
task PCI_read;
    inout int data;
    output int address;
    input logic bMode, iReady;
    ...
endtask
```

Here is an example of the usage of a task. Key to note is how the task blocks when it has time-consuming elements in it:

```

module mTask( );

logic iReady;
int data, address;

task PCI_read ( inout int data, output int address);

@(posedge iReady); //edge sensitive time control
$display($stime,,,"`t In task - posedge iReady");

address = $urandom; data = $urandom;
$display($stime,,,"`t address=%h data=%h", address, data);

@(negedge iReady); //edge sensitive time control
$display($stime,,,"`t In task - negedge iReady");

address = $urandom; data = $urandom;
$display($stime,,,"`t address=%h data=%h", address, data);

endtask

initial begin
    iReady = 0;
    #100 $finish(2);
end

always begin
    #20; $display($stime,, "CALL PCI_read");
    PCI_read (data, address); //blocks till task is over
end

always begin
    #10; iReady = ! iReady;
end
endmodule

```

*Simulation log:*

```

20  CALL PCI_read
30      In task - posedge iReady
30      address=12153524 data=c0895e81
40      In task - negedge iReady

```

```

40      address=8484d609 data=b1f05663
60  CALL PCI_read
70      In task - posedge iReady
70      address=06b97b0d data=46df998d
80      In task - negedge iReady
80      address=b2c28465 data=89375212
$finish called from file "testbench.sv", line 50.
$finish at simulation time      100
    V C S   S i m u l a t i o n   R e p o r t

```

A task *PCI\_read* is defined which takes “data” and “address” as inout and output ports. In the task, we consume time using event control – posedge iReady and negedge iReady. This means that when we enter the task, we will wait on these events before we exit the task.

We then call “*PCI\_read*” from an “always” block which executes every 20 time unit. So, you would think that “*PCI\_read*” will be called every 20 time unit. NO. First time we will call “*PCI\_read*” at time 20. Once the task is invoked, it will wait for posedge iReady and then negedge iReady. In other words, the task will wait until these two events are completed. So, the task will block. Only when it is over at time 40 that the next invocation of “*PCI\_read*” will take place – which is at time 60. This proves that a task, when it consumes time, will block the code that follows its invocation, until the task is over. Study the log carefully and understand this point.

### 22.1.1 Static and Automatic Tasks

As noted above, tasks (outside a “class”) are *static* by default. Static means the memory is allocated once for all the variables of the task (there is no call stack for arguments or variables local to the routines) and shared for each invocation of the task. This means you cannot have recursive or re-entrant routines. Automatic allows a task to be re-entrant. A re-entrant task is one in which the variables of the task are allocated memory upon each individual call of the task. Static tasks are a problem if the task gets called simultaneously from two different places. The variables of the task for each call will clobber the variables of the other invocation. They will write over each other. Automatic task will maintain a separate variable space for each invocation of the task.

A task may be enabled more than once concurrently. All variables of an automatic task will be replicated on each concurrent task invocation to store state specific to that invocation. All variables of a static task will be static in that there will be a single variable corresponding to each declared local variable in a module instance, regardless of the number of concurrent activations of the task.

Here is an example, where a task is declared “automatic.” We will see how the task variables behave when the task is called simultaneously from a fork...join. Later, we will see how the same example works when the task is static:

```
module mTask( );
    int data, address;

    task automatic PCI_read ( inout int data, input int
address);

#10;
    address = address + 1; data = data + 1;
$display($stime,,,"\\t address=%h data=%0d",address,data);

#10;
    address = address + 1; data = data + 1;
$display($stime,,,"\\t address=%h data=%0d",address,data);

#10;
    address = address + 1; data = data + 1;
$display($stime,,,"\\t address=%h data=%0d",address,data);

#10;
    address = address + 1; data = data + 1;
$display($stime,,,"\\t address=%h data=%0d",address,data);

endtask

initial begin
    address = 0; data = 9;
    #100 $finish(2);
end

always begin

fork
begin
    #20; $display($stime,,,"CALL PCI_read");
    PCI_read (data, address); //blocks till task is over
end
begin
    #1; $display($stime,,,"CALL PCI_read");
    PCI_read (data, address); //blocks till task is over
end
join
end
```

```
endmodule
```

*Simulation log:*

```

1 CALL PCI_read
11   address=00000001 data=10
20 CALL PCI_read
21   address=00000002 data=11
30   address=00000001 data=10
31   address=00000003 data=12
40   address=00000002 data=11
41   address=00000004 data=13
50   address=00000003 data=12
60   address=00000004 data=13
61 CALL PCI_read
71   address=00000001 data=14
80 CALL PCI_read
81   address=00000002 data=15
90   address=00000001 data=14
91   address=00000003 data=16
$finish called from file "testbench.sv", line 26.
$finish at simulation time      100

```

V C S S i m u l a t i o n R e p o r t

We declare the task “PCI\_read” as an “automatic” task. We fork off invocation of this task from an “always” block where one branch of the fork...join calls the task at time 1, while the other branch calls the task at time 20. Note that both invocations are concurrent (because of fork...join).

The invocation at time 1 enters the task at time 1 and then goes through the task with an increment of #10 time units. The invocation in parallel at time 20 also go through the task with increments of #10 time units.

Since the task is declared automatic, the I/O variables of the task are automatic, meaning these variables of the task are allocated memory upon each individual call of the task. This is evident from the simulation log:

```

1 CALL PCI_read
11   address=00000001 data=10
20 CALL PCI_read
21   address=00000002 data=11
30   address=00000001 data=10
31   address=00000003 data=12
40   address=00000002 data=11
41   address=00000004 data=13
50   address=00000003 data=12
60   address=00000004 data=13

```

First call to PCI\_read is at time 1 and the parallel call is at time 20. The call at time 1 goes through the task at time increments of #10. At each timestamp, the variables “address” and “data” increment (at times 11, 21, 31, 41). Similarly, call at time 20 goes through #10 time stamps (at times 20, 30, 40, 50). Each of this call has a separate copy of “address” and “data,” and we see that they increment independently of each other.

Now, let us change only the following line in the above example.

Change:

```
task automatic PCI_read (inout int data, input int address);
```

to

```
task PCI_read (inout int data, input int address); //  
static task
```

Let us see how the simulation log changes:

```
1 CALL PCI_read  
11    address=00000001 data=10  
20 CALL PCI_read  
21    address=00000001 data=10  
30    address=00000002 data=11  
31    address=00000003 data=12  
40    address=00000004 data=13  
41    address=00000005 data=14  
50    address=00000006 data=15  
60    address=00000007 data=16  
61 CALL PCI_read  
71    address=00000001 data=17  
80 CALL PCI_read  
81    address=00000001 data=17  
90    address=00000002 data=18  
91    address=00000003 data=19  
$finish called from file "testbench.sv", line 26.  
$finish at simulation time      100
```

### V C S   S i m u l a t i o n   R e p o r t

In this case, since the task is not declared “automatic,” the task is “static” by default. So, its variables are static, meaning the memory is allocated once for all the variables of the task and shared for each invocation of the task. No separate copies of “address” and “data” are maintained. So, in the simulation log, you see that for both calls to PCI\_read, the “address” and “data” increment linearly – each invocation does not have a separate copy of the variables. The variables in the static task share the same memory. In other words, the variables for the two invocations

overwrite each other. So, in the simulation log, you see that the “address” and “data” variables increment linearly.

But note the following in the simulation log:

```
1 CALL PCI_read
11    address=00000001 data=10
20 CALL PCI_read
21    address=00000001 data=10
```

The address and data do not increment from time 11 to time 21. Here is what is going on.

The first call to PCI\_read is at time 1, so after #10, at time 11, time address = 00000001 and data = 10. Then the task waits for #10 to increment address and data at time 21. But a new (parallel) invocation of the task comes at time 20. At the same time, the first invocation is waiting for #10 to increment address and data at time 21. But since the parallel invocation came in at 20, it will clobber the first increment of address and data and reset those to the values address = 00000001 and data = 10. Hence, address and data do not increment at time 21. This is the issue with concurrent invocation of a task that is “static.” Study the log carefully to see what is going on.

Here is another example:

```
module test();

task automatic add (int a, int b);
#2;
$display($stime,,, "the sum is %0d", a + b);
endtask

initial
fork
begin
$display($stime,,,"`t call add(2,3)");
add(2,3);
end
begin
#1;
$display($stime,,,"`t call add(3,4)");
add(3,4);
end
join
endmodule
```

*Simulation log:*

```

0    call add(2,3)
1    call add(3,4)
2  the sum is 5
3  the sum is 7
V C S   S i m u l a t i o n   R e p o r t

```

*Exercise: Can you figure out the simulation log?* Two calls are made to “add” one at time 0 and another at time 1. Since the task is automatic, the variables “a” and “b” maintain their own memory. Rest, please figure out.

Next, we make the following change.

Change:

```

task automatic add(int a, int b);
to

```

```
task add(int a, int b);
```

Simulation log with this change:

```

0    call add(2,3)
1    call add(3,4)
2  the sum is 7
3  the sum is 7
V C S   S i m u l a t i o n   R e p o r t

```

*Exercise: Can you figure out the simulation log? Note that “a” and “b” are now static and share the same memory.*

Note that you can declare individual variables inside a task as static or automatic. In other words, in an automatic task, you can declare a variable as “static.” Or in a “static” task, you can declare a variable as “automatic.” For example:

```

task static PCI_read ( );
//...
automatic int i3;
endtask

task automatic PCI_write ( );
//...
static int i2;
endtask

```

Finally, here are some things not allowed in a task. This is mainly because the variables declared in automatic tasks are deallocated at the end of the task invocation. So, they cannot be used in certain constructs that might refer to them after the deallocation:

- These variables cannot be assigned values using non-blocking assignments.
- These variables cannot be referenced by procedural continuous assignment or procedural “force” statements.
- These variables cannot be referenced in intra-assignment event controls of non-blocking assignments.
- These variables cannot be traced with system tasks such as \$monitor.

## 22.2 Functions

As mentioned before, a function is a zero time (single time step) executing construct. Unlike tasks, functions have the restrictions that make certain that they return without suspending the process that enables them. Hence, a function cannot contain any time-consuming statements. From that point of view, a function *cannot* have the following operators:

#, ##, @, fork..join, fork..join\_any, wait, wait\_order or expect.

But statements that do *not* block are allowed. For example, non-blocking assignments, named event trigger, fork\_join\_none, are allowed.

Since time control is not allowed in a function, it cannot enable a task regardless of whether those tasks contain time-controlling statements.

A function’s primary purpose is to return a value that is to be used in an expression.

Here is the syntax in its simplest form:

```
function <automatic | static> <data_type_or_implicit_or_void>
function_name
    function_body
endfunction
```

The return type can be an explicit data\_type or “void.”

For example:

```
function logic [2:0] getStatus (int a, int b);
    ...
endfunction
```

This can also be written as:

```
function logic [2:0] getStatus;
    input int a;
    input int b;
    ...
endfunction
```

From this, we see that the function declarations default to the formal direction “input.” Once a direction is given, the subsequent formals default to the same direction. For example:

```
function logic [2:0] getParity (int a, int b, output logic [2:0] u, v);
```

In this example, “a” and “b” are inputs while “u” and “v” are outputs. Such formal arguments are the same as those for tasks. “input” is the copy value input at the beginning; “output” is the copy value at the end of the function; “inout” is the copy input at the beginning and output at the end; and “ref” to pass reference.

Each formal argument has a data type that can be explicitly specified or inherited from the previous argument. The default data type is “logic.”

Multiple statements can be written in a function (between function header and endfunction). Statements are executed sequentially just as in a begin...end block. Or you may have no statements at all; in which case the function returns the current value of the implicit variable that has the same name as the function.

### **22.2.1 Function Called as a Statement**

Here is an example of function called as a statement:

```
module mTask();
    int data, address;

    function int func1 ( inout int dataIO, input int addressIN );
        addressIN = addressIN + 1; dataIO = dataIO + 1;
        $display($stime,,,"t addressIN = %h dataIO=%0d",addressIN,dataIO);
    endfunction

    initial begin
        address = 0; data = 9;

        #1; $display($stime,,,"CALL func1");
        void '(func1 ( data, address));
            //function call as a statement
    end
endmodule
```

*Simulation log:*

1 CALL func1

```
1      addressIN=00000001 dataIO=10
V C S   S i m u l a t i o n   R e p o r t
```

Functions that return a value can be used in an assignment or as expression. Note that in the function call “func1” as a statement, if you do not declare it as “void,” you will get the following warning. So, if you called it as follows:

```
func1 ( data,  address); //function call as a statement
```

you will get the following warning:

Warning-[SV-NFIVC] Non-void Function Used In Void Context  
testbench.sv, 19  
mTask, "func1()"

The non-void function 'func1' is used as a void function. Hence the value returned by the function will be ignored.

Please cast the function call with void in case the value returned by the function is not needed.

In other words, “func1” does not return a value; it simply assigns a value to its IO. Hence, “void(func1 (data, address));” is needed.

### **22.2.2   Function Name or “return” to Return a Value**

Here is an example of how a function name can be used to return a value (or how to use a “return” statement to return a value from the function):

```
module mFunc( );
    int data, address;
    int newval;
    int add;

    function int func1 ( inout int dataIO, input int addressIN );
        func1 = addressIN + dataIO;
        //return value assigned to function name

        //OR

        //return addressIN + dataIO;
        //return value specified using 'return' statement
    endfunction

    initial begin
```

```

address = 1; data = 9; add= 10;

#1;
newval = add + func1 ( data, address);
$display($stime,,, "newval = %0d",newval);

end
endmodule

```

*Simulation log:*

```

1 newval = 20
VCS Simulation Report

```

You can return a value from the function by using the function name or explicitly calling a “return” statement.

### 22.2.3 Void Functions

Void functions do not return a value. They simply execute statements within the function body. For example:

```

module mVoid( );
    function void func1;
        $display ("Time = %0t",$stime);
    endfunction

    initial begin
        for (int i = 0; i < 4; i++) begin
            #1;
            func1 ( );
        end
    end
endmodule

```

The function “func1” is void and does not return a value. It simply executes statements in its body. Here we are simply displaying time.

*Simulation log:*

```

Time = 1
Time = 2
Time = 3
Time = 4
VCS Simulation Report

```

## 22.2.4 Static and Automatic Functions

Just like “automatic” tasks, you can also have “automatic” functions. The rules and working of “automatic” vs. “static” are pretty much the same as that for tasks. Automatic functions allocate unique, stacked storage for each function call. *By default, functions are “static.”* You have to use the keyword “automatic” with function declaration to make it automatic. Note that, just as with tasks, *functions defined within a class are always automatic.* Static means the memory is allocated once for all the variables of the function (there is no call stack for arguments or variables local to the routines) and shared for each invocation of the function. This means you cannot have recursive or re-entrant routines with static functions. Automatic allows a function to be re-entrant. A re-entrant function is one in which the variables of the function are allocated memory upon each individual call of the function.

You may have local variables declared as “automatic” within a “static” function or declared as “static” in an “automatic” function.

Here is an example of “automatic” function (SystemVerilog – LRM):

```
module funcAuto;
    integer result;

        function automatic integer factorial (input [31:0]
operand);
            if (operand > 1)
                factorial = factorial (operand - 1) * operand;
            else
                factorial = 1;
        endfunction: factorial

        initial begin
            for (int n = 0; n <= 7; n++) begin
                result = factorial(n);
                $display("%0d factorial=%0d", n, result);
            end
        end
    endmodule
```

*Simulation log:*

```
0 factorial=1
1 factorial=1
2 factorial=2
3 factorial=6
4 factorial=24
5 factorial=120
6 factorial=720
```

```
7 factorial=5040
```

### V C S S i m u l a t i o n R e p o r t

One advantage (of many) that automatic storage class enables is *recursive functions*. That is what we have in this example (

```
" factorial = factorial (operand - 1) * operand; "
```

). In our example, the re-entrant function dynamically makes multiple recursive instantiations of itself. Each instance gets its “automatic” variables mapped on the stack. As we progress into the recursion, the stack grows, and each function gets to make its computations using its own set of variables. When the function calls return, the computed values are collated, and a final result made available. With only static variables, each function call will store the variable values at the same common address, thus erasing any benefit of having multiple calls (instantiations).

As explained by Goel, with automatic storage class, each recursive function call would have its own place in the memory (on the stack) to store variable **n**. As a result, consecutive calls to factorial will not overwrite variable **n** of the caller. As a result, when the recursion unwinds, we will have the right value for factorial(n) as  $n*(n-1)(n-2) \dots *1$ . Hence, the factorial results show the value of factorial as non-overridden value (see simulation log).

Let us take the same example but change the “automatic” function as “static” (which is the default). So, we will change the following line:

```
function automatic integer factorial (input [31:0] operand);
```

to

```
function integer factorial (input [31:0] operand);
```

Here is the simulation log we get with this change:

```
0 factorial=1
1 factorial=1
2 factorial=1
3 factorial=1
4 factorial=1
5 factorial=1
6 factorial=1
7 factorial=1
```

### V C S S i m u l a t i o n R e p o r t

As explained by Goel, with the function “factorial” defined as static, since all variables in a function would be mapped to a fixed location, when we call factorial(n-1) from inside factorial(n), the recursive call would overwrite any variable inside the caller context. In the factorial function as defined in the above code

snippet, if we do not specify the storage class as automatic, both n and the result factorial would be overwritten by the recursive call to factorial(n-1). As a result, the variable “n” would consecutively be overwritten as n-1, n-2, n-3, and so on till we reach the terminating condition of n = 1. The terminating recursive call to factorial would have a value of 1 assigned to n, and when the recursion unwinds, factorial(n-1) \* n would evaluate to 1 in each stage.

With automatic storage class, each recursive function call would have its own place in the memory (actually on the stack) to store variable **n**. As a result, consecutive calls to factorial will not overwrite variable **n** of the caller. As a result, when the recursion unwinds, we shall have the right value for factorial(n) as  $n*(n-1)(n-2)\dots*1$ .

Finally, for both tasks and functions, it is recommended that you declare your “module” or “program” as automatic, which will make all the functions and tasks in your module/program automatic. Automatic should have been the default for tasks/functions (outside of a “class”), but for backward compatibility, they were left “static” by default. Note that tasks/functions inside a class are automatic by default:

```
module automatic top;
    task doIT ( ... ); //task 'doIT' is now automatic
endmodule
```

## 22.3 Passing Arguments by Value or Reference to Tasks and Functions

SystemVerilog allows two means for passing arguments to tasks and functions: by value or by reference. Arguments can be bound by name or by position (just as we do with module instantiation). You can also give default values to task or function arguments.

### 22.3.1 Pass by Value

The examples we have seen so far pass the arguments by value. That is the default mechanism. Passing by value means that you copy each argument into the subroutine area. Since it is a copy, if the arguments are changed within the function/task, the changes are not visible outside the subroutine. Here is what we have seen so far:

```
function int func1 ( inout int dataIO, input int addressIN );
    func1 = addressIN + dataIO;
endfunction
```

And you call “func1” by passing values to its input/output arguments:

```

int data, address;
...
data = 1234;
address = 2345;
...
newval = func1 ( data, address); //pass by value

```

Similarly, if you have a task header as follows:

```
task automatic PCI_read (inout int data, input int address);
```

you can pass the arguments values during task invocation:

```
PCI_read ('h1234, 'h2345); //pass by value
```

Here is an example that shows that when you send the argument value by “pass by value,” the change made to the value of the argument inside the function is not reflected outside the function:

```

module top;
    int a, result;

    initial begin
        a = $urandom % 6;
        $display ("Before calling function : a=%0d result=%0d",
a, result);

        result = mult(a);
        $display ("After calling function : a=%0d result=%0d",
a, result);
    end

    function int mult(int a);
        a = a + 5;
        return a * 10;
    endfunction
endmodule

```

*Simulation log:*

Before calling function : a=2 result=0

After calling function : a=2 result=70

V C S S i m u l a t i o n R e p o r t

In this example, we declare a function “mult” with input argument “int a.” “a” is a variable in module “top.” We assign a random value to “a” and pass it (by value)

to the function “mult.” Within the function, we change the value of “a” ( $a = a + 5$ ). Now, since the function “mult” was passed “a” by value, the change in “a” within the function is not reflected outside the function. Hence, in the simulation log, we first display the value of “a” passed to the function ( $a = 2$ ):

Before calling function : a=2 result=0

Then we call the function with this value. The “mult” function changes the value of “a” to “ $a + 5$ .” But when we call the display, after calling the function, the value of “a” is still 2:

After calling function : a=2 result=70

This proves that when values of arguments to a subroutine are passed by value, the changes in these arguments within the subroutine are not reflected on these arguments outside of the subroutine. We will see how the same example works when we pass the argument by reference. Next section.

### 22.3.2 Pass by Reference

Arguments passed by reference are not copied into task/function area; rather, a reference to the original argument is passed to the subroutine. The subroutine can then access the argument data via the reference.

Normally, task and function arguments declared as input are copied by value upon entry to the routine, and arguments declared as output are copied by value upon returning from the routine. Inout arguments are copied twice, once upon entry and once upon return from the routine.

Arguments declared with `ref` are not copied but instead are references to the actual arguments used when making the call to the routine.

To indicate argument passing by reference, the argument declaration is preceded by the `ref` keyword:

```
function/task ( ref type argument);
```

As the argument within the task/function is pointing (referencing) to an original argument, any changes to the argument within the subroutine will be visible outside. One of the advantages of passing by reference is when you are dealing with large arrays. With pass by value, you copy the entire array into the subroutine area, while with pass by reference, you only pass a reference to the array. This is a performance implication.

If you do not want the “`ref`” argument to be changed from within the task/function, you have to declare it as “`const ref`.” Any attempt at changing the value of “`const ref`” argument will result in a compilation error.

Here is the same example that we saw with “pass by value.” But the function “mult” now takes its input argument by reference:

```

module top;
    int a, result;

    initial begin

        a = $urandom % 6;
        $display ("Before calling function : a=%0d result=%0d",
a, result);

        result = mult(a);
        $display ("After calling function : a=%0d result=%0d",
a, result);
    end

    function int mult(ref int a);
        a = a + 5;
        return a * 10;
    endfunction
endmodule

```

*Simulation log:*

Before calling function : a=2 result=0

After calling function : a=7 result=70

V C S   S i m u l a t i o n   R e p o r t

In this example, we declare a function “mult” with input argument ‘*ref int a.*’ “a” is a variable at the module “top” level. We assign a random value to “a” and pass it (by reference) to the function “mult.” Within the function, we change the value of “a” ( $a = a + 5$ ). Now, since the function “mult” was passed “a” by reference, the change in “a” within the function will be reflected outside the function. In the simulation log, we first display the value of “a” passed to the function ( $a = 2$ ):

Before calling function : a=2 result=0

Then we call the function with this value. The “mult” function changes the value of “a” to “ $a + 5$ .” Since “a” was passed by reference, the value of “a” is now reflected outside of the function as well:

After calling function : a=7 result=70

This proves that when values of arguments to a subroutine are passed by reference, the changes in these arguments within the subroutine are reflected outside of the subroutine.

And as mentioned before, if you do not want the “*ref*” argument to be changed from within the task/function, you have to declare it as “*const ref*.” Any attempt at

changing the value of “const ref” argument will result in a compilation error. So, if we make the following change:

Change.

```
function int mult(ref int a); //ref
    a = a + 5;
    return a * 10;
endfunction
```

to

```
function int mult(const ref int a); //const ref
    a = a + 5;
    return a * 10;
endfunction
```

We will get the following compile error (Synopsys – VCS):

Error-[IUCV] Invalid use of 'const'  
testbench.sv, 14

'const' variable is either driven or connected to a non-const variable.

Variable 'a' declared as 'const' cannot be used in this context

Source info: a = (a + 5);

It is legal to pass by reference: a variable, a class property, a member of an unpacked structure, or an element of an unpacked array. But it is not legal to pass nets by reference.

Let us look at an example where a task uses “ref”-type arguments. The point here is that whenever the “ref” argument changes in the task, its effect is observed outside the task:

```
module top;
    logic [31:0] data;
    int addr;

    task refT (ref logic [31:0] data);
        #10; data = 'h1234;
        $display($stime,,, "From inside the task - data =
%0h",data);
        #10; data = 'h2345;
        $display($stime,,, "From inside the task - data =
%0h",data);
    endtask

    initial begin
```

```

    fork
        refT (data); //call task

        forever @(data) begin //monitor 'data'
            $display($stime,,, "From outside the task - data =
%0h",data);
        end
    join

    end
endmodule

```

*Simulation log:*

- 10 From inside the task - data = 1234
- 10 From outside the task – data = 1234
- 20 From inside the task - data = 2345
- 20 From outside the task – data = 2345

V C S   S i m u l a t i o n   R e p o r t

The example defines a task named “refT” which takes in a “ref” argument “data.” “data” changes within the task. In the initial block, we fork off calling the task in parallel to monitoring the “data” changes. The point is that the changes in “data” in the task are observed outside the task, as soon as the changes on “data” takes place in the task. The task changes “data” at times 10 and 20. The display statements in simulation log shows that when “data” changes at time 10 inside the task that they are observed at the same time outside the task. The same happens at time 20.

## 22.4 Default Argument Values

For commonly used arguments or to allow for unused arguments, SystemVerilog allows a function/task declaration to specify a default value for each item. The syntax is:

```

function/task ( [direction] [type] argument
= default_expression);

```

When the function or task is called, arguments with default values can be omitted from the call, and the compiler inserts corresponding default value on that argument. If an argument does not have a default value and it is omitted from the call, a compiler error is issued.

Here is an example. The comments in the example explain what is going on:

```

module top;
    logic [31:0] data;
    int addr;
    bit req, gnt;

    task refT (logic [31:0] data = 'h1234, int addr, bit
req = 0);
        $display($stime,,, "data = %0h addr = %0h req = %b",
                data, addr, req);
    endtask

    initial begin

        refT( , addr , ); //data = 1234 addr = 0 req = 0

        refT( , 'h4567); //data = 1234 addr = 4567 req = 0

        refT('h3456, 0, ); //data = 3456 addr = 0 req = 0

        refT(0, 1, 1); //data = 0 addr = 1 req = 1

        //refT( , , ); //ERROR - 'addr' does not have a
default value
        //refT( ); //ERROR - 'addr' does not have a default value
        //refT(0, , 1); //ERROR - 'addr' does not have a
default value
    end
endmodule

```

*Simulation log:*

```

0  data = 1234 addr = 0 req = 0
0  data = 1234 addr = 4567 req = 0
0  data = 3456 addr = 0 req = 0
0  data = 0 addr = 1 req = 1
V C S  S i m u l a t i o n  R e p o r t

```

Any of the following line will give a compile error:

```

refT( , , ); //ERROR - 'addr' does not have a default value
refT( ); //ERROR - 'addr' does not have a default value
refT(0, , 1); //ERROR - 'addr' does not have a
default value

```

The compile error is as follows (Synopsys – VCS):

Error-[TFAFTC] Too few arguments to function/task call  
testbench.sv, 21

"refT(/\* data = 'h00001234 \*/, /\* req = 0 \*/);"

The above function/task call is not done with sufficient arguments.

## 22.5 Argument Binding by Name

So far, we have seen argument binding (when you bind actual to the formal of a subroutine) by position. But just like with a module instantiation, you can bind the formals of a subroutine with actuals by name. Here is an example:

```

module top;
    logic [31:0] data;
    int addr;
    bit req, gnt;

    task refT (logic [31:0] data = 'h1234, string s = "hi");
        $display($stime,,, "data = %0h s = %s",
                 data, s);
    endtask

    initial begin
        refT (.data('h2345), .s("bye") ); //data = 2345 s = bye
        refT (.data('h6789)); //data = 6789 s = hi
        refT (.s("go") ); //data = 1234 s = go
        refT ( , "Hello"); //data = 1234 s = Hello
        refT ( , .s("for") ); //data = 1234 s = for
        refT (.s("fine"), .data('h3456)); //data = 3456 s = fine
        refT (.data ( ), .s ( ) ); //data = 1234 s = hi
        refT ( 'h 4567); //data = 4567 s = hi
        refT ( );
    end
endmodule

```

The comments in the example explain the workings. Note that when you bind a named argument before a positional argument as follows:

```
refT (.data('h5678), );
```

you will get the following compile error (Synopsys – VCS):

```
Error-[IPNA] Incorrect placement of named arguments
testbench.sv, 24
"refT(.data('h00005678), /* s = "hi" */);"
    Named arguments cannot be passed before positional arguments
```

## 22.6 Parameterized Tasks and Functions

Parameterized tasks and functions allow reusable and modular code development. Such subroutines allow you to generically specify or define an implementation. Once such a subroutine is defined, one may provide the parameters that fully define the subroutine's functionality. This allows for only one definition to be written and maintained instead of multiple subroutines with the same functionality but different array sizes, variable widths, etc.

There are two ways in which you can create/invoke a parameterized task/function. One is to pass the parameter through a parameterized module. The other is through the use of static methods in parameterized classes.

First, let us look at how to pass a parameter to a task via parameterized module:

```
module bottom #(data_width = 7); //parameterized module
  logic [31:0] data;

  //use module parameter to parameterize task
  task refT (logic [data_width:0] data);
    #1; data = $urandom % data_width;
    $display($stime,,,"data_width = %0d data = %h", data_
width, data);
  endtask

  //task refT #(data_width = 15) (logic [data_width:0] data);
  //ERROR
endmodule

module top;
  int top_data;

  bottom #(15) b1( );
```

```

bottom #(23) b2( );
initial begin
    b1.refT (top_data);
    b2.refT (top_data);

    //b1.refT ([23:0] top_data); //ERROR
end
endmodule

```

*Simulation log:*

```

1 data_width = 15 data = 0008
2 data_width = 23 data = ffffff
V C S S i m u l a t i o n R e p o r t

```

Module “bottom” is parameterized with a parameter “data\_width = 7.” This parameter is used in task “refT” to parameterize the task. Note that you cannot directly define the parameter in task definition itself. So, the following is illegal and will give a compile error:

```
task refT #(data_width = 15) (logic [data_width:0] data); //ERROR
```

Error-[SE] Syntax error

Following verilog source has syntax error :

"testbench.sv", 10: token is '#'

```
task refT #(data_width = 15) (logic [data_width:0] data); //ERROR
```

In module “top,” we instantiate module “bottom” and pass it a value to the parameter “data\_width” of module “bottom.” This parameter will in turn be used by the task “refT.” We instantiate module “bottom” twice, passing a new value to the parameter with each instance. The simulation log shows the value of data\_width for each instance of “bottom.” This shows that you can have the same task used by different instances of modules with different parameter values. Reuse of code.

Note that you cannot do the following either:

```
b1.refT ([23:0] top_data); //ERROR
```

You cannot directly pass a new value to “data\_width” during task invocation. You will get the following compile error:

Error-[SE] Syntax error

Following verilog source has syntax error :

"testbench.sv", 24: token is '['

```
b1.refT ([23:0] top_data); //ERROR
```

Now, let us see how to parameterize a task using static methods in parameterized classes. The example shows how to use static class methods along with class parameterization to implement parameterized subroutines:

```
class bottom #(data_width = 7); //parameterized class

    //must be a static function
    static function logic [data_width:0] refT;
        //use class parameter to parameterize the function
        refT = $urandom % data_width;
        $display($stime,,, "data_width = %0d data = %h", data_width, refT);
    endfunction

endclass

module top;
    logic [31:0] data_out;

    bottom b1 = new ( );

    initial begin

        //using class resolution operator (::); data_width = 15
        void' (bottom #(15) :: refT );

        data_out = bottom #(23) :: refT; //data_width = 23
        $display($stime,,, "data_out = %0h", data_out);

        void ' (b1.refT); //default data_width

    end
endmodule
```

*Simulation log:*

```
0  data_width = 15 data = 0008
0  data_width = 23 data = fffff1
```

```
0  data_out = fffff1
0  data_width = 7 data = fd
V C S  S i m u l a t i o n  R e p o r t
```

Class “bottom” is parameterized with the parameter “data\_width” with a default value of 7. It contains the static function “refT,” which uses the class parameter “data\_width” to size its output. Note that this must be a static function. By default, a function is automatic in a class. So, you have to explicitly declare the function as static. If you do not make the function “static,” you will get the following error (Synopsys – VCS):

```
Error-[ISRF] Illegal scoped reference found
testbench.sv, 20
"bottom#(15)::refT"
    Scoped reference to the non-static class task/function
    '_vcs_unit__1778430441_bottom_2685413023941984662_8::refT'      is      not
        allowed.

Error-[ISRF] Illegal scoped reference found
testbench.sv, 22
"bottom#(23)::refT"
    Scoped reference to the non-static class task/function
    '_vcs_unit__1778430441_bottom_3363648766886695086_8::refT'      is      not
        allowed.
```

In the module top, we instantiate “bottom.” Then in the initial block, we access the function “refT” using the static class scope resolution operator (:) as well as using the instance name. We pass different “data\_width” parameter values through these invocations. Simulation log shows that the function receives different data\_width with each invocation.

You can also parameterize datatype of a function. In the following example, datatype is parameterized and can be overridden during instantiation:

```
class myClass #(type dataT = real); //parameterize datatype
                                         //default dataT of type 'real'
    dataT inven;

    //function 'mult' of type dataT
    function dataT mult (dataT inA);
        mult = inven * inA;
    endfunction
endclass

module top;
    real dR;
    bit [7:0] bR;
    int iR;
```

```

//dataT of default type 'real'
myClass realT;

// change datatype of dataT to type 'bit[7:0]'
myClass #(bit[7:0]) bitT;

// change datatype of dataT to type 'int'
myClass #(int) intT;

initial begin
    realT = new;
    bitT = new;
    intT = new;

    realT.inven = 3.14; // 'inven' is of type real
    dR = realT.mult (1.2); // 'mult' is of type real
    $display ("mult with 'real' type = %0.2f", dR);

    bitT.inven = 'h ff; // 'inven' is of type bit[7:0]
    bR = bitT.mult('h11); // 'mult' is of type bit[7:0]
    $display ("mult with 'bit' type = %h", bR);

    intT.inven = 10; // 'inven' is of type int
    iR = intT.mult (10); // 'mult' is of type int
    $display ("mult with 'int' type = %0d", iR );
end
endmodule

```

*Simulation log:*

```

mult with 'real' type = 3.77
mult with 'bit' type = ef
mult with 'int' type = 100

```

V C S   S i m u l a t i o n   R e p o r t

A class “myClass” is declared with input parameter “dataT” which is of datatype “real.” We are parameterizing the class with a datatype and declaring the function “mult” of that datatype. The idea is to be able to change the type of class parameter and pass it as the type of the function. Class property “inven” is also of type dataT and so is the input “inA” to the function “mult.”

In module “top,” we declare three classes:

```

// dataT of default type 'real'
myClass           realT;

// change datatype of dataT to 'bit[7:0]'

```

```
myClass #(bit[7:0]) bitT;  
  
// change datatype of dataT to 'int'  
myClass #(int) intT;
```

Each class instance changes the datatype of the class parameter “dataT.” This in turn changes the type of the function “mult.” We then assign a value to “inven” which takes on different datatypes based on the class instance it uses (realT for “real” datatype or bitT for “bit[7:0]” datatype or intT for “int” datatype). The simulation log shows the result output of “mult” which inherits different datatypes with different instances of the class “myClass.”

# Chapter 23

## Procedural and Continuous Assignments



**Introduction** This chapter will delve into the nuances of procedural and continuous assignments. It discusses features such as blocking and non-blocking procedural assignment, assign/deassign, force-release, etc.

We have seen procedural assignments throughout this book. We have not seen continuous assignments so far. This chapter will delve into further nuances of both types of assignments. As we have seen, assignments are the basic mechanism to put values (assign values) to nets and variables.

Procedural assignment assigns values to variables (in procedural blocks), and continuous assignment assigns values to nets (outside of the procedural block). There is also a “procedural continuous” assignment that assigns values to nets and variables inside a procedural block.

Continuous assignments drive nets in a manner similar to the way gates drive nets. Continuous assignments, as the name suggests, drive the net/variable continuously. As soon as the right-hand side (RHS) of the assignment changes, the left-hand side (LHS) of the assignment will change. Think of RHS as a combinational circuit that drives the net continuously.

Procedural assignments assign values to variables. The assignment is not continuous. The variable holds the value of assignment until the next procedural statement assigns new value to the variable.

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_23](https://doi.org/10.1007/978-3-030-71319-5_23)) contains supplementary material, which is available to authorized users.

## 23.1 Procedural Assignments

Procedural assignments occur in procedural blocks such as “always” (and variations thereof), “initial,” “task,” and “function.” Event controls, delay controls, if...else statements, case statements, looping statements, etc. can all be used to control when assignments are evaluated and/or assigned.

The RHS of the procedural assignment can be any expression that evaluates a value. But note that the LHS of the assignment may restrict what is a legal expression on the RHS. We will see examples of that in the coming sections.

The LHS can be singular variable, aggregate variable, bit-select, part-select, slices of packed array, and slices of unpacked array. We have seen many such examples throughout the book.

There are basically following types of procedural assignments:

1. Blocking assignment
2. Non-blocking assignment

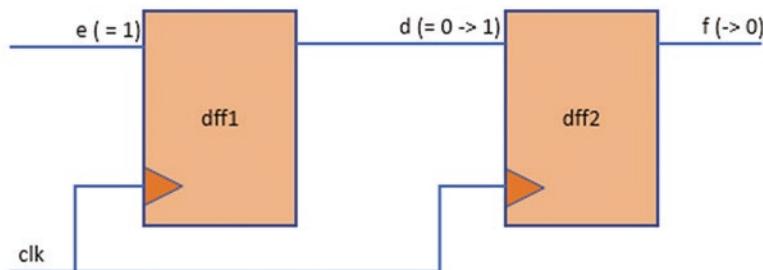
The simplest form of assignment is using the operator “`= .`” But there are other operators as well, such as `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `<<<=` and `>>>=`. We will see examples of these in coming the sections.

### 23.1.1 Blocking Versus Non-Blocking Procedural Assignments

We have seen blocking assignments throughout this book. So, we will refrain from further discussion on that. Non-blocking is new, and we need to understand the difference between the two. The non-blocking procedural assignment allows assignment scheduling without blocking the procedural flow. The non-blocking procedural assignment statement can be used whenever several variable assignments within the same time step can be made without regard to order or dependence upon each other.

Syntax for non-blocking is:

`variable_name <= [delay_or_event_control] expression.`



**Fig. 23.1** Non-blocking assignment shown as sequential flops

`<=` is the non-blocking assignment operator. The non-blocking assignment operator is the same as the less-than-or-equal-to relational operator. But non-blocking is decided from the context in which it appears.

Non-blocking execution can be viewed as a two-step process:

1. Evaluate the RHS of the non-blocking statement at the beginning of the time step.
2. Evaluate the LHS of the non-blocking statement at the end of the time step.

Let us look at a simple example. We will contrast blocking with non-blocking:

```
module nblock;
    logic a, b, c, d, e, f, temp;

    initial begin
        a = 0; b = 1;

        //Blocking assignment
        a = b;
        c = a;

        $display($stime,,, "Blocking a=%b b=%b c=%b", a, b, c);
    end

    initial begin
        d = 0; e = 1;

        //Non-Blocking assignment
        d <= e;
        f <= d;

        $monitor($stime,,, "Non-blocking d=%b e=%b f=%b",
d, e, f);
    end

    initial begin
        #10;
        d = 0; e = 1;

        //Following is the same as non-blocking.
        temp = e;
        f = d;
        d = temp;

        $display($stime,,, "Using temp d=%b e=%b f=%b", d, e, f);
    end
endmodule
```

*Simulation log:*

```
0  Blocking a=1 b=1 c=1
0  Non-blocking d=1 e=1 f=0
10 Using temp d=1 e=1 f=0
V C S  S i m u l a t i o n  R e p o r t
```

There are three “initial” blocks. The first one does a blocking assignment:

```
a = b;
c = a;
```

With this assignment, “b” is assigned to “a” – *then* – the new value of “a” is assigned to “c.” So, the new value of “a” (from  $a = b$ ) is assigned to “c” ( $c = a$ ). Since  $a = 0$  and  $b = 1$ ,  $a = b$  results in  $a = 1$  and then  $c = a$  results in  $c = 1$ . Hence, you see the following in simulation log:

0 Blocking a=1 b=1 c=1

In contrast, the non-blocking assignment works as follows. Think of it as two flops connected in series:

```
d <= e;
f <= d;
```

The two assignments take place in parallel since they are non-blocking. The first assignment does not block the second assignment. The current value of “e” is assigned to “d,” and at the same time, the current value (*– before* – the assignment  $d <= e$ ) of “d” is assigned to “f.” With  $d = 0$  and  $e = 1$ , you get the following in the simulation log:

0 Non-blocking d=1 e=1 f=0

You can think of the above non-blocking as the following expanded version:

```
temp = e;
f = d;
d = temp;
```

With  $d = 0$  and  $e = 1$ , the current value of “e” (= 1) is put in a temp variable ( $temp = 1$ ). Then the current value of “d” (= 0) is assigned to “f” ( $f = d$ ). So,  $f = 0$  and then the value of “e” which was stored in “temp” is assigned to “d.” So,  $d = 1$ . Hence, the simulation log shows the following. Note that this result is the same as that of non-blocking assignment:

10 Using temp d=1 e=1 f=0

Here is a hardware description of how non-blocking works. Two flops connected in serial.

As you know from hardware design, when clk toggles, the value of “e” (= 1) gets propagated to “d” (so, “d” goes from 0 to 1), and at the *same* time, the value of “d” (= 0) gets propagated to “f” (so, f = 0).

Here is another example, which shows intra-assignment delays and difference between blocking and non-blocking assignments:

```
module nonblock1;
  logic a, b, c, d, e, f;

  // blocking assignments
  initial begin
    a = #10 1; // a will be assigned 1 at time 10
    b = #2 0; // b will be assigned 0 at time 12
    c = #4 1; // c will be assigned 1 at time 16
  end

  // nonblocking assignments
  initial begin
    d <= #10 1; // d will be assigned 1 at time 10
    e <= #2 0; // e will be assigned 0 at time 2
    f <= #4 1; // f will be assigned 1 at time 4
  end

  initial begin
    $monitor($stime,,, "a=%b b=%b c=%b d=%b e=%b f=%b", a, b,
c, d, e, f);
  end
endmodule
```

*Simulation log:*

```
0  a=x b=x c=x d=x e=x f=x
2  a=x b=x c=x d=x e=0 f=x
4  a=x b=x c=x d=x e=0 f=1
10 a=1 b=x c=x d=1 e=0 f=1
12 a=1 b=0 c=x d=1 e=0 f=1
16 a=1 b=0 c=1 d=1 e=0 f=1
V C S  S i m u l a t i o n  R e p o r t
```

This example shows that the blocking assignments block until their assignment delay is over, while non-blocking assignment evaluates the assignment and gives control to the next statement without waiting for the delay in its assignment.

The comments in the code show when these assignments will take place. The blocking assignments execute in a linear time order, while the non-blocking assignments execute in a parallel time order. The simulation log shows how the assignments take place. Please study it carefully and you will see the difference between the two types of assignments.

Here's a simple example showing how race condition exists with blocking assignment and how non-blocking removes it. As an exercise try it out with different values of i1.

With blocking:

```
module race (output logic a, b, c, input i1);
  always @ (i1) begin
    #2 a = i1; //race condition
    #3 b = i1;
  always @ (i1)
    #2 c = a; //race condition
  endmodule
```

With non-blocking:

```
module race (output logic a, b, c, input i1);
  always @ (i1) begin
    #2 a <= i1; //no race condition
    #3 b <= i1;
  always @ (i1)
    #2 c <= a; //no race condition
  endmodule
```

Finally, the following rules are key to avoiding race conditions (Cummings, [www.sunburst-design.com](http://www.sunburst-design.com)) in SystemVerilog:

1. For sequential logic, always use non-blocking assignments.
2. For latches use non-blocking assignments.
3. For combinational logic use blocking assignments.
4. Do not mix blocking and non-blocking assignments in a single “always” block. But if you have sequential and combinational logic in the same “always” block, use non-blocking assignments.
5. Do not make assignments to the same variable from two different procedural blocks.
6. Use \$strobe to display values assigned using non-blocking assignments. \$strobe displays values at the end of a time step which is when the non-blocking assignment completes.
7. Do not use #0 procedural assignments. You will simply play with the ordering of events in a given time step. It will create a debug nightmare.

### 23.1.2 Continuous Assignment

Continuous assignment allows you to have an assignment outside of a procedural block where whenever the RHS of the continuous assignment changes, the LHS will change. This is very useful in designing combinatorial logic in RTL. You should not assign a variable both from a procedural block and in a continuous assignment. You will have a race condition.

Syntax is:

```
assign variable = expression;
```

Here is an example:

```
module cassign;
bit[3:0] a, b, c;
wire [31:0] bus;

assign a = b; //continuous assignment
assign bus = c + b; //continuous assignment

initial begin
    forever begin
        b = $urandom % 2;
        c = $urandom % 2;
        #10;
    end
end

initial #40 $finish(2);

initial begin
    $monitor($stime,,,"a=%0d b=%0d c=%0d bus=%0d", a, b,
c, bus);
end
endmodule
```

*Simulation log:*

```
0 a=0 b=0 c=0 bus=0
10 a=1 b=1 c=0 bus=1
20 a=1 b=1 c=1 bus=2
30 a=0 b=0 c=1 bus=1
$finish called from file "testbench.sv", line 21.
$finish at simulation time      40
VCS Simulation Report
```

There are two continuous assignments in the model. “assign a = b;” will change the value of “a” whenever the value of “b” changes. Value of “a” can change from a procedural assignment or from another continuous assignment. Similarly, “assign bus = c + b;” causes “bus” to change whenever the expression “c + b” changes. Simulation log shows that “a” and “bus” continually change whenever their RHS expressions change.

You can also have a function call on the RHS of a continuous assignment. Whenever the function arguments change, the RHS will be reevaluated. Here is an example, identical to the one above, but we have replaced

```
"assign bus = c + b;"
```

with

```
"assign bus = add(c,a);"
```

where “add” is a function:

```
module cassign;
bit[3:0] a, b, c;
wire [31:0] bus;

    assign a = b; //continuous assignment
    //assign bus = c + a; //continuous assignment
    assign bus = add (c, a); //continuous assignment with
function

    function int add (fc, fa);
        add = fc + fa;
    endfunction

initial begin
    forever begin
        b = $urandom % 2;
        c = $urandom % 2;
        #10;
    end
end

initial #40 $finish(2);
```

```

initial begin
    $monitor($stime,,, "a=%0d b=%0d c=%0d bus=%0d", a, b,
c, bus);
end
endmodule

```

*Simulation log:*

```

0  a=0 b=0 c=0 bus=0
10 a=1 b=1 c=0 bus=1
20 a=1 b=1 c=1 bus=2
30 a=0 b=0 c=1 bus=1
$finish called from file "testbench.sv", line 21.
$finish at simulation time      40
V C S  S i m u l a t i o n  R e p o r t

```

## 23.2 Procedural Continuous Assignment: Assign and Deassign

You can have continuous assignment outside of a procedural block (as we saw above), or you can have that in a procedural block as well. The procedural continuous assignment will override all procedural assignments to a variable. The keywords are *assign – deassign*.

Assign will “assign” a variable in the procedural block (this keyword is the same as for continuous assignment – outside – of a procedural block). But note that you cannot assign the same variable both procedurally and non-procedurally outside a procedural block.

The “deassign” statement will end a procedural continuous assignment to a variable.

You can assign the same variable multiple times. The value of the variable will remain the same until the variable is assigned a new value through a procedural assignment or a procedural continuous assignment. For example, this feature is suitable for modeling of asynchronous clear/preset on a flop.

The LHS of the assignment will be a singular variable or concatenation of variables. But it cannot be a bit-select or part-select of a variable.

If there is already a procedural assignment to a variable, the continuous procedural assignment will deassign the variable and then make the new procedural assignment.

This is not a very widely used feature. This is basically a legacy feature.

Here is a simple example:

```

module cassign;
bit[3:0] a, b, c;

//Error: cannot assign non-procedurally if procedural is
assigned
//assign a = b;

initial begin
    assign a = b; //procedural continuous assignment
    #40;
    deassign a; //deassign
    #40;
    assign a = b; //assign again
end

initial begin
    b = 1;
    forever begin
        b = $urandom;
        #10;
    end
end

initial #100 $finish(2);
initial begin
    $monitor($stime,,,"a=%0d b=%0d", a, b);
end
endmodule

```

*Simulation log:*

```

0  a=4 b=4
10 a=1 b=1
20 a=9 b=9
30 a=3 b=3
40 a=3 b=13
60 a=3 b=5
70 a=3 b=2
80 a=1 b=1
90 a=13 b=13

```

\$finish called from file "testbench.sv", line 25.

\$finish at simulation time 100

VCS Simulation Report

We procedurally assign “assign a = b;” in the initial block. This means that whenever “b” changes, “a” will now change. “b” can change anywhere in your model, and “a” will change accordingly. This is different from a simple assignment (“a = b;”) where “a” will change one time when “b” changes. In our example, “b” change continually, and that is reflected on “a” as shown in the simulation log until time 40:

```
0  a=4 b=4
10 a=1 b=1
20 a=9 b=9
30 a=3 b=3
```

As time 40, we “deassign” “a” (“deassign a;”). This means that changes on “b” will not be reflected on “a” anymore (until you reassign “a”). Here is the simulation log that shows that “a” does not change anymore even if “b” keeps changing, from time 40 to 70:

```
40  a=3 b=13
60  a=3 b=5
70  a=3 b=2
```

We then reassign (“assign a = b;”) at time 80 and the “a” starts changing again. That is shown in simulation log from time 80 to 90:

```
80  a=1 b=1
90  a=13 b=13
```

### 23.3 Procedural Continuous Assignment: Force-Release

Right off the bat, I should say that this is a very rarely used feature. I do not recommend using it. You will end up playing with the simulator scheduling semantics resulting in spaghetti code. But here it is for the sake of completeness.

This is another type of procedural continuous assignment. It has a similar effect as we saw in assign-deassign pair. The only difference is that a “force” can be applied to both nets (wire) and variables. The left-hand side of the assignment can be a reference to a singular variable, a net, a constant bit-select of a vector net, a constant part-select of a vector net, or a concatenation of these. It cannot be a bit-select or a part-select of a variable or of a net with a user-defined nettype.

A “force” will override any other assignment to the variable (procedural assignment, continuous assignment, or procedural continuous assignment). When released (*release*), if the variable is not driven by a continuous assignment and does not currently have an active “assign,” procedural continuous assignment, the variable cannot immediately change value and will maintain its current value until the next procedural assignment to the variable is executed.

Here is an example:

```

module cassign;
  bit[3:0] b, c, d;
  bit [7:0] a;
  wire [3:0] e;

initial begin
  assign a = b + c; //procedural continuous assignment
  #40;
  force a = b + c + d; //force
  force e = b & c & d;
  #40;
  release a; //release
  release e;
end

initial begin
  b = 1;
  forever begin
    b = $urandom; c = $urandom; d = $urandom;
    #10;
  end
end

initial #120 $finish(2);
initial begin
  $monitor($stime,,, "a=%0d b=%0d c=%0d d=%0d e=%0d", a, b,
c, d, e);
  end
endmodule

```

*Simulation log:*

```

0  a=3 b=3 c=0 d=13 e=z
10 a=9 b=2 c=7 d=1 e=z
20 a=17 b=5 c=12 d=8 e=z
30 a=24 b=12 c=12 d=11 e=z
40 a=20 b=11 c=5 d=4 e=0
50 a=31 b=12 c=7 d=12 e=4
60 a=19 b=10 c=6 d=3 e=2
70 a=33 b=13 c=12 d=8 e=8

```

```

80  a=22 b=11 c=11 d=3 e=z
90  a=25 b=13 c=12 d=6 e=z
100 a=18 b=9 c=9 d=10 e=z
110 a=22 b=14 c=8 d=13 e=z
$finish called from file "testbench.sv", line 25.
$finish at simulation time      120
    V C S   S i m u l a t i o n   R e p o r t

```

In this example, we first continuous assign “assign  $a = b + c;$ ” in the initial block. We do not assign anything to the “wire” “e.” So, “a” will start changing as “b” and “c” changes. This is shown in the simulation log as follows. “a” is the addition of  $b + c$ . Note that we have not assigned anything to the “wire” “e,” so it will stay in its default state “Z”:

```

0  a=3 b=3 c=0 d=13 e=z
10 a=9 b=2 c=7 d=1 e=z
20 a=17 b=5 c=12 d=8 e=z
30 a=24 b=12 c=12 d=11 e=z

```

Then we “force” “a” to  $a = b + c + d$ . This will *override* the previous continuous assign to “a.” Now “a” will change according to “ $b + c + d$ ”. We also force the wire “e,” and it will also change according to “ $b + c + d$ ;”. That is shown in the simulation log as follows:

```

40  a=20 b=11 c=5 d=4 e=0
50  a=31 b=12 c=7 d=12 e=4
60  a=19 b=10 c=6 d=3 e=2
70  a=33 b=13 c=12 d=8 e=8

```

Then, we “release” both “a” and “e.” Here is the important point to note. Once we release “a,” its previous continuous assignment (“assign  $a = b + c$ ”) will come into effect again. “a” will again start changing according to “b” and “c.” When we release the wire “e,” it will go back to its default value “Z.” This is shown in simulation log as follows:

```

80  a=22 b=11 c=11 d=3 e=z
90  a=25 b=13 c=12 d=6 e=z
100 a=18 b=9 c=9 d=10 e=z
110 a=22 b=14 c=8 d=13 e=z

```

# Chapter 24

## Utility System Tasks and Functions



**Introduction** SystemVerilog offers multitudes of utility system tasks and functions. This chapter discusses, simulation control system tasks, simulation time system functions, timescale system functions, conversion functions, array querying system functions, math functions, bit-vector functions, severity system tasks, random and probabilistic distribution functions, queue management stochastic analysis tasks, etc.

### 24.1 Simulation Control System Tasks

We have used simulation control tasks throughout the book. To recap, there are three such tasks: \$finish, \$stop, and \$exit.

\$finish(n) causes simulator to exit (finish) the simulation.

\$finish takes three argument values, 0, 1, and 2. \$finish(0) means do not print anything. Just quit the simulation. \$finish(1) means print simulation time and location when you quit simulation. \$finish(2) (which is what I have used throughout the book) prints simulation time, location, and statistics about the memory usage and CPU time used by simulation. Or you can simply use \$finish without any arguments.

\$stop suspends simulation but does not finish it. During debug mode, you may want to suspend simulation but not finish/end it. \$stop also takes the same three arguments as \$finish and has the same meaning.

\$exit applies to “program” blocks (as opposed to “module” blocks). It waits for all “program” blocks to complete and then make an implicit call to \$finish. A program block may terminate the threads of all its initial procedures as well as all of their

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_24](https://doi.org/10.1007/978-3-030-71319-5_24)) contains supplementary material, which is available to authorized users.

descendants, explicitly by calling the \$exit system task. Calling \$exit from a thread or its descendent thread that does not originate in an “initial” procedure in a program is ignored, and a warning is issued to indicate that the call to \$exit has been ignored.

## 24.2 Simulation Time System Functions

These system functions provide access to current simulation time. They are \$time, \$stime, and \$realtime.

\$time returns an integer that is a 64-bit time, scaled to the time precision of the module that invoked it.

\$stime is the same as \$time but is a 32-bit time. So, in simulation log, if you do not want 64-bit leading zeros, you use \$stime. I have used \$stime throughout the book. If the actual simulation time does not fit in 32-bits, the low-order 32-bits of the current simulation time are returned.

\$realtime returns a real number time, scaled to the time precision of the module that invoked it.

Here is a simple example showing all three functions:

```
`timescale 1ns / 1ps
module cassign;
    bit [7:0] a;

initial begin

    $printtimescale( );

#1.1 a = $urandom;
$display($time,,, "a = %0d",a); //64-bit time

#2.1 a = $urandom;
$display($stime,,, "a = %0d",a); //32-bit time

#3.1 a = $urandom;
$display($realtime,,, "a = %0d",a); //real number time
end
endmodule
```

*Simulation log:*

TimeScale of cassign is 1 ns / 1 ps

```
1  a = 54
3  a = 60
6.3 a = 125
VCS Simulation Report
```

In this example, we are setting “a” to a random value at interval of 1.1 ns. Note that the time precision of the module is 1ps. In other words, when we say 1.1, it means that the 1.1 is in nanoseconds (ns) and that simulation will run with time precision of 1100ps. 1ns is the unit of measurement for delay, and 1ps is how the delays are rounded before being used in the simulation.

Both \$time and \$stime return an integer value. Hence, both these functions truncate time. So, 1.1 becomes 1 and 1.1+2.1 becomes 3. Note carefully that, in the simulation log, the first display statement at time 1 (a= 36) has a much larger leading space – since \$time is a 64-bit number. The second display statement at time 3 (a = 129) has lesser leading space since \$stime is a 32-bit number. And the last display statement shows the real-time value of time ( $1.1 + 2.1 + 3.1 = 6.3$ ).

## 24.3 Timescale System Tasks

There are two timescale system tasks: \$printtimescale and \$timeformat.

### 24.3.1 *\$printtimescale*

We just saw the use of \$printtimescale in the above example. It displays the time unit and precision for a particular module. It’s syntax is:

```
$printtimescale [ (hierarchical_identifier) ];
```

If you do not specify the hierarchical\_identifier, it displays the time unit and precision of the module that is the current scope. Else, it will display the time unit and precision of the hierarchical module passed to it.

### 24.3.2 *\$timeformat*

When you want to display time in a uniform way (e.g., a model with time precision 1ps/1ps; you want to display this time in “ns”), you can do that using \$timeformat system task. It specifies how the %t format specification reports time information for many display system tasks, such as \$write, \$display, \$strobe, \$monitor, \$fwrite, \$fdisplay, \$fstrobe, and \$fmonitor. The syntax is:

```
$timeformat [ (units_number, precision_number, suffix_string, minimum_field_width) ];
```

For example, \$timeformat ( -9, 5, “ns”, 10);

where -9 stands for nanosecond time unit, 5 is the number of digits of precision, “ns” is a string that will be attached to the time representation, and 10 is the entire field width of the time display.

**Table 24.1** \$timeformat units\_number arguments

Unit number	Time unit	Unit number	Time unit
0	1 s	-8	10 ns
-1	100 ms	-9	1 ns
-2	10 ms	-10	100 ps
-3	1 ms	-11	10 ps
-4	100 us	-12	1 ps
-5	10 us	-13	100 fs
-6	1 us	-14	10 fs
-7	100 ns	-15	1 fs

**Table 24.2** \$timeformat default value of arguments

Argument	Default
units_number	The smallest time precision argument of all the `timescale compiler directive in the source description
precision_number	0
suffix_string	Null character
minimum_field_width	20

The unit\_number arguments are enumerated as shown in Table 24.1.

The default value of the arguments is as shown in Table 24.2.

Here is an example:

```

`timescale 1ps / 1ps
module cassign;
  bit [7:0] a;

initial begin

$printtimescale( );
$timeformat(-9, 5, "ns", 10);

#3.111 a = $urandom;
$display("@time %t a = %0d",$stime,a);

#7111 a = $urandom;
$display("@time %t a = %0d",$time,a);

#5111.11 a = $urandom;
$display("@time %t a = %0d",$realtime,a);

end
endmodule

```

*Simulation log:*

```
TimeScale of cassign is 1 ps / 1 ps
@time 0.00300ns a = 54
@time 7.11400ns a = 60
@time 12.22500ns a = 125
VCS Simulation Report
```

`timescale and \$timeformat go hand in hand. `timescale defines the time precision of the time units in the model. \$timeformat will convert that time precision into whatever time unit you want to display the time using “%t”.

In our model, time precision is 1ps (`timescale 1ps/1ps). So, all the time delays in the model have a precision of 1ps (e.g., #3.111 means 3.111ps). In the model, we want to display all time values in nanoseconds “ns.” So, we specify \$timeformat (-9, 5, “ns”, 10).

For #3.111 a = \$urandom, we get the following in simulation log. #3.111ps converted to “ns” is 0.00300ns:

```
@time 0.00300ns a = 54
```

For #7111 a = \$urandom, we get the following. #7111ps = 7.111ns. So, 0.00300ns+7.111ns = 7.114ns:

```
@time 7.114ns a = 60
```

For #5111.11ps a = \$urandom, we get the following: #5111.11ps = 5.111ns. So, 0.003ns+7.111ns+5.111ns = 12.225ns:

```
@time 12.225ns a = 125
```

## 24.4 Conversion Functions

You can convert to/from real numbers. You can also convert values to signed or unsigned value (refer to Sect. 2.13 for static casting that also allows conversions, such as real to string, real to int, signed, etc.).

The real to/from conversion functions are as follows:

```
integer      $rtoi (real_value)
real         $itor(int_value)
[63:0]       $realtobits (real_value)
real         $bitstoreal (bit_value)
[31:0]       $shortrealtobits (shortreal_value)
shortreal   $bitstoshortreal (bit_value)
```

\$rtoi converts real values to integer type by truncating the real value (e.g., 234.56 becomes 234). Note that \$rtoi differs from casting a real value to an integer (see static casting in Sect. 2.13), in that casting will round the number instead of

truncating it. Also, when you directly assign a real value to an integer, that also will round the number and not truncate it.

\$itor converts integer value to real value:

\$realtobits converts values from a real type to a 64-bit value.

\$bitstoreal converts a bit pattern created by \$realtobits to a “real” value.

\$shortrealtobits converts a shortreal-type value to a 32-bit value.

\$bitstoshortreal converts a bit pattern created by \$shortrealtobits to a “short-real” value.

Here is an example:

```

module convert;
integer i1;
real r1;
bit [63:0] b1;
shortreal srl;
bit [31:0] b2;

initial begin
r1 = 3456.78;
i1 = $rtoi(r1);
$display("$rtoi: r1 = %f, i1 = %0d",r1,i1);

i1 = 1234;
r1 = $itor(i1);
$display("$itor: i1 = %0d, r1 = %f \n",i1,r1);

r1 = 4567.678012;
b1 = $realtobits(r1);
$display("$realtobits: r1 = %f b1 = %b",r1,b1);

r1 = $bitstoreal(b1);
$display("$bitstoreal: b1 = %b r1 = %f \n",b1,r1);

srl = 7890.123047;
b2 = $shortrealtobits(srl);
$display("$shortrealtobits: srl = %f b2 = %b",srl,b2);

srl = $bitstoshortreal(b2);
$display("$bitstoshortreal: b2 = %b srl = %f",b2,srl);
end
endmodule

```

*Simulation log:*

\$rtoi: r1 = 3456.780000, i1 = 3456

\$itor: i1 = 1234, r1 = 1234.000000

```
$realtobits: r1 = 4567.678012 b1 = 01000000101100011010111010110110010
      01000110001100011001001100
$bitstoreal:   b1 = 010000001011000110101110101101100100011000
      11000011001001100 r1 = 4567.678012
$shortrealtobits: sr1 = 7890.123047 b2 = 010001011110110100100001111100
$bitstoshortreal: b2 = 010001011110110100100001111100 sr1 = 7890.123047
V C S S i m u l a t i o n R e p o r t
```

All the conversion functions are used in the model. Note the difference between conversions from \$real and \$shortreal. If you did a \$shortrealtobits to a 64-bit vector, the first 32-bits will be padded with zeros (I have not shown that in the example – but try it out).

#### 24.4.1 *Conversion to/from Signed/Unsigned Expression*

\$signed and \$unsigned system functions are available for casting the signedness of expressions. The static casting explained in Sect. 2.13 is directly applicable to usage of these system functions. Refer to that section on how the signedness conversion of numbers is achieved.

For the sake of completeness, here is an example:

```
module convert;
  logic [15:0] A1; //unsigned
  logic signed [15:0] s1; //signed

  initial begin
    A1 = $unsigned(-1); //convert -1 to unsigned
    $display("$unsigned(-1): A1 = %b",A1);

    s1 = $signed(16'h ffff); //convert 'h ffff to signed
    $display("$signed(16'b ffff): s1 = %0d",s1);
  end
endmodule
```

*Simulation log:*

\$unsigned(-1): A1 = 1111111111111111

\$signed(16'b ffff): s1 = -1

V C S S i m u l a t i o n R e p o r t

## 24.5 \$bits: Expression Size System Function

The \$bits system function returns how many bits are represented by an expression. The expression can contain any type of value, including packed or unpacked arrays, structures, unions, and literal numbers. The syntax is:

```
$bits(expression)
```

Here is an example:

```
module convert;
    logic [15:0] A1;
    bit [31:0] b1;
    struct { byte byt1; logic [31:0] bl1; } st1;
    logic [3:0] [7:0] ar1 [3:0];

    initial begin
        $display("Size of 'logic [15:0]' is = %0d", $bits(A1) );
        $display("Size of 'bit [31:0]' is = %0d", $bits(b1) );
        $display("Size of struct is = %0d", $bits(st1) );
        $display("Size of 'logic [3:0] [7:0] ar1 [3:0]' is = %0d", $bits(ar1) );
    end
endmodule
```

*Simulation log:*

```
size of 'logic [15:0]' is = 16
size of 'bit [31:0]' is = 32
size of struct is = 40
Size of 'logic [3:0] [7:0] ar1 [3:0]' is = 128
```

Different data types, structure, and array are defined in the model. The size of “A1” and “b1” are obvious. The size of struct “st1” is “byte byt1( = 8)” plus “logic[31:0] bl1 ( = 32).” So,  $8+32 = 40$ . The size of the array “ar1” is packed dimension ( $[3:0][7:0]$ ) =  $4*8 = 32$ . And unpacked dimension of  $[3:0] = 4$  means the total number of bits in “ar1” =  $4*8*4 = 128$ .

The return value of \$bits system function can be determined statically at elaboration time, meaning it is treated as a simple literal value and hence synthesizable.

## 24.6 Array Querying System Functions

SystemVerilog adds special system functions for working with arrays. They return information about a particular dimension of an array. The return type is an integer, and the default for the optional dimension expression is 1 (we will see an example on what dimension expression means).

These system functions are shown in Table 24.3:

Here is an example:

```
module arr;
  // Dimensions
  //      3      4          1      2
  bit [15:0][7:0] arr [0:31][63:31];

  initial begin
    //DEFAULT Dimension 1 - [0:31]
    disp(1);

    //Dimension 2 - [63:31]
    disp(2);
  
```

**Table 24.3** Array querying system functions

Array querying function	Description
\$left (array_name, dimension)	Returns the left bound of the dimension. For a packed dimension, this is the index of the most significant element. For a queue or dynamic array dimension, \$left returns 0. Dimensions begin with number 1, starting from the left-most unpacked dimension, the dimension number continues with the left-most packed dimension and ends with the right-most packed dimension
\$right (array_name, dimension)	Returns the right bound of the dimension. Returns the LSB number of the specified dimension. Dimensions are numbered the same as with \$left. For a packed dimension, this is the index of the least significant element. For a queue or dynamic array dimension whose current size is zero, \$right will return -1
\$increment (array_name, dimension)	Returns 1 if \$left is greater than or equal to \$right and -1 if \$left is less than \$right. For a queue or dynamic array dimension, \$increment returns -1. Dimensions are numbered the same as with \$left
\$low (array_name, dimension)	Returns the lowest bit number of the specified dimension, which may be either the MSB or LSB. Dimensions are numbered the same way as with \$left. Returns the same value as \$left if \$increment returns -1 and the same value as \$right if \$increment returns 1
\$high (array_name, dimension)	Returns the highest bit number of the specified dimension, which may be either the MSB or LSB. Dimensions are numbered the same as with \$left. Returns the same value as \$right if \$increment returns -1 and the same value as \$left if \$increment returns 1
\$size (array_name, dimension)	Returns the total number of elements in the dimension, which is equivalent to: \$high - \$low + 1. Dimensions are numbered the same as with \$left
\$dimensions (array_name)	The total number of dimensions in the array (packed and unpacked, static or dynamic) 1 for the string data type or any other non-array type 0 for any other type
\$unpacked_dimensions (array_name)	The total number of unpacked dimensions for an array (static or dynamic) 0 for any other type

```

//Dimension 3 - [15:0]
disp (3);

//Dimension 4 - [7:0]
disp (4);
end

task disp (input int d);
    $display ("Dimension = %0d :: $left %0d $right %0d
$low %0d $high %0d $increment %0d $size %0d $dimensions %0d", d,
$left(arr,d),$right(arr,d),$low(arr,d),$high(arr,d),$increment(ar
r,d),$size(arr,d),$dimensions(arr) );
    endtask
endmodule

```

*Simulation log:*

```

Dimension = 1 :: $left 0 $right 31 $low 0 $high 31 $increment -1 $size 32
$dimensions 4
Dimension = 2 :: $left 63 $right 31 $low 31 $high 63 $increment 1 $size 33
$dimensions 4
Dimension = 3 :: $left 15 $right 0 $low 0 $high 15 $increment 1 $size 16
$dimensions 4
Dimension = 4 :: $left 7 $right 0 $low 0 $high 7 $increment 1 $size 8 $dimensions 4
V C S   S i m u l a t i o n   R e p o r t

```

In this example, we declare an array “arr” as follows:

bit [15:0] [7:0] arr [0:31] [63:31];

As we know from the discussion on arrays in a previous chapter, the dimensions of an array are determined from left to right on unpacked array elements and then from left to right on packed array elements. So:

[0:31] = dimension 1 – default dimension

[63:31] = dimension 2

[15:0] = dimension 3

[7:0] = dimension 4

This is important to understand because the “dimension” arguments of these querying system functions relate to these dimensions. Note that dimension 1 is the default dimension. So, if you do not specify a dimension, you will get results for dimension 1. But I strongly recommend specifying the dimension number explicitly.

We then display, for each dimension, the \$left, \$right, \$low, \$high, \$increment, \$size, and \$dimension system function values.

So, for dimension 1, ([0:31]), we get the following:

```

Dimension = 1 :: $left 0 $right 31 $low 0 $high 31 $increment -1 $size 32
$dimensions 4

```

Increment is “-1” because \$left is less than \$right. Total dimensions are 4.

Similarly, for dimensions 2, 3, and 4, we get results for that dimension.

## 24.7 Math Functions

SystemVerilog provides integer and real math functions. They are used in constant expressions.

### 24.7.1 Integer Math Functions

`$clog2(x)` is the integer math function. It returns the ceiling of the log base 2 of “x.” It rounds up the log value to an integer value. The argument can be an integer or an arbitrary sized vector value. The argument is treated as an unsigned number. An example is presented below.

### 24.7.2 Real Math Functions

The real math functions are described in Table 24.4. They accept a real value argument and return a real result type.

Here is an example showing some of the functions:

**Table 24.4** Real math functions

Real math function	Description
<code>\$ln(x)</code>	Natural log
<code>\$log10(x)</code>	Log base 10
<code>\$exp(x)</code>	Exponential
<code>\$sqrt(x)</code>	Square root
<code>\$pow(x,y)</code>	$x^{**}y$
<code>\$floor(x)</code>	Floor
<code>\$ceil(x)</code>	Ceiling
<code>\$sin(x)</code>	Sine
<code>\$cos(x)</code>	Cosine
<code>\$tan(x)</code>	Tangent
<code>\$asin(x)</code>	Arc-sine
<code>\$acos(x)</code>	Arc-cosine
<code>\$atan(x)</code>	Arc-tangent
<code>\$atan2(y, x)</code>	Arc-tangent of $y/x$
<code>\$hypot(x, y)</code>	$\sqrt{x^*x + y^*y}$
<code>\$sinh(x)</code>	Hyperbolic sine
<code>\$cosh(x)</code>	Hyperbolic cosine
<code>\$tanh(x)</code>	Hyperbolic tangent
<code>\$asinh(x)</code>	Arc-hyperbolic sine
<code>\$acosh(x)</code>	Arc-hyperbolic cosine
<code>\$atanh(x)</code>	Arc-hyperbolic tangent

```
module math;
    int i1, iR;
    real result;
    real r1, r2;

    initial begin
        i1 = 1234;
        iR = $clog2(i1);
        $display("$clog2(%0d) = %0d",i1,iR);

        r1 = 3.14;
        result = $ln(r1);
        $display("$ln(%0f) = %0f",r1,result);

        result = $sin(r1);
        $display("$sin(%0f) = %0f",r1,result);

        r1 = -1;
        result = $asin(r1);
        $display("$asin(%0f) = %0f",r1,result);

        result = $cos(r1);
        $display("$cos(%0f) = %0f",r1,result);

        r1 = 0.01;
        result = $acos(r1);
        $display("$acos(%0f) = %0f",r1,result);

        result = $tan(r1);
        $display("$tan(%0f) = %0f",r1,result);

        r2 = 0.02;
        result = $hypot(r1,r2);
        $display("$hypot(%0f,%0f) = %0f",r1,r2,result);

        r1 = 3.1; r2 = 2.1;
        result = $pow(r1, r2);
        $display("$pow(%0f, %0f) = %0f",r1,r2,result);

        result = $floor(r1);
        $display("$floor(%0f) = %0f",r1,result);

        result = $ceil(r1);
        $display("$ceil(%0f) = %0f",r1,result);
```

```

    end
endmodule

```

*Simulation log:*

```

$clog2(1234) = 11
$ln(3.140000) = 1.144223
$sin(3.140000) = 0.001593
$asin(-1.000000) = -1.570796
$cos(-1.000000) = 0.540302
$acos(0.010000) = 1.560796
$tan(0.010000) = 0.010000
$hypot(0.010000,0.020000) = 0.022361
$pow(3.100000, 2.100000) = 10.761172
$floor(3.100000) = 3.000000
$ceil(3.100000) = 4.000000

```

V C S   S i m u l a t i o n   R e p o r t

## 24.8 Bit-Vector System Functions

We have discussed the following bit-vector system functions in Sect. 14.19 (under the SystemVerilog Assertions chapter). There we saw how they are used in assertions. Here they are listed for the sake of completeness:

### \$countbits (expression, control bits)

Count the number of bits in an expression whose value match one of the control bits. For example, \$countbits (expression, '1) will count the number of 1's in the expression. \$countbits (expression, 'z) will count the number of z's (tri-state) in the expression. The control bits can be '1, '0, 'x, and 'z.

### \$countones (expression)

Count the number of 1's in an expression. Equivalent to \$countbits (expression, '1).

### \$onehot (expression)

Returns true if there at most one “1” in the expression, else return false.  
Equivalent to \$countbits (expression, '1) == 1.

### \$onehot0 (expression)

Returns true if there is at most one “1” or all “0”s in the expression, else return false. Equivalent to \$countbits (expression, '1) <= 1.

### \$isunknown (expression)

Returns true if any of the bits in the expression are “x” or “z,” else return false.  
Equivalent to \$countbits (expression, 'x, 'z) != 0.

Here is an example:

```

module bitV;
  logic [15:0] vec1;
  int i1;

  initial begin
    vec1 = 'h0xzf;
    i1 = $countbits(vec1, '1);
    $display("$countbits(%h, '1) = %0d",vec1,i1);

    i1 = $countbits(vec1, 'z, 'x);
    $display("$countbits(%h, 'z,'x ) = %0d",vec1,i1);

    i1 = $countbits(vec1, '0);
    $display("$countbits(%h, '0 ) = %0d\n",vec1,i1);

    i1 = $countones(vec1);
    $display("$countones(%h) = %0d \n",vec1,i1);

    vec1 = 'h0001;
    i1 = $onehot(vec1);
    if (i1)
      $display("PASS: $onehot(%h) = %0d\n",vec1,i1);
    else
      $display("FAIL: $onehot (%h) = %0d\n",vec1,i1);

    vec1 = 'hf000;
    i1 = $onehot0(vec1);
    if (i1)
      $display("PASS: $onehot0(%h) = %0d\n",vec1,i1);
    else
      $display("FAIL: $onehot0(%h) = %0d\n",vec1,i1);

    vec1 = 'h0XZf;
    i1 = $isunknown(vec1);
    if (i1)
      $display("PASS: $isunknown(%h) = %0d\n",vec1,i1);
    else
      $display("FAIL: $isunknown(%h) = %0d\n",vec1,i1);
  end
endmodule

```

*Simulation log:*

```

$countbits(0xzf,'1)=4
$countbits(0xzf,'z,'x )=8
$countbits(0xzf,'0 )=4

```

```
$countones(0xzf) = 4
PASS: $onehot(0001) = 1
FAIL: $onehot0(f000) = 0
PASS: $isunknown(0xzf) = 1
V C S   S i m u l a t i o n   R e p o r t
```

## 24.9 Severity System Tasks

These tasks are discussed in Sect. 14.14 (in the chapter SystemVerilog Assertions). Here they are listed again.

All of the severity system tasks print a tool-specific message, indicating the severity of the exception

condition and specific information about the condition. Each of the severity system tasks can include optional user-defined information to be reported. The user-defined message shall use the same syntax as the \$display system task and thus can include any number of arguments:

**\$fatal (finish\_number [,list\_of\_arguments]);**

Generates a run-time fatal error and the simulation terminates. The finish\_number is the same as for \$display task. It can be 0 or 1 or 2. It basically dictates the level of diagnostic information reported by the simulator.

**\$error ([list\_of\_arguments]);**

This denotes a run-time error. User may take (or not) a termination action based on this. For example, \$error("FIFO overflow. FIFO Count = %0d",fifocount);

**\$warning ([list\_of\_arguments]);**

This denotes a run-time warning.

**\$info ([list\_of\_arguments]);**

Information only message. Mainly for debug purpose.

## 24.10 \$random and Probabilistic Distribution Functions

### 24.10.1 \$random

*Important Note: \$random is present in the language only for backward compatibility. It is now replaced by \$urandom. I am presenting \$random just for the sake of completeness. \$urandom function is described in Sect. 13.11.2.*

Its syntax is:

```
$random [ (seed) ];
```

It basically generates pusedo-random numbers. It returns a new 32-bit random number each time it is called. The random number generated is a signed integer. The optional “seed” argument allows you to generate different random streams. It also means the same sequence of random numbers will be returned for the same seed.

For example, you may pass “seed” as a command line argument to your regression runs and provide a different “seed” for each run, thereby generating different random streams of data with each regression.

One of the useful ways to restrict the range of random numbers generated is to use the “mod” (%) operator. For example:

```
int rVal;
rVal = $random % 10;
```

This will generate random numbers between –9 and 9.

If you want only the positive numbers, you have to enclose \$random into {} brackets:

```
rVal = {$random} % 10;
```

This will generate random numbers from 0 to 9.

### **24.10.2 Probabilistic Distribution Functions**

The following probabilistic distribution functions are provided by the language:

```
$dist_uniform (seed, start, end)
$dist_normal (seed, mean, standard_deviation)
$dist_exponential (seed, mean)
$dist_poisson (seed, mean)
$dist_chi_square (seed, degree_of_freedom)
$dist_t (seed, degree_of_freedom)
$dist_erlang (seed, k_stage, mean)
```

Each of these functions returns a pseudo-random number. For example, \$dist\_uniform returns a random number uniformly distributed within the interval specified by the “start” and “end” limit numbers. Seed is used to generate different random streams with each new “seed” number. It also means the same random numbers will be returned for the same seed.

For \$dist\_uniform, the “start” value should be smaller than the “end” value.

The following is a simple example, showing the functions \$dist\_uniform, \$dist\_normal, and \$dist\_poisson:

```

module distF;
    int i1, ril, sil;

initial begin
    ril = 1234;
    for (int n = 0; n <=3; n++) begin
        i1 = $dist_uniform(ril, 0, 10);
        $display("$dist_uniform(1234,0,10) = %0d",i1);
    end

    for (int n = 0; n <=3; n++) begin
        i1 = $dist_normal(ril, 4, 20);
        $display("$dist_normal(1234,4,20) = %0d",i1);
    end

    for (int n = 0; n <=3; n++) begin
        i1 = $dist_poisson(ril,10);
        $display("$dist_poisson(1234,10) = %0d",i1);
    end
end
endmodule

```

*Simulation log:*

```

$dist_uniform(1234,0,10) = 0
$dist_uniform(1234,0,10) = 6
$dist_uniform(1234,0,10) = 1
$dist_uniform(1234,0,10) = 2

$dist_normal(1234,4,20) = -7
$dist_normal(1234,4,20) = -1
$dist_normal(1234,4,20) = 34
$dist_normal(1234,4,20) = -24

$dist_poisson(1234,10) = 11
$dist_poisson(1234,10) = 8
$dist_poisson(1234,10) = 5
$dist_poisson(1234,10) = 17

```

V C S   S i m u l a t i o n   R e p o r t

## 24.11 Queue Management Stochastic Analysis Tasks

SystemVerilog provides a set of system tasks and functions that manage queues. They describe implementation of stochastic queuing models.

### 24.11.1 \$q\_initialize

\$q\_initialize creates a new queue. Its syntax is:

```
$q_initialize(q_id, q_type, max_length, status);
```

where:

*q\_id* is the queue ID and is an integer input that uniquely identifies the new queue.  
*q\_type* is an integer input which can have value 1 or 2:

- *q\_type* = 1: First-in, first-out queue
- *q\_type* = 2: Last-in, first-out queue

*max\_length* is an integer input that specifies the maximum number of entries allowed in a queue.

*status* is the status of success or failure from a queue task. The status numbers have the following meaning (Table 24.5).

We will see an example on how \$q\_initialize works in later section.

### 24.11.2 \$q\_add

\$q\_add adds an entry on the queue. Its syntax is:

```
$q_add(q_id, job_id, inform_id, status);
```

where:

*q\_id* is the integer input that indicates which queue to add this entry to.  
*job\_id* argument identifies the job.

*inform\_id* is an integer input that is associated with the queue entry. Its meaning is user-defined.

*status* is described in Table 24.5.

We will see an example of how \$q\_add works in a later section.

### 24.11.3 \$q\_remove

The \$q\_remove removes/retrieves an entry from the queue. The syntax is:

```
$q_remove( q_id, job_id, inform_id, status);
```

where:

**Table 24.5** Status code for queue system tasks

Status code	Description
0	Queue creating successful
1	Queue full cannot add new elements
2	Undefined q_id
3	Queue empty cannot remove an element
4	Unsupported queue type
5	Specified length <= 0 cannot create queue
6	Duplicate q_id cannot create queue
7	Not enough memory cannot create queue

*q\_id* is the integer input that identifies the queue from which the entry is removed.  
*job\_id* is the integer output that identifies the entry being removed.  
*inform\_id* is the integer output that we stored during \$q\_add call.  
*status* is described in Table 24.5.

#### 24.11.4 \$q\_full

This function checks where there is room for another entry on the queue. It returns 0 when the queue is not full and 1 when the queue is full. Syntax is:

```
$q_full(q_id, status);
```

*q\_id* is the queue ID.  
*status* is described in Table 24.5.

#### 24.11.5 \$q\_exam

\$q\_exam provides statistical information about activity on the *q\_id*. Here is the syntax:

```
$q_exam (q_id, q_stat_code, q_stat_value, status);  
where:
```

*q\_id* is the ID of the queue.  
*q\_stat\_code*. This is a code based on which the *q\_stat\_value* is returned by the task.  
 Table 24.6 defines the relation value requested in *q\_stat\_code* and corresponding *q\_stat\_value*.  
*status* is described in Table 24.5.

**Table 24.6** q\_stat\_code and q\_stat\_value of task \$q\_exam

q_stat_code	q_stat_value
1	Current queue length
2	Mean interarrival time
3	Max queue length
4	Shortest wait time
5	Longest wait time for jobs still in queue
6	Average wait time in queue

#### 24.11.6 Example of Queue Management Stochastic Analysis Tasks and Functions

```
module qu;
    int status, qfstatus, q_stat_value;

    initial begin
        $q_initialize(1,1,1,status);
        $display("$q_initialize");
        check_status;

        $q_add(1, 1, 20, status);
        $display("$q_add - First entry");
        check_status;

        $q_add(1, 1, 20, status);
        $display("$q_add - Second entry");
        check_status;

        $q_remove(1, 1, 20, status);
        $display("$q_remove - remove an entry");
        check_status;

        $q_remove(1, 1, 20, status);
        $display("$q_remove an entry");
        check_status;

        qfstatus = $q_full(1, status);
        $display("$q_full status = %0d",qfstatus);

        $q_add(2, 1, 20, status);
        $display("$q_add - add to an undefined queue");
        check_status;
```

```
$q_initialize(1,1,1,status);
$display("$q_initialize with same id");
check_status;

$q_initialize(1,1,0,status);
$display("$q_initialize with zero length");
check_status;

$q_exam(1, 1, q_stat_value, status);
$display("$q_exam:q_stat_code = 1: Current queue length = %0d",q_stat_value);

$q_exam(1, 2, q_stat_value, status);
$display("$q_exam:q_stat_code = 2: Mean Interarrival time = %0d",q_stat_value);

$q_exam(1, 3, q_stat_value, status);
$display("$q_exam:q_stat_code = 3: Max. queue length = %0d",q_stat_value);

$q_exam(1, 4, q_stat_value, status);
$display("$q_exam:q_stat_code = 4: Shortest wait time ever = %0d",q_stat_value);

$q_exam(1, 5, q_stat_value, status);
$display("$q_exam:q_stat_code = 5: Longest wait time for jobs still in queue = %0d",q_stat_value);

$q_exam(1, 6, q_stat_value, status);
$display("$q_exam:q_stat_code = 6: Average wait time = %0d",q_stat_value);
end

task check_status;
  if (status == 0)
    $display("\tStatus = Success");
  else if (status == 1)
    $display("\tStatus = Queue full, cannot add");
  else if (status == 2)
    $display("\tStatus = Undefined q_id");
  else if (status == 3)
    $display("\tStatus = Queue empty, cannot remove");
  else if (status == 4)
    $display("\tStatus = Unsupported queue type, cannot create queue");
```

```

        else if (status == 5)
            $display("\tStatus = Specified length <= 0, cannot
create queue");
        else if (status == 6)
            $display("\tStatus = Duplicate q_id, cannot create
queue");
        else if (status == 7)
            $display("\tStatus = Not enough memory, cannot create
queue");
    endtask
endmodule

```

*Simulation log:*

```

$q_initialize
    Status = Success
$q_add - First entry
    Status = Success
$q_add - Second entry
    Status = Queue full, cannot add
$q_remove - remove an entry
    Status = Success
$q_remove an entry
    Status = Queue empty, cannot remove
$q_full status = 0
$q_add - add to an undefined queue
    Status = Undefined q_id
$q_initialize with same id
    Status = Duplicate q_id, cannot create queue
$q_initialize with zero length
    Status = Specified length <= 0, cannot create queue
$q_exam:q_stat_code = 1: Current queue length = 0
$q_exam:q_stat_code = 2: Mean Interarrival time = 0
$q_exam:q_stat_code = 3: Max. queue length = 1
$q_exam:q_stat_code = 4: Shortest wait time ever = 0
$q_exam:q_stat_code = 5: Longest wait time for jobs still in queue = 0
$q_exam:q_stat_code = 6: Average wait time = 0

```

V C S   S i m u l a t i o n   R e p o r t

Let us take the example step by step.

First, we create a new queue using \$q\_initialize with a q\_id of 1, q\_type = 1(meaning this is a FIFO-type queue), max\_length of 1, and return status provided in the “int status.” Here is the code for that:

```
$q_initialize(1,1,1,status);
$display("$q_initialize");
check_status;
```

We check the status to see if the queue was successfully created by checking the “status” output from the task. It was indeed successfully created as shown in the simulation log:

```
$q_initialize
Status = Success
```

Next, we add an entry to the queue we just created: Q\_id = 1, Job\_id = 1, inform\_id = 20.

```
$q_add(1, 1, 20, status);
$display("$q_add - First entry");
check_status;
```

Then we check the status to see that we were indeed able to add an entry to the queue. Here is the simulation log:

```
$q_add - First entry
Status = Success
```

We had success!

Then we add another entry to the queue. So, this will be the second entry. Here is the code to do that:

```
$q_add(1, 1, 20, status);
$display("$q_add - Second entry");
check_status;
```

Simulation log shows that this operation was – not – successful because we had created a queue of length 1. So, the second entry is an overflow. Hence, we get the following simulation log:

```
$q_add - Second entry
Status = Queue full, cannot add
```

Then we remove an entry from the queue:

```
$q_remove(1, 1, 20, status);
$display("$q_remove - remove an entry");
check_status;
```

This operation was successful as shown in the simulation log.:

**\$q\_remove - remove an entry**  
Status = Success

Then, we try to remove one more entry:

```
$q_remove(1, 1, 20, status);
$display("$q_remove an entry");
check_status;
```

But since the queue was only of length 1, and we had already removed 1 entry, this second attempt will be a failure as shown in simulation log:

**\$q\_remove an entry**  
Status = Queue empty, cannot remove

Then we check to see if the queue is full or empty:

```
qfstatus = $q_full(1, status);
$display("$q_full status = %0d", qfstatus);
```

Since the queue is empty, we get the status from \$q\_full that the queue is not full. Here is the simulation log showing that:

**\$q\_full status = 0**

Then, we try to add an entry to a queue that has not yet been created:

```
$q_add(2, 1, 20, status);
$display("$q_add - add to an undefined queue");
check_status;
```

And the status reflects that the queue is undefined as shown in the simulation log:

**\$q\_add - add to an undefined queue**  
Status = Undefined q\_id

Then, we try to create a new queue with the same q\_id ("1") that we used when we created the original queue:

```
$q_initialize(1,1,1,status);
$display("$q_initialize with same id");
check_status;
```

That is not allowed as shown in the simulation log:

**\$q\_initialize with same id**  
Status = Duplicate q\_id, cannot create queue

Then, we try to initialize our queue with “0” length. That is not allowed:

```
$q_initialize(1,1,0,status);
$display("$q_initialize with zero length");
check_status;
```

Simulation log reflects that this is not allowed:

```
$q_initialize with zero length
Status = Specified length <= 0, cannot create queue
```

Then, we use \$q\_exam and provide it with different q\_stat\_code to get corresponding q\_stat\_value. Here is the code that does that:

```
$q_exam(1, 1, q_stat_value, status);
$display("$q_exam:q_stat_code = 1: Current queue length
= %0d",q_stat_value);

$q_exam(1, 2, q_stat_value, status);
$display("$q_exam:q_stat_code = 2: Mean Interarrival
time = %0d",q_stat_value);

$q_exam(1, 3, q_stat_value, status);
$display("$q_exam:q_stat_code = 3: Max. queue length =
%0d",q_stat_value);

$q_exam(1, 4, q_stat_value, status);
$display("$q_exam:q_stat_code = 4: Shortest wait time
ever = %0d",q_stat_value);

$q_exam(1, 5, q_stat_value, status);
$display("$q_exam:q_stat_code = 5: Longest wait time
for jobs still in queue = %0d",q_stat_value);

$q_exam(1, 6, q_stat_value, status);
$display("$q_exam:q_stat_code = 6: Average wait time =
%0d",q_stat_value);
```

Here is the simulation log that we get for above \$q\_exam queries:

```
$q_exam:q_stat_code = 1: Current queue length = 0
$q_exam:q_stat_code = 2: Mean Interarrival time = 0
$q_exam:q_stat_code = 3: Max. queue length = 1
$q_exam:q_stat_code = 4: Shortest wait time ever = 0
$q_exam:q_stat_code = 5: Longest wait time for jobs still in queue = 0
$q_exam:q_stat_code = 6: Average wait time = 0
```

# Chapter 25

## I/O System Tasks and Functions



**Introduction** In this chapter we will discuss:

1. Display tasks
2. File I/O tasks and functions
3. Memory load tasks
4. Memory dump tasks
5. Command line input
6. VCD tasks

In this section we will see:

1. Display tasks
2. File I/O tasks and functions
3. Memory load tasks
4. Memory dump tasks
5. Command line input
6. VCD tasks

### 25.1 Display Tasks

We have seen \$display tasks throughout this book. There are four different types of display tasks, namely, \$display, \$write, \$strobe, and \$monitor, which are available in the language.

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_25](https://doi.org/10.1007/978-3-030-71319-5_25)) contains supplementary material, which is available to authorized users.

**\$display** – Displays information in a given format and *displays information at the current time unit*. It automatically adds a new line at the end of the display.

**\$write** – Same as \$display, except that it *does not add a new line at the end of the display*.

**\$strobe** – Same as \$display, except that it *displays information at the end of the time step*, when all simulation events have occurred for that simulation time, just before the simulation time is advanced.

**\$monitor** – Same as \$strobe, except that the simulator sets up a mechanism whereby *each time a variable or an expression in the argument list changes value, the entire argument list is displayed* at the *end of the time step*. This is also called continuous monitoring.

Table 25.1 describes all the simulation display tasks.

Let us look at an example showing use-cases of these tasks:

**Table 25.1** Display tasks

Display task	Description
\$display	Displays information in a given format and <i>displays information at the current time unit</i> . It automatically adds a new line at the end of the display
\$displayb	Displays information in binary format
\$displayh	Displays information in hex format
\$displayo	Displays information in octal format
\$strobe	Same as \$display, except that it <i>displays information at the end of the time step</i> , when all simulation events have occurred for that simulation time, just before the simulation time is advanced
\$strobeb	Displays information in binary format
\$strobeh	Displays information in hex format
\$strobeo	Displays information in octal format
\$write	Same as \$display, except that it <i>does not add a new line at the end of the display</i>
\$writeb	Displays information in binary format
\$writeh	Displays information in hex format
\$writeo	Displays information in octal format
\$monitor	Same as \$display except that the simulator sets up a mechanism whereby <i>each time a variable or an expression in the argument list changes value, the entire argument list is displayed</i> at the <i>end of the time step</i> . This is also called continuous monitoring
\$monitorb	Displays information in binary format
\$monitorh	Displays information in hex format
\$monitoro	Displays information in octal format
\$monitoroff	Interactively turns <i>off</i> \$monitor
\$monitoron	Interactively turns <i>on</i> \$monitor

```
module disp;
    bit [7:0] i1;

initial begin // $display tasks
    i1 = 12;
    $display("Display Decimal i1 = ", i1);
    $displayb ("Display binary i1 = ", i1);
    $displayh ("Display hex i1 = ", i1);
    $displayo ("Display octal i1 = ", i1);
end

initial begin // $strobe tasks
    #1;
    i1 = 12;

    $strobe("Strobe Decimal i1 = ", i1);
    $strobeb ("Strobe binary i1 = ", i1);
    $strobeh ("Strobe hex i1 = ", i1);
    $strobeo ("Strobe octal i1 = ", i1);

    i1 = 23;
    i1 = 34; // $strobe will display this last value of 'i1'
end

initial begin // $write tasks
    #2;
    i1 = 45;
    $write("Write Decimal i1 = ", i1);
    $writeb (" :: Write binary i1 = ", i1);
    $writeh (" :: Write hex i1 = ", i1);
    $writeo (" :: Write octal i1 = ", i1);
end

initial begin
    $monitor ("\n Monitor decimal i1 = ", i1); // Start
$monitor
end

initial begin
    $monitoroff; // turn OFF $monitor
    #2 $monitoron; // turn ON $monitor
end
endmodule
```

*Simulation log:*

```
run -all;
# KERNEL: Display Decimal i1 = 12
# KERNEL: Display binary i1 = 000001100
# KERNEL: Display hex i1 = 0c
# KERNEL: Display octal i1 = 014
# KERNEL: Strobe Decimal i1 = 34
# KERNEL: Strobe binary i1 = 001000010
# KERNEL: Strobe hex i1 = 22
# KERNEL: Strobe octal i1 = 042
# KERNEL: Write Decimal i1 = 45 :: Write binary i1 = 00101101 :: Write hex i1
= 2d :: Write octal i1 = 055
# KERNEL: Monitor decimal i1 = 45
# KERNEL: Simulation has finished.
```

The \$display tasks display “i1” in all different formats. \$strobe is interesting. We do the following for \$strobe:

```
initial begin // $strobe tasks
    #1;
    i1 = 12;
    $strobe("Strobe Decimal i1 = ", i1);
    $strobeb ("Strobe binary i1 = ", i1);
    $strobeh ("Strobe hex i1 = ", i1);
    $strobeo ("Strobe octal i1 = ", i1);
    i1 = 23;
    i1 = 34; // $strobe will display this last value of 'i1'
end
```

Note that first we assign “i1 = 12;” at time 1. Then we invoke \$strobe (for all formats). Then, we assign “i1 = 23;” and then “i1 = 34;” at the same time 1. Since \$strobe displays values of variables at the end of the time unit, it will pick the last assignment (“i1 = 34;”) and display that value of “i1” in all different formats.

For \$monitor, we invoke it at time 0. But then we turn it off immediately at time 0. We then turn it *on* at time 2. So, the changes in variables at times 0 and 1 will not be displayed. Hence, we see it only display the value of “i1” (=45) a time 2.

## 25.2 Escape Identifiers

We have seen the escape identifier “\n” for new line. There are many other escape identifiers available for displaying special characters. They are described in Table 25.2.

**Table 25.2** Escape identifiers

Escape identifier	Character produced
\n	newline
\t	tab
\\\	\ character
\\"	" character
\v	vertical tab
\f	form feed
\a	bell
\%%	% character
\ddd	A character specified in 1–3 octal digits. If fewer than three characters are used, the following character will not be an octal digit. Implementations may issue an error if the character represented is greater than \377
\xdd	A character specified in two hex digits

Here is a simple example using some of the escape identifiers:

```
module esc;
initial begin
    $display("Newline \n");
    $display("\t tab");
    $display("\\");
    $display("\\\"");
    $display("%%");
end
endmodule
```

#### *Simulation log:*

```
Newline
tab
\
"
%
```

## 25.3 Format Specifications

We have seen format specifiers %d, %h, %b, %s, etc. throughout the book. There are other such format specifiers also available. These are shown in Table 25.3.

Here is an example showing some of the format specifiers:

**Table 25.3** Format specifiers for display statements

Format specifier	Description
%h or %H	Hexadecimal format
%d or %D	Decimal format
%o or %O	Octal format
%b or %B	Binary format
%c or %C	ASCII character format
%l or %L	Library binding information
%v or %V	Display net signal strength
%m or %M	Display hierarchical name to the instance
%p or %P	Display as an assignment pattern
%s or %S	String format
%t or %T	Time format
%u or %U	Binary file I/O. Unformatted two value data. It is defined for writing data without formatting (binary values). Any unknown ("x") or tri-state ("z") bits in the source are treated as zero. Not defined for unpacked arrays
%z or %Z	Binary file I/O. Unformatted four value data. "x" and "z" bits are preserved. Not defined for unpacked arrays
%e or %E	Display real numbers in an exponential format
%f or %F	Display real numbers in decimal format
%g or %G	Display real numbers in exponential or decimal format, whichever format results in the shorter printed output

```

module top;
    form fr1 ( );
endmodule

module form;
    int i1, f1;
    real r1;
    string s1;
    int carray[ ];
    wire w1;
    supply1 w2;

    initial begin
        i1 = 1234;
        r1 = 3.142;
        s1 = "hello";
        carray = new [5];
        carray = '{4,5,6,7,8};
        f1 = $fopen("outFile");
    end

```

```

$display("Character s1 = %c",s1);
$display("String s1 = %s",s1);

$display("Hierarchical name = %m");

$display("carray = %p", carray);

$display("Real r1 = %f",r1);
$display("Real r1 in exponential format = %e",r1);
$display("Real r1 in expo or decimal format = %g",r1);

$display("wire w1 = %v",w1);
$display("supply1 wire w2 = %v",w2);

i1 = 'h0xz1;
$display ("Unformatted 2 value i1 = %u", i1);
$fwrite(f1,"Unformatted 2 value i1 = %u",i1);
end
endmodule

```

*Simulation log:*

Character s1 = h  
 String s1 = hello  
 Hierarchical name = top.fr1  
 carray = '{4, 5, 6, 7, 8}  
 Real r1 = 3.142000  
 Real r1 in exponential format = 3.142000e+00  
 Real r1 in expo or decimal format = 3.142  
 wire w1 = HiZ  
 supply1 wire w2 = Su1  
 Unformatted 2 value i1 = 00000001

**V C S   S i m u l a t i o n   R e p o r t**

Note the difference between `%c` and `%s`. `%c` displays only one character from the string, while `%s` displays the entire string.

`%m` displays the hierarchical name of the instance where `$display` is called from. This is an excellent debug feature. For highly hierarchical designs, it would be necessary to know which `$display` is executed at what hierarchical level.

Note the display of real number “`r1`” in three different formats (`%f`, `%e`, and `%g`). Note `%v` that displays net signal strength.

For “`wire w1`,” the net strength is `HiZ` (since the net is not driven), while for the net “`supply1 w2`” (net is attached to `supply1` power rail), the strength is “`Su1`.” `%v` displays the strength. The following strength mnemonics are used:

- Su – supply drive – e.g., Supply0 (Su0) or Supply1 (Su1)
- St – strong drive – e.g., St1 for strong1, StX for strong driving unknown
- Pu – pull drive – e.g., Pu0 for a pull driving 0 value
- La – large capacitor – associated with *trireg* nets
- We – weak drive
- Me – medium capacitor – associated with *trireg* nets
- Sm – small capacitor – associated with *trireg* nets
- HiZ – high impedance

The %u format is used to produce an output stream to be output to a file. It performs binary file I/O. The file “outFile” will have “i1” written as ‘b00000001, since %u converts “x” and “z” to zero.

## 25.4 File I/O System Tasks and Functions

SystemVerilog provides system tasks and function that allow to open/close files, read from files, and write to them, system tasks that output values into variables, etc.

Having opened a file, what can you do with it? You can write to it and you can read from it.

To write to a file, you use \$fdisplay, \$fwrite, \$fmonitor, and \$fstroke (write formatted data to a text file).

To read from a file, you use the following system tasks:

- Read formatted data from a text file using \$fscanf (system function).
- Read (\$sscanf) and write (\$swrite) from and to a string variable.
- Read a text file one line at a time (\$fgets).
- Read a text file one character at a time (\$fgetc).
- Perform random file access (\$fseek, \$rewind).
- Perform binary file I/O using %u (2-bit) and %z (4-bit) format specification.

Let us look at each in upcoming sections.

### 25.4.1 \$fopen and \$fclose

The function \$fopen opens a file and \$fclose closes a file that was opened. Syntax is:

```
multi_channel_descriptor = $fopen (filename);
file_descriptor = $fopen(filename, type);
fclose(multi_channel_descriptor);
fclose (file_descriptor);
```

\$fopen returns a 32-bit integer value.

Filename is a string literal and “type” is also a string literal.

When you call \$fopen(filename); (i.e., without specifying the “type”), it returns the so-called multi\_channel\_descriptor (mcd). The mcd value has a single bit set for each file opened. Since it sets only a single bit, two or more “mcd’s can be bitwise OR-ed together to represent multiple files (output is directed to two or more files opened with an mcd). Bit 0 represents the simulator’s output display (standard output) and, if open, the simulator’s output log file. The MSB (bit 31) of a multichannel descriptor is reserved and will always be cleared, limiting an implementation to at most 31 files opened for output via “mcd’s. You can only write to multiple files; you cannot read from multiple files at the same time.

When you call \$fopen(filename, type) (i.e., with type specification), it returns the so-called file\_descriptor (fd). “fd” is a 32-bit packed array value and can be considered a single-channel file descriptor. Because “fd” has multiple bits set with each file opened/closed, unlike multichannel descriptors, file descriptors cannot be combined via bitwise OR in order to direct output to multiple files. When writing a file opened with an “fd,” only a single file maybe written at one time. Instead, files are opened via file descriptor for input, output, and both input and output, as well as for append operations, based on the value of file “type” as described in Table 25.4. The MSB (bit 31) of an “fd” is reserved and is always set. Three “fd” values are reserved: fd = 32'h8000\_0000 represents STDIN, fd = 32'h8000\_0001 represents STDOUT, and fd = 32'h8000\_0002 represents STDERR.

\$fclose closes the file specified by “fd” or files specified by “mcd.”

Here is an example of \$fopen and \$fclose:

**Table 25.4** File types used with \$fopen

file_type	Description
“r”	Open a text file for reading
“rb”	Open a binary file for reading
“w”	Create a text file for writing
“wb”	Create a binary file writing
“a”	Open a text file for writing at the end of the file (append only)
“ab”	Open a binary file for writing at the end of the file (append only)
“r+”	Open a text file for update (reading and writing)
“r+b” or “rb+”	Open a binary file for update (reading and writing)
“w+”	Create a new text file for update (reading and writing)
“w+b” or “wb+”	Create a new binary file for update (reading and writing)
“a+”	Open a text file for update at the end of the file (append only)
“a+b” or “ab+”	Open a binary file for update at the end of the file (append only)

```

module files;
int mcd, fd;

initial begin

mcd = $fopen("fd1_file");
if (mcd) $display("fd1 open :: mcd = %b",mcd);
else $display("File fd1_file cannot be opened");

mcd = $fopen("fd2_file");
$display("fd2 open :: mcd = %b",mcd);

mcd = $fopen("fd3_file");
$display("fd3 open :: mcd = %b",mcd);

mcd = $fopen("fd4_file");
$display("fd4 open :: mcd = %b",mcd);

fd = $fopen("fd5_file", "w");
$display("fd5 open for write :: fd = %b",fd);

fd = $fopen("fd6_file", "w");
$display("fd6 open for write :: fd = %b",fd);

fd = $fopen("fd7_file", "w");
$display("fd7 open for write :: fd = %b",fd);

fd = $fopen("fd5_file","r"); //open for reading
$display("fd5 open for read :: fd = %b",fd);

$fclose(fd);
$fclose(mcd);

end
endmodule

```

*Simulation log:*

## VCS Simulation Report

We open multiple files using `$fopen`. First, we open files using `$fopen` without the “type” specification. This returns a `multi_channel_descriptor` (`mcd`). As we discussed above, only a single bit of “`mcd`” is set with each open of a new file. Note that ‘32h0000\_0001’ is reserved for `STDOUT`, so the file numbering starts from 32’h0000\_0002. This is evident from the simulation log for the first four files opened:

Next, we open four more files using `$fopen(filename, type)`. Since a “type” is specified with this `$fopen`, a `file_descriptor` (`fd`) is returned which will set multiple bits in an incremental order. As noted above, bits `8'h1000_0001` and `8'h1000_002` are always pre-fixed for `STDOUT` and `STDERR`. So, the `fd` numbering starts from `8'h1000_0003`. This is shown in the simulation log:

We finally close the files with \$fclose.

Here is an example of how we can OR together “mcd’s and use these file output tasks to broadcast to multiple files at the same time:

```

module files;
  int mcd1, mcd2, mcd3;
  int broadcast, sendOut;
  int bus;

initial begin

  mcd1 = $fopen("PCI_file");
  $display("PCI_file = %b",mcd1);

  mcd2 = $fopen("PCIe_file");
  $display("PCIe_file = %b",mcd2);

  mcd3 = $fopen("AXI_file");
  $display("AXI_file = %b",mcd3);

  sendOut = mcd1 | mcd2 | mcd3; //OR of 'mcd's

```

```
broadcast = 1 | sendOut; //OR with STDOUT
$display("broadcast = %b", broadcast);

$fdisplay (broadcast, "Bus = %h", bus);

#10; bus = 'hffff_ffff;
$fstrobe (broadcast, "at %0t Bus = %h", $stime, bus);

d
module
```

*Simulation log:*

In this example, we open three files, each returning an “mcd” (mcd1, mcd2, mcd3). We then OR all these three mcds into one integer called “sendOut”:

```
sendOut = mcd1 | mcd2 | mcd3; //OR of 'mcd's
```

We then OR the STDOUT (= 32'h0000\_0001) to “sendOut” so that we can send the output to both OR-ed files and STDOUT:

```
broadcast = 1 | sendOut; //OR with STDOUT
```

As you notice the OR value of “broadcast” is as shown in the simulation log (OR of all mcds and the STDOUT):

We finally use \$fdisplay and \$fstrobe to write to “broadcast” mcd:

```
$fdisplay (broadcast, "Bus = %h", bus);
#10; bus = 'ffff_ffff;
$fstrobe (broadcast, "at %0t Bus = %h", $stime, bus);
```

Which results in the simulation log:

Bus = 00000000  
at 10 Bus = ffffffff

### ***25.4.2 \$fdisplay, \$fwrite, \$fmonitor, and \$fstrobe***

SystemVerilog provides file output system tasks such as:

- \$fdisplay, \$fdisplayb, \$fdisplayh, and \$fdisplayo
- \$fwrite, \$fwriteb, \$fwriteh, and \$fwriteo
- \$fmonitor, \$fmonitorb, \$fmonitorh, and \$fmonitoro
- \$fstrobe, \$fstrobeb, \$fstrobeh, and \$fstrobeo

The syntax is:

```
file_output_task (multi_channel_descriptor [, list_of_arguments]);
      file_output_task (file_descriptor [,list_of_arguments]);
```

The tasks \$fdisplay, \$fwrite, \$fmonitor, and \$fstrobe are identical to \$display, \$write, \$monitor, and \$strobe, except that the file output tasks write to file(s). As noted above, each task has a “binary,” “hex,” and “octal” counterpart.

### ***25.4.3 \$swrite and \$sformat***

SystemVerilog allows formatting data to a string. There is the \$swrite family of tasks and there is the \$sformat task.

The first argument to \$swrite is a string data type (or a variable of integral, unpacked array of byte) to which the resulting string is written. The syntax is:

```
$swrite (output_var [, list_of_arguments]);
```

There are also the binary, hex and octal \$swrite tasks, namely, \$swriteb, \$swritesh, and \$swriteo.

\$sformat is similar to \$swrite, except that it interprets its second argument, as a format string. It only allows one format string. The format argument can be a string literal. Syntax is:

```
$sformat (output_var, format_string [,list_of_arguments]);
```

Also, there is the function \$sformatf, which is similar to \$sformat, except that the string result is passed back as the result value of the function and not placed in the first argument as for \$sformat.

Here is an example:

```

module files;
    int bus;
    string s1, s2, instance_name;

    initial begin
        //Difference between $swrite and $sformat
        //See $swrite - multiple arguments
        //See $sformat - only one argument is allowed -
        //else compile Error. See below

        $swrite(s1,"Hello there at time %0t",$stime, " Hello
again");
        $display("s1 = %s",s1);

        bus = 'hf0f0_f0f0;
        $sformat(s1,"bus = %h", bus);
        $display("s1 = %s",s1);
        //$sformat(s1,"bus = %h", bus," hello"); //Compile Error

        for(int i=0; i < 2; i++) begin
            $sformat(instance_name, "slave[%0d]", i);
            $display("instance_name = %s", instance_name);
        end

        #1;
        s2 = $sformatf ("time is %0t",$stime);
        $display("s2 = %s",s2);
    end
endmodule

```

*Simulation log:*

s1 = Hello there at time 0 Hello again

s1 = bus = f0f0f0f0

instance\_name = slave[0]

instance\_name = slave[1]

s2 = time is 1

#### V C S   S i m u l a t i o n   R e p o r t

The \$swrite tasks write the formatted string with two format strings – \$swrite(s1,"Hello there at time %0t",\$stime, " Hello again"); . Hence, we get the simulation output as follows:

s1 = Hello there at time 0 Hello again

In contrast, \$sformat (s1, "bus = %h", bus); takes the entire string “bus = %h” as its format string and assigns it to the string “s1.” But it can take only

one format string. Hence, we get the simulation output as follows (contrast that with \$swrite):

```
s1 = bus = f0f0f0f0
```

With \$sformat if you did the following (two format strings), you'll get a compile error:

```
$sformat(s1,"bus = %h", bus," hello");
```

Here is the compile error from Synopsys – VCS:

Warning-[STASKW\_SFRTMATR] More arguments than required  
testbench.sv, 15

Number of arguments (2) passed to place in the format string are more than the number of format specifiers (1).

But if you specified more than one format specifier within one format string, that would work. So, the following will work:

```
$sformat(s1,"bus = %h %s", bus, " hello");
```

To further explain \$sformat, look at the following loop:

```
for(int i=0; i < 2; i++) begin
    $sformat(instance_name, "slave[%0d]", i);
    $display("instance_name = %s", instance_name);
end
```

As you can see, the string “slave[%0d]” is assigned to “instance\_name.” The simulation log shows how the string “instance\_name” gets formatted:

```
instance_name = slave[0]
instance_name = slave[1]
```

Finally, we are using \$sformatf which is the same as \$sformat, except that \$sformatf is a function that returns a value. In our case, it returns the value to string var “s2.” Here is the code for that:

```
s2 = $sformatf ("time is %0t",$stime);
```

The simulation log shows the value returned by \$sformatf:

```
S2 = time is 1
```

## 25.5 Reading Data from a File

### 25.5.1 \$fgetc, \$ungetc, and \$fgets

\$fgetc lets you read a single character (byte) from a file. \$ungetc inserts a character into a file. \$fgets let you read a line from a file. In the following, \$fgetc will get a byte from the file specified by the file descriptor “fd”:

```
integer c3;
c3 = $fgetc(fd);
```

During \$fgetc, if an error occurs reading from the file, then “c3” will be set to (end of file) EOF (-1).

The following is pseudo code for \$ungetc:

```
integer c2;
c2 = $ungetc("s", fd);
```

\$ungetc inserts the character specified by “c2” into the file descriptor “fd” (pushes the character back into the file stream). A subsequent \$fgetc will extract the character inserted by \$ungetc. The file itself remains unchanged. If an error occurs, then “c2” is set to EOF.

Here is an example:

```
module files;
  int fd;
  integer c2, c3;

  initial begin
    fd = $fopen("fd1_file", "w");
    if (fd) $display("fd1_file opened for write");
    else $display("File fd1_file cannot be opened");

    c2 = $ungetc("s", fd); //insert character "s" into 'fd'

    c3 = $fgetc(fd);
    $display("Character read from file = %s", c3);

    $fclose(fd);
  end
endmodule
```

*Simulation log:*

```
fd1_file opened for write
Character read from file = s
V C S  Simulation Report
```

First, we open the file “fd1\_file” for writing (file descriptor “fd”). Then, we insert character “s” into the file using \$ungetc. And then, we read that character back using \$fgetc. Simulation log shows that we read back the character “s” that was inserted by \$ungetc.

\$fgets reads one line from a file:

```
integer i1;
i1 = $fgets( str, fd);
```

It reads characters from the file into the variable “str” until “str” is filled or a new line character is read (or an EOF is encountered). If “str” is not an integral number of bytes in length, the most significant partial byte is not used. If an error occurs reading from the file, then “i1” will be set to 0; else the number of characters read is returned in “i1.”

Let us look at an example, where we write to a file using \$fdisplay and then read back from that file using \$fgets:

```
module fileIO;
int fd, i1;
string str;

initial begin
fd = $fopen ("file1", "w"); //OPEN FOR WRITE

for (int i = 0; i < 3; i++) begin
$fdisplay (fd, "Line = %0d", i);
end
$fclose(fd); //Close file

fd = $fopen ("file1", "r"); //OPEN FOR READ

while (!$feof (fd)) begin //while not EOF
i1 = $fgets(str, fd);
$display ("Line read is : %s", str);
end
$fclose(fd); //Close file
end
endmodule
```

*Simulation log:*

Line read is : Line = 0

Line read is : Line = 1

Line read is : Line = 2

Line read is :

### V C S   S i m u l a t i o n   R e p o r t

In this example, we use \$fdisplay to write to the file “file1” which was opened for writing. We use a loop to write three lines to the file “file1.” Each line is a string that is formatted using \$fdisplay (fd, “Line = %0d”,i). So, this will write three lines as “Line = 0,” “Line = 1,” and “Line = 2” to the file “file1.”

Then we open “file1” for read. We read the file “file1” (pointed to by file descriptor “fd”) until the end of the file using \$feof. We read each line using \$fgets(str, fd), which will return each line into the string “str.” So, “str” will get the values “Line = 0”...“Line = 2.” We display these lines read from file “file1” and get the lines as displayed in the simulation log. The last line is empty, which is when EOF is realized and hence it is displayed empty.

\$fgets reads a whole line of text into “str,” including new line \n. It stops when “str” is full, at the end of line or at the end of file. It returns the number of bytes read from the line. Returns a 0 for error or end of file (does not return a -1).

## 25.5.2 \$fscanf and \$sscanf

When you want to read data from a file in a certain format, you use \$fscanf (or \$sscanf, if you want to read from a string). \$fscanf system function can be used to format data as it is read from a file. Syntax is:

```
integer is1;
is1 = $fscanf ( fd, format, args);
is1 = $sscanf ( str, format, args);
```

\$fscanf reads input lines from a file, interpret them according to a format, and place the results into “args.” The format can be an expression containing a string, string data type, or an integral data type. The string contains conversion specification, which directs the conversion of input into the arguments. A conversion specification directs the conversion of the next input field, and the result is placed in the variable specified in the corresponding argument.

For example, let us say you want to read the following file into your testbench. It has three fields: first one is a character string (“read” or “write”), the second and third fields are hex fields. Let us say the file name is “busFile.txt.” It has two lines:

```

read ffff0000 11110000
write 0000ffff 00001111

```

You want to read this file into your testbench. You will do that as follows:

```

module tb;
int fd;
string mode;
int addr, data;
int numMatches;

initial begin
    fd = $fopen("busFile.txt","r");

    //Read first line
    numMatches = $fscanf(fd,"%s %h %h", mode, addr, data);
    $display("numMatches = %0d mode = %s addr = %h data = %h",
             numMatches, mode, addr, data);

    //Read second line
    numMatches = $fscanf(fd,"%s %h %h", mode, addr, data);
    $display("numMatches = %0d mode = %s addr = %h data = %h",
             numMatches, mode, addr, data);
end
endmodule

```

#### *Simulation log:*

```

numMatches = 3 mode = read addr = ffff0000 data = 11110000
numMatches = 3 mode = write addr = 0000ffff data = 00001111

```

#### V C S   S i m u l a t i o n   R e p o r t

We open the “busFile.txt” for read. Then we scan in the input file using \$fscanf. Since there are three fields in the input file, we need three arguments, one for each input field. %s will read in the string input and %h %h will read in the hex fields. The input values will be stored in the three arguments, namely, mode, addr, and data. We can see in the simulation log that correct values for mode, addr, and data were read in. Then in your testbench, you can use mode, addr, and data as three variables in any way you want. You have read in the input file values and they are available to you in your testbench. \$fscanf returns the number of matches (=3 in our example).

The integer format specifiers are %h (or %H), %d (or %D), %o (or %O), %b (or %B), %c (or %C), %u (or %U), %z (or %Z), %m, %v, %t, and %s. The following definition is taken from (SystemVerilog – LRM). Please refer to the LRM for detailed definition:

- %h (or %H) – Matches a hexadecimal number, consisting of a sequence of characters from the set 0,1,2,3,4,5,6,7,8,9,a,A,b,B,c,C,d,D,e,E,f,F,x,X,z,Z,?, and \_.
- %d (or %D) – Matches an optionally signed decimal number, consisting of the optional sign from the set + or -, followed by a sequence of characters from the set 0,1,2,3,4,5,6,7,8,9, and \_, or a single value from the set x,X,z,Z,?.
- %b (or %B) – Matches a binary number, consisting of a sequence from the set 0,1,X,x,Z,z,?, and \_.
- %o (or %O) – Matches an octal number, consisting of a sequence of characters from the set 0,1,2,3,4,5,6,7,X,x,Z,z,?, and \_.
- %c (or %C) – Matches a single character, whose 8-bit ASCII value is returned.
- %u (or % U) – Matches unformatted (binary) data.
- %z (or %Z) – The formatting specification %z (or %Z) is defined for reading data without formatting (binary values). This formatting specifier is intended to be used to support transferring data to and from external programs that recognize and support the concept of x and z. Applications that do not require the preservation of x and z are encouraged to use the %u I/O format specification.
- %m – Returns the current hierarchical path as a string. Does not read data from the input file or str argument.
- %t – Matches a floating-point number. Used for time specification. The time value matched is scaled and rounded according to the current timescale as set by \$timeformat.

Here is an example, similar to the one we saw earlier in the chapter (where we used \$fgets to read in a line), but this one uses \$fscanf to read lines from an input file:

```

module fileIO;
    int      fd, i1;
    string str;

initial begin
    fd = $fopen ("file1", "w");

    for (int i = 0; i < 3; i++) begin
        $fdisplay (fd, "Line = %0d", i);
    end
    $fclose(fd);

    fd = $fopen ("file1", "r");

    while ($fscanf (fd, "%s = %0d", str, i1) == 2) begin
        $display ("Line: %s = %0d", str, i1);
    end

    $fclose(fd);

```

```

    end
endmodule

```

*Simulation log:*

Line: Line = 0

Line: Line = 1

Line: Line = 2

### V C S S i m u l a t i o n R e p o r t

We write to the file “file1” (file descriptor “fd”) three lines:

```

for (int i = 0; i < 3; i++) begin
    $fdisplay (fd, "Line = %0d", i);
end

```

Then, we read from “fd” these three lines using \$fscanf. The line written to the file has the format “Line = %0d.” So, for \$fscanf, we need two input fields as specified in the \$fscanf task as follows:

```
$fscanf (fd, "%s = %0d", str, i1)
```

%s corresponds to the word “Line” and %0d corresponds to “%0d” of the input lines. We read from the file as long as the number returned by \$fscanf is = 2 (2 because there are two input fields that we are reading into two arguments – recall that \$fscanf returns the number of matches). Once we reach the EOF, the number returned by \$fscanf will not be two and we will stop reading. So, \$fscanf (fd, “%s = %0d,” str, i1) reads the input line as “Line = 0,” “Line = 1,” and “Line = 2.” That is what we see in the simulation log.

Let us look at an example where we write to a string (using \$swrite) and read back from the string (using \$sscanf). \$sscanf works exactly the same way as \$fscanf, except that it reads from a string and not a file. So, use this example to also understand \$fscanf:

```

module top;

    string str, str1, str2, str3;
    reg [15:0] hex1;
    reg [7:0] hex2;
    reg [7:0] hex4,hex5;
    reg [63:0] hex3;
    reg [15:0] bin;
    reg [7:0] char, char1;
    realtime tim;
    integer count, fd;

```

```

initial begin

    $swrite(str,"      FFFF 1234 Ashok 10NS Hello 0101");
    $display("str = %s",str);

    count = $sscanf(str,"%h %h %s %t %c %c %b",
                    hex1, hex2, str1, tim, char, char1, bin);
    //Note the <space> before %h.
    //It will skip all the leading spaces in 'str'.

    $display("count=%0d hex1=%h hex2=%h str1=%s tim=%0t
char=%c char1=%c bin=%h", count, hex1, hex2, str1, tim, char,
char1, bin);

    $swrite(str," hello      goodbye      ");
    count = $sscanf(str,"%s %s %s",str1, str2, str3);
    //Note the <space> before %s.
    //It will skip all the leading spaces in 'str'.
    $display("count=%0d str1=%s str2=%s str3=%s ",
            count,str1,str2,str3);

    $swrite(str,"      hello      goodbye      ");
    count = $sscanf(str,"%c %s %s",char, str1, str2);
    //Note the <space> before %c.
    //It will skip all the leading spaces in 'str'.
    $display("count=%0d char=%c str1=%s str2=%s",
            count,char,str1,str2);

    $swrite(str,"ff001110");
    count = $sscanf(str,"%2h%2h%2h%2h",hex1,hex2,hex4,hex5);
    //%2h hex width is a must
    $display("count = %0d hex1=%h hex2=%h hex4=%h hex5=%h",
            count,hex1,hex2,hex4,hex5);

    count = $sscanf(str,"%h",hex3);
    $display("count = %0d hex3=%h",count,hex3);

end
endmodule

```

*Simulation log:*

str = FFFF 1234 Ashok 10NS Hello 0101

Warning: Too many digits in hexadecimal number '1234', truncating to target size 8 bits at time 0 in file testbench.sv line 18

```
count=6 hex1=ffff hex2=34 str1=Ashok tim=10 char=N char1=S bin=xxxx
count=2 str1=hello str2=goodbye str3=
count=3 char=h str1=ello str2=goodbye
count = 4 hex1=00ff hex2=00 hex4=11 hex5=10
count = 1 hex3=00000000ff001110
```

Study the log carefully and you'll see how strings, numbers, characters, etc. are scanned in. Even though the example uses \$sscanf, the same model is applicable directly to \$fscanf.

Some points to note:

1. For the string

“str = FFFF 1234 Ashok 10NS Hello 0101”,

the line

“count = \$sscanf(str, “%h %h %s %t %c %c %b”, hex1, hex2, str1, tim, **char**, **char1**, **bin**);”

gives the result

“count=6 hex1=ffff hex2=34 str1=Ashok tim=10 **char=N** **char1=S** **bin=xxxx**”.

First %h is preceded by a <space>. This skips all the leading spaces before the number “FFFF” in “str.” Further, note that the %c %c reads in the values “N” and “S.” After that the %b tries to read in the string “Hello.” But %b format specifier cannot read in a string. Hence, the value returned by %b is “xxxx” which is assigned to the argument “bin.” And the “count” is 6 instead of 7, since the last %b did not result in a match.

“hex2” is defined as 8-bit-wide reg. But we are reading “1234” field into it. So, the simulator will truncate the field “1234” to “34” and assign it to hex2. Hence, you get the warning as shown in the simulation log.

2. For the string

```
$swrite(str," hello goodbye ");
```

the line

count = \$sscanf(str, "%s %s %s",str1, str2, str3);

gives the result

count=2 str1=hello str2=goodbye str3=

The key thing to note here is that there are spaces before the string “hello” in “str.” The \$sscanf uses “<space> %s” as the first format specifier. The <space> before first %s will skip all the blank spaces in “str” (i.e., blank spaces before “hello”) and read in the string “hello.” Then the rest are read as shown in the

simulation log. Also, there are only two strings in “str” (hello and goodbye), so the last %s will result in a null string and count will be 2 and not 3.

### 3. For the string

```
$swrite(str,"ff001110");
```

the line

```
count = $sscanf(str,"%2h %2h %2h%2h",hex1,hex2,hex4,hex5);
```

gives the result

```
count = 4 hex1=00ff hex2=00 hex4=11 hex5=10
```

You must specify the exact format “%2h%2h%2h%2h” to read in 8-bit hex values in four different arguments, hex1, hex2, hex4, and hex5. Note that “hex1” is 15-bit wide and hence the value is 0 extended resulting in “00ff” as the value returned for “hex1” argument.

Again, compare the simulation log carefully with the model, and you will see how scanning of different characters and numbers work.

## 25.5.3 ***\$ftell, \$fseek, and \$rewind***

These tasks allow you to randomly access a file. For example:

integer offset, where:

```
offset = $ftell (fd); // tells you where you are in the file (bytes from the start)
where = $fseek(fd, offset, 0); //“0” stands for offset in bytes from the start of file
where = $fseek(fd, 10, 1); //“1” stands for offset from current position
where = $fseek(fd, -20, 2); //“2” stands for offset from the end of the file
where = $rewind (fd); // rewind the file. Go to the start of the file.
```

## 25.5.4 ***\$fread***

\$fread function reads in binary data files. In its simplest form, it requires two arguments: name of a register or name of a memory and name of the file:

```
$fread (register/mem, file_descriptor);
```

There are a few flavors of this (fd = file\_descriptor).

Integer status:

```

Status = $fread ( integral_var, fd); //for all packed data
Status = $fread (mem, fd);
Status = $fread (mem, fd, start);
Status = $fread (mem, fd, start, count);
Status = $fread (mem, fd, , count);

```

where:

Start is an optional argument. Start is used as the address of the first element in the memory to be loaded. If not present, the lowest numbered location in the memory is used. For example, if you have declared your memory as “mem[ 10: 20]” and your “start” number is 12, the first data will be loaded at mem[12]. The memory loading will begin at the start address and increment in address.

Count is also an optional argument. It is the maximum number of locations in “mem” that will be loaded. If not present, the “mem” will be filled with what data are available.

The data in the binary file are read byte by byte. If your memory is 8-bit wide, it will be loaded using one byte per memory word. If it is 9-bit wide, it will use two bytes per memory word. The data are read from the file in a big-endian manner; the first byte read is used to fill the most significant location in the memory element. If the memory width is not evenly divisible by 8 (8, 16, 24, 32), not all data in the file are loaded into memory because of truncation.

You cannot read “x” or “z” using \$fread. The data loaded from the file are taken as two value data. A bit set in the data is interpreted as a 1, and bit not set is interpreted as a 0.

### **25.5.5 \$readmemb and \$readmemh**

\$readmemb reads in a file with binary data and loads it into a memory array, while \$readmemh does the same for a hex file. These tasks basically allow you to initialize your memory. These are some of the most useful system tasks in the language. Syntax is:

```
$readmemb ("filename", mem_array [, start_addr [, end_addr]]);
```

```
$readmemh ("filename", mem_array [, start_addr [, end_addr]]);
```

Start\_addr and end\_addr apply to the addresses of the memory array. This address range represents the highest dimensions of the data in the “filename.”

The file to be read in can contain only:

- White space (space, new line, tabs, form feeds).
- Comments (// or /\* \*/).

- Binary or hex numbers. The numbers cannot have either the length or the base format specified (e.g., 16'h FFFF is not allowed; only FFFF). Both upper-case and lower-case numbers are allowed.

As the file is read, each number encountered is assigned to a successive work element of the memory array. Note that in the file, you do not have to have data for contiguous memory; you can have data for sparse memory as well. For example, you can have input file as:

```
ffff
0000
1111
0101
```

And the mem\_array is defined as “logic [15:0] mem[0:3]”; then the mem will be loaded starting address 0 to end address 3.

But you can also have the input data file as (address specification along with data):

```
@0 ffff
@2 1111
```

This will load data only at location 0 and 2 of the memory array.

If no addressing information is specified within the system task and no address specifications appear within the data file, then the default start address will be the lowest address in the memory. Consecutive words will be loaded until either the highest address in the memory is reached or the data file is completely read. If the start address is specified in the task without the finish address, then loading will start at the specified start address and will continue upward toward the highest address in the memory.

Let us look at an example.

The example reads in three different data files.

“readmemFileH” (hex data file) which has the following content:

```
FFFF
FOFO
0101
1111
```

“readmemFileB” (binary data file) which has the following content:

```
xxxx_0000_1111_0000
0000_1111_0000_1111
1010_1010_1010_1010
0101_0101_0101_zzzz
```

“readmemFileBAddr” (binary data file with address specification) which has the following content:

```
@0 0000_1111_0000_1111
@3 1110_0001_1010_0101
```

```
@5 1010_1111_1010_1111
@9 0101_0101_1111_0101
@B 1111_0000_1111_0000
@F 1100_1100_1100_1100
```

Here is the testbench:

```
module fileIO;
    int      fd, status;
    logic [15:0] memH [0:3];
    logic [15:0] memB [0:3];
    logic [15:0] memBAddr [0:15];
    logic [15:0] memm [ ];

    initial begin
        memm = new [4];

        $readmemh ("readmemFileH", memH);
        for (int i = 0; i <= 3; i++) begin
            $display ("memH[%0d] = %h", i, memH[i]);
        end

        $readmemb ("readmemFileB", memB);
        for (int i = 0; i <= 3; i++) begin
            $display ("memB[%0d] = %b", i, memB[i]);
        end

        $readmemb ("readmemFileBAddr", memBAddr);
        for (int i = 0; i <= 15; i++) begin
            $display ("memBAddr[%0d] = %b", i, memBAddr[i]);
        end

        $readmemb ("readmemFileB", memm);
        for (int i = 0; i <= 3; i++) begin
            $display ("memm[%0d] = %b", i, memm[i]);
        end
    end
endmodule
```

*Simulation log:*

```
memH[0] = ffff
memH[1] = f0f0
memH[2] = 0101
memH[3] = 1111
```

```

memB[0] = xxxx000011110000
memB[1] = 0000111100001111
memB[2] = 1010101010101010
memB[3] = 010101010101zzzz

memBAddr[0] = 0000111100001111
memBAddr[1] = xxxxxxxxxxxxxxxxxx
memBAddr[2] = xxxxxxxxxxxxxxxxxx
memBAddr[3] = 1110000110100101
memBAddr[4] = xxxxxxxxxxxxxxxxxx
memBAddr[5] = 1010111110101111
memBAddr[6] = xxxxxxxxxxxxxxxxxx
memBAddr[7] = xxxxxxxxxxxxxxxxxx
memBAddr[8] = xxxxxxxxxxxxxxxxxx
memBAddr[9] = 0101010111110101
memBAddr[10] = xxxxxxxxxxxxxxxxxx
memBAddr[11] = 1111000011110000
memBAddr[12] = xxxxxxxxxxxxxxxxxx
memBAddr[13] = xxxxxxxxxxxxxxxxxx
memBAddr[14] = xxxxxxxxxxxxxxxxxx
memBAddr[15] = 1100110011001100

memm[0] = xxxx000011110000
memm[1] = 0000111100001111
memm[2] = 1010101010101010
memm[3] = 010101010101zzzz

```

### V C S   S i m u l a t i o n   R e p o r t

We read in three different files with three different \$readmem tasks. \$readmemh is used to read in the “readmemFileH” data file which has hex data in it. First \$readmemb is used to read in the “readmemFileB” data file which has binary data in it. Second \$readmemb is used to read in the “readmemFileBAddr” data file which also has binary data but with address specification. The simulation log displays what was written into the memory arrays with each of the \$readmem task.

Important to note here is the data read from the file “readmemFileBAddr.” This file has fewer data than the maximum number of locations in the memory array in which the data is to be loaded. The memory\_array for this file is “logic [15:0] memBAddr [0:15];” with 16 memory locations (16-bit wide each). But the input data file has data only at certain addresses. So, only part of the memory (memBAddr) is loaded from the input file. Hence, in the simulation log, you see only a few of the locations with known data; the rest are not initialized (unknown data).

Also note that we can load a dynamic array as shown with the array “logic [15:0] memm[ ];.” “memm” is initialized to four elements and then loaded with the file “readmemFileB.” The last part of simulation log shows its contents.

### 25.5.6 \$writememb and \$writememh

\$readmem tasks read from a file into an array. \$writemem tasks do just the opposite. They write to a file from an array. There is the write hex data to an array \$writememb, and the binary data write to an array \$writememh:

```
$writememb (filename, memory_array_name [, start_addr [, end_addr]]);
$writememh (filename, memory_array_name [, start_addr [, end_addr]]);
```

If the “filename” exists at the time these tasks are called, the file will be overwritten (i.e., there is no append mode). The \$writemem tasks treat packed data identically to \$readmem tasks, as described before.

Here is an example, where we read data (\$readmemh) from a file and dump that data into an array. Then we read that array and write that data into a file (\$writememh). Then we read back the data that were written to the file with \$writememh into an array.

We first read in the file readmemFileH, which contains:

```
FFFF
F0F0
0101
1111
```

Here is the testbench:

```
module fileIO;
    int fd, status;
    logic [15:0] memH [0:3];
    logic [15:0] memH1 [0:3];

    initial begin

        //read from file to memory array
        $readmemh ("readmemFileH", memH);
        for (int i = 0; i <= 3; i++) begin
            $display ("File readmemFileH : memH[%0d] = %h", i,
memH[i]);
        end

        //write from memory array to output file
        $writememh("writememFileH", memH);
```

```

//read back from the file written above
$readmemh ("writememFileH", memH1);
for (int i = 0; i <= 3; i++) begin
    $display ("File writememFileH : memH1[%0d] = %h", i, memH1[i]);
end
end
endmodule

```

*Simulation log:*

```

File readmemFileH : memH[0] = ffff
File readmemFileH : memH[1] = f0f0
File readmemFileH : memH[2] = 0101
File readmemFileH : memH[3] = 1111
File writememFileH : memH1[0] = ffff
File writememFileH : memH1[1] = f0f0
File writememFileH : memH1[2] = 0101
File writememFileH : memH1[3] = 1111

```

#### V C S   S i m u l a t i o n   R e p o r t

First, we read from the file “readmemFileH” (using \$readmemh) and dump the data into the array “memH.” Then we read the array “memH” and write that data out to the file “writememFileH” using \$writememh. Then we read back from the file “writememFileH” and dump the data into the array “memH1.” From the simulation log, you can see that the data read from “readmemFileH” and data read from “writememFileH” are identical.

## 25.6 \$test\$plusargs and \$value\$plusargs

There are times when you want to pass command line arguments to the simulator to take some action based on those command line arguments. Such arguments are distinguished from the other simulator command line arguments with the plus (+) character. Such arguments are called “plusargs,” and they are read in through the following system functions:

```

$test$plusargs (string)
$value$plusargs (user_string, variable)

```

### 25.6.1 \$test\$plusargs

```
$test$plusargs (string)
```

The \$test\$plusargs function searches the list of “plusargs” (command line arguments) for the user-specified string. Note that the string specified with \$test\$plusargs function cannot include the “+” sign of the command line arguments. The plusargs present on the command line are searched in the order provided. If the string provided by the user matches the command line argument, the function returns a non-zero value; else it returns a 0 (false).

Here is an example. The command line arguments provided to the simulator are:

```
+football +soccer

module testarg;

initial begin
    if ($test$plusargs("football"))
        $display("string football found");

    if ($test$plusargs("foot"))
        $display("Substring foot found");

    if ($test$plusargs("f"))
        $display("Substring starting with f found");

    if ($test$plusargs("soccer"))
        $display("string soccer found");

    if ($test$plusargs("baseball"))
        $display("string baseball found");
    else
        $display("string baseball NOT found");
end
endmodule
```

*Simulation log:*

```
string football found
Substring foot found
Substring starting with f found
string soccer found
string baseball NOT found
```

## V C S S i m u l a t i o n R e p o r t

Two command line arguments +football and +soccer are provided to the simulator. We check for these arguments using \$test\$plusargs in the testbench. We can not only check for the entire string (e.g., “football”), but we can also search for substrings of that string (e.g., “foot” or “f”). The simulation log shows the results. Note that we did not specify +baseball on the command line argument, so when we tested for it, we could not find it, as displayed in the simulation log (“string baseball *not* found”).

You can specify as many command line arguments as you like, each starting with a “+” sign.

### 25.6.2 \$value\$plusargs

```
$value$plusargs (user_string, variable)
```

The user\_string is of the form “plusargs\_string format\_string.” This system function first searches the list of “plusargs” (command line arguments) for the user-specified “plusargs\_string.” The string is specified in the first argument to the system function as a string variable.

This user\_string will not include the leading +sign of the command line argument. The “plusargs” are searched in the order provided.

If the “plusargs” matches all characters of the provided “plusargs\_string,” the function returns a non-zero integer, and the remainder of the string is converted to the type specified in the user\_string (“format\_string”). The resulting value is stored in the “variable” provided with the system function. The “format\_string” is the same as that for \$display system task that we have seen throughout the book.

Here is an example. For the example, the command line arguments are:

```
+GAME=cricket +TEST=5 +REAL3.142
```

```
module testval;
    string game;
    int testNum;
    real realNum;

    initial begin
        if ($value$plusargs ("GAME=%s", game))
            $display("GAME=%s", game);

        if ($value$plusargs ("TEST=%d", testNum))
            $display("TEST=%0d", testNum);
```

```
if ($value$plusargs ("REAL%f", realNum) )
    $display("REAL=%f", realNum);
end
endmodule
```

*Simulation log:*

GAME=cricket

TEST=5

REAL=3.142000

V C S   S i m u l a t i o n   R e p o r t

Note that the user\_string (“plusargs\_string format\_string”) provided in the testbench with the \$value\$plusargs function needs to *exactly match the command line argument*. In this example, note that the command line argument, “REAL3.142,” does not have an “=” sign, and so the user\_string provided in the testbench (“REAL%f”) with the \$value\$plusargs function should also not have it in the provided user\_string. In other words, the user\_string must exactly match the format of the command line argument.

## 25.7 Value Change Dump (VCD) File

In order to facilitate debugging or to analyze simulation results, systemverilog provides the so-called value change dump (VCD) file. It contains value changes on the IO (or internal) signals of your design. It is an ascii file which needs to be post-processed to make the value changes of signals more meaningfully readable. Some EDA tools read this file and convert it into graphical display files for debug. Some tools analyze this file for toggle analysis. Many such applications can be derived from a VCD file.

The steps to creating a VCD file are simple:

1. You insert VCD system tasks such as \$dumpfile (“filename”), \$dumpvars(...), etc. in your testbench.
2. Run simulation.
3. Simulator generates the VCD file.
4. Post-process the VCD file.

### 25.7.1 *\$dumpfile*

\$dumpfile system task allows you to specify the name of the VCD file:

```
$dumpfile("filename");
```

The filename is optional and defaults to the string literal “dump.vcd,” if not specified.

For example:

```
initial begin
$dumpfile("PCIe.dump");
end
```

### **25.7.2 \$dumpvars**

\$dumpvars task is used to list the variables to dump into the file specified by \$dumpfile. You can invoke \$dumpvars as often as desired (e.g., from various blocks). Syntax is:

```
$dumpvars;
$dumpvars(levels [, list_of_modules_or_variables]);
```

If no arguments are given, \$dumpvars will dump all the variables in a given model to the VCD file. If arguments are given, the first argument indicates how many levels of the hierarchy below each specified module instance to dump to the VCD file. A level of “0” means dump all the variables of all the instances below a given module. The second and further arguments specify the scope of the model. It can specify the entire modules or individual variables within a module. For example:

```
$dumpvars (1, top);
```

The level of “1” means dump all variables within module “top” but do not dump variables in any of the modules instantiated by “top”:

```
$dumpvars (0, top);
```

The level of “0” means dump all variables within the module “top” and in all module instances below the module “top”:

```
$dumpvars (0, PCI.Master, PCI.Target.cbe);
```

This task will dump all variables at instance-level PCI.Master (and all instances below it) and also dump the specific variable “PCI.Target.cbe.”

### 25.7.3 \$dumpon/\$dumpoff

\$dumpvars task starts dumping values when it is invoked. But you can suspend this dump using \$dumpoff task and then reinvoke it using \$dumpon task.

No value changes are dumped between \$dumpoff and \$dumpon.

When \$dumpoff is executed, a checkpoint is made in which all selected variables are dumped as “x” value. And then when \$dumpon is executed, the current value of each variable is dumped at that time. When dumping is enabled, the value change dumper records the values of the variables that change during each time increment. For example:

```
always @(negedge reset) $dumpoff;  
always @(posedge reset) $dumpon;
```

### 25.7.4 \$dumplimit

\$dumplimit sets the size of the VCD file:

```
$dumplimit (filesize);
```

The filesize specifies the maximum number of bytes to be dumped in the dumpfile. When the filesize is reached, the dumping stops with a comment in the VCD file indicating that dumping has stopped.

### 25.7.5 \$dumpflush

\$dumpflush task is used to empty the current VCD file buffer of the OS to verify that all the data in the data buffer are flushed/stored in the VCD file. After \$dumpflush, dumping resumes as before. A common application is to call \$dumpflush to update a dump file so that another application program can read the VCD file *during* simulation.

### 25.7.6 \$dumpall

\$dumpall creates a checkpoint, meaning when it is called, *all* the variables are dumped into the dump file at that particular time, regardless of whether the port values have changed since the last time step. For example:

```
always #100_000 $dumpall;
```

Create a checkpoint at every 100\_000 time units and dump all the variables at that time.

### 25.7.7 *\$dumpports*

\$dumpports dumps all the ports, i.e., primary I/O pins, in the model. However, any ports that exist in the instantiations below the given scope list are not dumped. Here is the syntax:

```
$dumpports (scope_list, filename);
```

The scope\_list is one or more module names. Only module names are allowed, not the variables. You can have more than one module name, separated by a comma. Hierarchical path names to the module are allowed. Filename is the name of the file where port VCD information will be dumped. If no filename is provided, a filename dumpports.vcd will be generated.

Note that each scope specified in the scope\_list will be unique. If multiple calls to \$dumpports are made, the scope\_list values in these calls will also be unique.

Both \$dumpports and \$dumpvars can be used in the same code.

There are also the \$dumpposrtoff and \$dumpportson tasks that turn off and on the dumping of ports into the VCD file. These tasks are analogous to \$dumpoff and \$dumpon tasks.

Also, similar to \$dumpall, there is the \$dumpportsall system task that creates a checkpoint in the VCD file that shows the values of *all* the selected ports at that time, regardless of whether the port values have changed since the last time step.

Also, similar to \$dumplimit, there is the analogous task \$dumpportslimit.

Also, similar to \$dumpflush, there is the analogous task \$dumpportflush.

# Chapter 26

## GENERATE Constructs



**Introduction** Generate blocks allow creating structural-level code. The chapter discusses nuances of generate constructs, including loop constructs and conditional constructs.

Generate blocks allow creating structural-level code. You can multiply instantiate a module or perform conditional instantiation of modules. You can create multiple instantiations of modules and code or conditionally instantiate blocks of logic code. You can generate multiple occurrences of variables, nets, tasks, functions, continuous assignments, initial procedures, and always procedures. Generate lets you build structural logic (structural hardware) with instantiations from a loop (or from a conditional; if-else or “case”). The idea behind “generate” is that it saves you writing the same code segment multiple times, preventing you from making errors. It helps you develop a cleaner code.

Generate loops are evaluated at elaboration time (think about it; you cannot generate hardware on the fly) and not at run time. So, the loop limits must be fully known at elaboration time. In contrast, procedural “for” loops are evaluated at run time when the procedural block is activated. The result of the generate block is determined before simulation begins. Therefore, all expressions in generate schemes are constant expressions.

There are two types of generate constructs: loops and conditionals. Loop generate construct allows a single generate block to be instantiated into a model multiple times. Conditional generate construct (if-else generate or “case” generate constructs) instantiates at most one generate block from a set of alternative generate blocks.

The keywords *generate* and *endgenerate* are used in a module to define a generate region. But these keywords are optional as we will see in forthcoming examples.

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_26](https://doi.org/10.1007/978-3-030-71319-5_26)) contains supplementary material, which is available to authorized users.

## 26.1 Generate: Loop Constructs

A loop generate construct allows a generate block to be instantiated multiple times. The syntax is similar to a normal “for loop” statement. But the loop index variable is declared in a *genvar* declaration prior to its use in a generate loop. The “*genvar*” is an integer used during elaboration time to evaluate the generate loop and create instances of generate block. The *genvar* keyword is a new data type, which stores positive integer values. It differs from other Verilog variables in that it can be assigned values and can be changed during compile or elaboration time. The index variable used in a generate loop must be declared as a *genvar*. Since generate structures are evaluated at elaboration time, the “*genvar*” does not exist at simulation time. It is an error if any bit of the “*genvar*” is set to “x” or “z” during the evaluation of the loop.

The syntax for a generate loop is as follows:

```
generate
    for (initial value, final value, increment) begin :
generate_label
    <logic>
end
endgenerate
```

Here is an example:

```
module haddr ( input   a, b,
               output  sum, cout);

    assign sum = a ^ b;
    assign cout = a & b;

    initial $display("module instance %m");
endmodule

module top_level
#(parameter N=4)
(  input [N-1:0] a, b,
  output [N-1:0] sum, cout);

    genvar i;

    //Generate N instantiation of 'haddr' module
    generate
        for (i = 0; i < N; i = i + 1) begin : haddr_block
```

```

        haddr u0 (a[i], b[i], sum[i], cout[i]);
    end
endgenerate
endmodule

```

*Simulation log:*

```

# run -all
# module instance top_level.haddr_block[0].u0
# module instance top_level.haddr_block[1].u0
# module instance top_level.haddr_block[2].u0
# module instance top_level.haddr_block[3].u0
# exit

```

We have declared a half-adder module called “haddr.” In the module “top\_level,” we declare a “parameter” N (parameter N=4) and size the I/O of the module according to this parameter. We create a “generate” loop construct that loops through “N” times (N being the parameter) and instantiates the module “haddr” that many times. A few things to note.

We did not have to individually instantiate the “haddr” N number of times. The “generate” loop does that for us, resulting in a very concise and error-free code.

The label assigned to the “for” loop (“haddr\_block”) is optional. If not assigned, the simulator will assign one for you. This label is essential since it determines the instance names given to the module (haddr) instances within the “for” loop.

Distinct instance names are given to the instantiated module “haddr” as shown in the simulation log (we display the hierarchical instance name from the module “haddr”). Note how the instance names are derived. They are based on the label given to the “for” loop of the generate block, namely, “haddr\_block.” That is the only way the simulator can distinguish between the various instances generated by the generate block. The instance name of “haddr” is “u0,” so that has to remain the same for all instances. The names are hierarchical starting from the module “top\_level.”

Study the simulation log carefully to see how instance names are derived.

Here is another example, where we generate multiple procedural assignment statements. The idea is that you can not only have module/gate instance blocks in a generate loop; you can also have non-module instance elements such as assign statements:

```

module mux(
    input logic [0:15] [127:0] mux_in,
    output logic [127:0] mux_out
);

genvar i;

```

```

generate
    for (i=0; i < 4; i++) begin
        assign mux_in[i] = $urandom;
        $info( );
    end
endgenerate

assign mux_out = mux_in[0] | mux_in[1] | mux_in[2] |
mux_in[3];

initial
$monitor("mux_out = %0d", mux_out);
endmodule: mux

```

Simulation log (mentor QuestaSim):

```

# ** Info:
# Time: 0 ns Scope: mux.genblk1[0] File: testbench.sv Line: 39
# ** Info:
# Time: 0 ns Scope: mux.genblk1[1] File: testbench.sv Line: 39
# ** Info:
# Time: 0 ns Scope: mux.genblk1[2] File: testbench.sv Line: 39
# ** Info:
# Time: 0 ns Scope: mux.genblk1[3] File: testbench.sv Line: 39
#
# vsim -voptargs+=acc=npr
# run -all
# mux_out = 93133385
# exit

```

In this example, we use “assign” statements in the generate block to generate four indexed “temp” and “mux\_in” vectors. Note that we did not give a block name to the “for” loop of the “generate” block. Hence, the instance names given to each iteration of the loop use a simulator-assigned name called “genblk1.” The \$info shows the names of four loop iterations of the “generate” block.

One more example showing how the generate block instance names are derived:

```

module for_gen;

generate
genvar i;
    for (i = 0; i < 4; i = i + 1)
begin : gen1
    genvar j;

```

```

        for (j = i; j < 4; j = j + 1)
            begin : gen2
                initial $display("%m");
            end
        end
endgenerate

endmodule

```

*Simulation log:*

```

# run -all
# for_gen.gen1[0].gen2[0]
# for_gen.gen1[0].gen2[1]
# for_gen.gen1[0].gen2[2]
# for_gen.gen1[0].gen2[3]
# for_gen.gen1[1].gen2[1]
# for_gen.gen1[1].gen2[2]
# for_gen.gen1[1].gen2[3]
# for_gen.gen1[2].gen2[2]
# for_gen.gen1[2].gen2[3]
# for_gen.gen1[3].gen2[3]
# exit

```

There are two “for” loops under the generate block. Each “for” loop is given a name, “gen1” and “gen2.” So, the instance names are derived from these names. Study carefully the simulation log, and you will see how the hierarchical instance names are derived.

## 26.2 Generate : Conditional Construct

Generate constructs can also be conditional. They can be if-generate and case-generate. The conditional generate construct selects at most one generate block from a set of alternative generate blocks based on *constant* if-else or case expression evaluated during elaboration. It is allowed to have more than one generate block with the same name within a single conditional operator, since at most one of the alternative generate blocks is instantiated (see example below).

Here is an example, showing a case-generate:

```

module func_gen_mod
#(parameter select = 0)
(
    input logic clk, rst,
    input logic [7:0] data_in,

```

```

        output logic [15:0] data_out
    );

always_ff @(posedge clk) begin
    if (rst) begin
        data_out <= 'd0;
    end
    else begin
        // To invoke a function within a generate block,
        // hierarchically call it
        // <generate_blk_name>.<function_name>
        data_out = func_gen.dout(data_in);
        $display($stime,,, "data_in = %h data_out = %h",
data_in, data_out);
    end
end

// The generate-endgenerate keywords are optional
// It is the act of using a parameter, 'select', in
the case
// statement that makes it a generate block
//
// Also notice how all the generate blocks are given
the same
// name `func_gen` and all the function names are the same
// `dout`. This is correct because only one of the
// function declarations is compiled during elaboration
// phase. Hence, 'select' needs to be a constant.

//generate //optional
case (select)
    1'b0:
        begin: func_gen
            function automatic [15:0] dout;
                input [7:0] data_in;

                dout = data_in + 'hffff;
            endfunction
        end
    1'b1:
        begin: func_gen
            function automatic [15:0] dout;
                input [7:0] data_in;

                dout = data_in + 'h0001;
            endfunction
        end
end

```

```

        endfunction
    end
    default:
        begin: func_gen
            function automatic [15:0] dout;
                input [7:0] data_in;

                dout = data_in + 'hff00;
            endfunction
        end
    endcase
//endgenerate
endmodule: func_gen_mod

```

Testbench:

```

module test;
    logic [7:0] data_in;
    wire [15:0] data_out;

    logic clk, rst;

    func_gen_mod #(1) f1(clk, rst, data_in, data_out);

    initial begin
        clk = 0; rst = 0;
        forever #10 begin
            data_in = $urandom;
            clk = !clk;
        end
    end

    initial #100 $finish(2);
endmodule

```

*Simulation log:*

```

# run -all
#      10  data_in = b8 data_out = 00b9
#      30  data_in = 66 data_out = 0067
#      50  data_in = c6 data_out = 00c7
#      70  data_in = 72 data_out = 0073
#      90  data_in = 67 data_out = 0068
# $finish  : design.sv(17)

```

We have a case-generate construct that selects at most one, one of the many function blocks. Note that case expression, namely, “select,” needs to be a constant because the generate block is evaluated at elaboration time, and it will select only one of the many case items. In our example, we use select = 1 when we instantiate func\_gen\_mod, and hence only one of the case statements will always be executed (which has dout = data\_in + ‘h0001; in it). Hence, in the simulation log, you see that data\_out is always data\_in+1.

Note also that the “generate” and “endgenerate” keywords are optional in this example. It is the fact that the case statement uses a parameter (“select”) as case expression, making this a case-generate block. I, however, like to explicitly use generate/endgenerate keywords for better readability.

Since only one of the “function” from the case block will be selected during elaboration time, it is ok to have all the generate blocks with the same name (“func\_gen” in this example), and also all the functions have the same name as well (“dout” in this example).

The testbench assigns a 1'b1 to the “select” parameter, and as you notice from the simulation log, the case item will “select” = 1 is executed.

Here is an example of using the if-generate construct (taken from (SystemVerilog – LRM)):

```

module test;
parameter p = 1, q = 0;
wire a, b, c;

generate
  if (p == 1)
    if (q == 0)
      begin : u1 // If p==1 and q==0, then instantiate
          and g1(a, b, c); // 'and' with hierarchical name
test.ul.g1
      end

    else if (q == 2)
      begin : u1 // If p==1 and q==2, then instantiate
          or g1(a, b, c); // 'or' with hierarchical name
test.ul.g1
      end

    else if (p == 2)
      case (q)
        0, 1, 2:
        begin : u1 // If p==2 and q==0, 1, or 2, then instantiate
          // 'xor' with hierarchical name test.ul.g1
          xor g1(a, b, c);
        end
      endcase
    end
  end
endmodule

```

```

default:
begin : u1 //If p==2 and q!=0, 1, or 2, then instantiate
// 'xnor' with hierarchical name test.u1.g1
    xnor g1(a, b, c);
end
endcase
endgenerate
endmodule

```

The module test has nested “if-else” generate statements. Note that the block name in each “if” (or “else-if”) condition is the same (“u1”). Again, this is because only one of the “if-else” branch will be selected during elaboration time based on the parameters “p” and “q.” The comments show how the different hierarchical instance names of generated blocks are derived.

Here’s an example that shows how behavioral code (an always block in this case) can be selectively generated:

```

module FlippedFlop
#(parameter ClkPolarity = 1)
(input Clk, D, output reg Q);
generate if (ClkPolarity)
    always @ (posedge Clk) Q <= D;
else
    always @ (negedge Clk) Q <= D;
endgenerate
endmodule

```

Another example:

```

generate
genvar i;
for (i = 0; i < N; i = i + 1) begin : ShiftReg
    case (i)
        0 :          // first stage
            always @ (posedge clk) SR[0] <= D;
        N - 1 :      // final stage
            always @ (posedge clk) Q <= SR[N - 2];
        default : // intermediate stages
            always @ (posedge clk) SR[i] <= SR[i - 1];
    endcase
end // ShiftReg
endgenerate

```

# Chapter 27

## Compiler Directives



**Introduction** SystemVerilog offers multitude of compiler directives to steer the course of your code. The chapter discusses `define, `ifdef, `elsif, `ifndef, `timescale, `default\_nettype, etc.

These are compile time directives to allow you to do many different things. They are preceded by the (`) character (*grave accent* character) (do not confuse that with the apostrophe character (')). *The compiler directives are global* in that the scope of a compiler directive extends from the point where it is processed across all the files processed in the current compilation unit, until a point where another compiler directive supersedes it.

### 27.1 `define

`define is one of the most widely used compiler directives. It is a text substitution macro. It helps you provide meaningful names to commonly used pieces of text. For example, where a constant number is repetitively used throughout a description, a text macro is useful in that only one place in the source description would need to be altered if the value of the constant needed to be changed. For example:

```
`define widebus 16'hFFFF
```

And use `widebus throughout your source description. If in the future, the definition of widebus changes to 32'hFFFF\_FFFF, you need to change only one line; the `define in one place (`define widebus 32'hFFFF\_FFFF) and the new definition will be applied throughout your source, wherever `widebus is used.

---

**Electronic Supplementary Material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-71319-5\\_27](https://doi.org/10.1007/978-3-030-71319-5_27)) contains supplementary material, which is available to authorized users.

As mentioned before, `define creates a macro for text substitution. It can be used both inside and outside of design elements (e.g., inside or outside a “module”). Its syntax is:

```
`define text_macro_name [(list of formal arguments)]
macro_text
```

where the formal arguments are optional. But they are very useful since it allows the macro to be customized for each use individually. The scope of the formal argument extends up to the end of the macro text. A formal macro argument may have a default.

The macro\_text can span over multiple lines using preceding the newline character with backslash (\ ) at the end of each line. The first newline character not preceded by a backslash will end the macro text.

*Redefinition of text macros is allowed*; the latest definition encountered by the compiler prevails.

`define is global in scope. It spans across multiple modules and multiple files.

Let us look at an example. It shows many different ways to use `define global substitution macro:

```
`define busWidth 31
`define delay #10

`define nandD(dly) nand #dly

`define comment $display("this is a long ", \
"comment that spans multiple lines");

`define disp(x,y) initial $display(x, y);

`define tdisp(a = 5, b = 10, c) initial $display(a,,b,,c);

`define conditional(a,b) (a > b ? a : b)

`define mF(filename) `"/springer/mydir/filename"`

module def;

`include `mF(examples) //`include "/springer/mydir/
examples"
```

```

    logic clk;
    wire o1, i1, i2, o2;
    logic [0:`busWidth] bus; //expands to - logic[0:31] bus;

    initial begin
        clk = 0;
    //expands to - forever #10 clk = !clk;
        forever `delay clk = !clk;
    end

    //expands to - nand #10 n1(o1, i1, i2);
    `nandD(10) n1 (o1, i1, i2);

    //expands to - nand #20 n2(o2, o1, i2);
    `nandD(20) n2(o2, o1, i2);

    initial `comment

        `disp("GoX"," GoY")
        `disp(10 , 20)
        `disp("End", )

        `tdisp(10, 20, 30) //10 20 30
        `tdisp( , , 15) //5 10 15
        `tdisp(10, ,25) //10 10 25
        `tdisp( , , ) //5 10
        `tdisp("first", "second", "third")

        assign bus = `conditional (20,10); //=(20 > 10 ? 20 : 5)
        initial $display("bus = %0d",bus);
    endmodule

```

*Simulation log:*

this is a long comment that spans multiple lines

GoX GoY

10	20
----	----

End

10	20	30
5	10	15
10	10	25
5	10	

first second third

bus = 20

V C S   S i m u l a t i o n   R e p o r t

We are using many different forms of `define in this example. Let us look at each. First, we define:

```
`define buswidth 31
```

We use this `define as follows in the model:

```
logic [0:`buswidth] bus;
```

This will expand into:

```
logic [0:31] bus
```

Next, we define:

```
`define delay #10
```

And use it in the forever block as:

```
forever `delay clk = !clk;
```

This will expand into:

```
forever #10 clk = !clk;
```

Then we define “nandD” with formal argument “dly”:

```
`define nandD(dly) nand #dly
```

And use it in instantiating “nand” gates:

```
`nandD(10) n1(o1, i1, i2); //expands to - nand #10 n1(o1,  
i1, i2);  
`nandD(20) n2(o2, o1, i2); //expands to - nand #20 n2(o2,  
o1, i2);
```

Then we define a macro text that spans over two lines. In order to do that we use “\” (backslash) before the new line in the text:

```
`define comment $display("this is a long \  
comment that spans multiple lines");
```

We then use it in the model as:

```
initial `comment
```

This will display the following in simulation log:

this is a long comment that spans multiple lines

Then, we declare another define with formal arguments:

```
`define disp(x,y) initial $display(x, y);
```

And use it as follows in the model:

```
`disp("GoX"," GoY")
`disp(10 , 20)
`disp("End", )
```

There are two formal arguments (x and y) in `define disp(x,y), and we can give these arguments different values. We can also omit an argument which will then remain “blank” in the display. The above three lines give us the following in simulation log:

```
GoX GoY
      10      20
End
```

We declare another define with formal arguments as follows:

```
`define tdisp(a = 5, b = 10, c) initial $display(a,,b,,c);
```

Here, we give default values to the first two formal arguments (a = 5 and b = 10). We do not give a default value to the last argument “c.” We then use this `define in the model as follows. Essentially, we are showcasing how the formal arguments and actual arguments work:

```
`tdisp(10, 20, 30) //10 20 30
`tdisp( , , 15) //5 10 15
`tdisp(10, ,25) //10 10 25
`tdisp( , , ) //5 10
`tdisp("first", "second", "third")
```

We are providing (or not) different actual values to the formals of the `define macro. Note how the default values take place in the absence of an actual on that formal argument. We get the following in the simulation log:

```
10      20      30
 5      10      15
10      10      25
```

5	10
first	second
	third

Then we declare an expression as a `define with formal arguments:

```
`define conditional(a,b) (a > b ? a : b)
```

This is to showcase that you can effectively use `define for such text substitution. If in the future, your expression definition changes, you have to change it in only one place, where you declared the `define macro. We use this in the code as follows:

```
assign bus = `conditional(20,10); //=(20 > 10 ? 20 : 5)
```

Since 20 is greater than 10, the “bus” will be assigned a value of 20, as shown in the simulation log:

bus = 20

Finally, we define a macro with ““” (quote) in it and also “/” (forward slash) in it:

```
`define mF(filename) `"/springer/mydir/$filename`"
```

An ``overrides the usual lexical meaning of” indicates that the expansion will include the quotation

mark, substitution of actual arguments, and expansions of embedded macros. This allows string literals to be constructed from macro arguments.

We use this `define for the filename in the `include directive:

```
`include `mF(examples)
```

This will expand into:

```
`include "/springer/mydir/examples"
```

So, this example shows many ways to use the text substitution macro `define effectively in your design.

### 27.1.1 `*undef* and `*undefinedall*

The directive `undef undefines the specified text macro if previously defined by a `define macro:

```
`undef text_macro_name
```

For example, in the preceding example, we can undefine the text macro `delay as:

```
`undef delay
```

And `delay will cease to exist. If you have used it someplace, you will get a compile error stating that the macro is undefined.

The `undefineall will undefine all text macros previously defined by `define compiler directive. It can appear anywhere in the source code and does not take any arguments.

## 27.2 `ifdef, `else, `elsif, `endif, and `ifndef

These are conditional compilation compiler directives. They allow you to conditionally include lines in your SystemVerilog source description. There are many uses of these compiler directives. If you want to exclude some experimental code (or include for that matter) based on a `define <name>, you can do so. Useful also for selecting different representations of design – structural vs. behavioral, choosing different timing requirements, choosing among different input stimulus or response checking, etc. Syntax is:

```
`ifdef text_macro_identifier (OR `ifndef text_macro_identifier)
    <lines of code>
`elsif text_macro_identifier
    <lines of code>
`endif
```

Nesting of these compiler directives is allowed.

Note that the text\_macro\_identifier can be declared within the module. But you can also declare them as compile-type options on the command line of a simulator (explained below). Let us look at an example:

```
`define true
`define false
`define behavioral

module directive;
    wire a, b, c;

    initial begin
        `ifdef true
```

```

$display("TRUE");
`ifndef false
    $display("NESTED FALSE after TRUE");
`endif
`else
    $display("NONE");
`endif
end

`ifndef behavioral //if 'behavioral' is -not- defined
    and a1 (a, b, c);
    initial $display("GATE LEVEL");
`else
    assign a = b & c;
    initial $display("BEHAVIORAL");
`endif
endmodule

```

*Simulation log:*

```

TRUE
NESTED FALSE after TRUE
BEHAVIORAL
    V C S   S i m u l a t i o n   R e p o r t

```

We have declared three `define, namely, `define true, `define false, and `define behavioral. Note how these are declared. You have simply given a text\_macro\_name after the `define keyword. This is different from, for example, doing `define true 1'b1.

Then in the code, we use `ifdef true to select a branch and do a nested `ifdef false to select an additional branch. These two nested `ifdef compile the lines as shown via \$display in the simulation log.

We also use `ifndef (i.e., if not defined) in the code to choose between behavioral and structural codes. `ifndef behavioral means that if “behavioral” is not defined, then choose the branch associated with `ifndef. In our case, we have indeed defined “behavioral,” so the `else branch is taken as evidenced from the simulation log.

Note that as mentioned before, you do not have to have the `defines in the model itself. You could have avoided declaring all the three `defines in our code and instead provided those on the command line of the simulator (compile time command line). For example, you could do the following on the command line:

```

+define+true
+define+false
+define+true+false+behavioral //all three with one +define+

```

Providing these `defines on the command line is a huge plus. That way you do not have to change the source code at all. You can conditionally compile out code

from the command line itself. This is the most popular way of providing these defines to the simulator.

## 27.3 `timescale

This is a widely used compiler directive, mostly at the testbench level. It specifies the time unit and time precision of the design elements (and #<delay>) that follow it. The time unit is the unit of measurement for time values such as the simulation time and delay values.

The syntax is:

```
`timescale time_unit / time_precision
```

The time\_unit argument specifies the unit of measurement for times and delays, and the time\_precision argument specifies how delays are rounded before being used in simulation. For example:

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

And you have #1.123ns a = 0; in your code. This means that the time\_unit is nanosecond (ns), and the time precision is 1 picosecond (ps). So, the simulator will convert 1.123 time unit into 1123 time ticks, i.e., convert 1.123 ns into 1123 ps.

`timescale being a compiler directive is global in nature. Once defined it will span file/module boundaries. All or no module must have timescale. It is compilation order dependent. For example, assume you have three files, TS1.v, TS2.v, and TS3.v:

### TS1.v

```
`timescale 1 ns/1 ns
module TS1 (...);
```

### TS2.v

```
`timescale 10 ns/1 ns
module TS2(...);
```

### TS3.v

```
//NO TIMESCALE
module TS3(...);
```

Here's how the `timescale is evaluated based on compile order:

verilog TS1.v TS2.v TS3.v //TS3 timescale is 10 ns/1 ns.

verilog TS2.v TS1.v TS3.v //TS3 timescale is 1 ns/1 ns.

verilog TS3.v TS1.v TS2.v //Compile ERROR – TS3 has no timescale.

There are system tasks available that help you see the time unit/precision in play during simulation.

`$printtimescale` – system task to display the time unit and time precision of the simulation.

`$time`, `$stime`, and `$realtime` system functions and the `$timeformat` (see Sect. 24.3.2) system task, along with the `%t` format specification, you can display the time information in play during simulation.

Let us look at a simple example:

```
`timescale 1ns/1ps
module timeIt;
    logic a, b, c;

    initial begin
        a = 1;
        #1.123ns a= 0;
    end

    initial $monitor("$realtime=%0t $time=%0t a = %0d", $realtime, $time, a);
endmodule
```

*Simulation log:*

```
$realtime=0 $time=0 a = 1
$realtime=1123 $time=1000 a = 0
```

V C S   S i m u l a t i o n   R e p o r t

We are using ``timescale 1 ns/1 ps`. So, the time unit is 1 ns and the time precision is 1 ps. So, the delays `#1.123 ns` will be converted to 1123 ps by the simulator. We see this from the `$display` in the simulation log. Note that `$realtime` will show the “unrounded” value of time, while `$time` will show the rounded value of time. `$realtime` shows that “a” changes at time 1123. `$time` rounds off the 1.123 ns to 1 ns and shows that “a” changes at time 1000.

Here is a similar example but with a different ``timescale`. Here we are using ``timescale 10 ns/1 ns`, meaning the time unit is greater than the time precision:

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

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

    initial begin
        a = 1;
        #1.55 a = 0;
```

```

#1.55 a = 1;
end

initial $monitor("$realtime=%0t a = %0d", $realtime, a);
endmodule

```

*Simulation log:*

```

$realtime=0 $time=0 a = 1
$realtime=16 a = 0
$realtime=32 a = 1

```

### V C S   S i m u l a t i o n   R e p o r t

Here `timescale 10 ns/1 ns means that the time unit for module test is 10 ns and time precision is 1 ns. As a result, the time values in the module are multiples of 10 ns, rounded to the nearest 1 ns; therefore, the value #1.55 is scaled to a delay of 16 ns. In other words, the value 0 is assigned to variable “a” at simulation time 16 ns ( $1.6 \times 10$  ns), and the value 1 at simulation time 32 ns. Here are the steps that took place.

The value #1.55 is rounded from 1.55 to 1.6 because the time precision is 1 ns.

The time unit of the module is 10 ns, so the #1.55 delay is scaled from 1.6 to 16.

The assignment of 0 to variable “a” is scheduled at simulation time 16 ns and the assignment to 1 at simulation time 32 ns.

*Note that you cannot do `timescale 1 ns/10 ns – the time precision cannot be larger than the time unit.*

You will get the following error from the simulator (Synopsys – VCS):

Error-[TPLTTU] Time precision larger than unit

It is illegal to set the time precision larger than the time unit.

Time unit : 1 ns

Time Precision : 10 ns

Note that \$timeformat can also be used to display times in specific time units. \$timeformat is discussed in Sect. 24.3.2.

As we saw, `timescale is compile order dependent. SystemVerilog added two further keywords to keep timescale local to the module where they are defined. SystemVerilog allows you to include a *timeunits* declaration, which consists of a *timeunit*, a *timeprecision*, or both as the first statement(s) in a module (before any declarations, except ports in an ANSI-style module). The timeunit declaration is applicable only to the module in which it occurs. It is also applicable to nested modules:

```

timeunit <time>
timeprecision <time>

```

They work exactly as `timescale time\_unit/time\_precision. For example:

```

module TU (...);
timeunit 1ns;
timeprecision 1ps;
always #10 ... //Means 10ns
...

```

#10 ns will be converted to 1 ps precision.

## 27.4 `default\_nettpe

As we know, by default, a net type is “wire.” But you can override the default net type with `default\_nettpe compiler directive:

```
`default_nettpe default_nettpe_value
```

where default\_nettpe\_value can be any of the net types, such as wire, tri, tri0, tri1, wand, triand, wor, trior, trireg, uwire, or “none.” When it is set to “none,” all nets have to be explicitly declared (else you will get an error). The directive must be defined outside of a design element (e.g., outside of a “module”). Multiple `default\_nettpe are allowed. The last occurrence controls the type of nets. The `resetall directive resets the directive to its default value.

## 27.5 `resetall

The `resetall resets all existing compiler directives to the default values. What you do is, before a module (any design element), you use a `resetall directive to reset all previously defined compiler directives and redefine those that are of interest and/or with different compiler directive values. This ensures that only directives that are desired in compiling a particular source file are active. Note that `define and `include do not have a default value and hence are not affected by `resetall.

# Bibliography

- Accellera: Accellera Standard V2.8.1 Open Verification Library (OVL) (n.d.)  
Ahmed Yehia, Mentor Graphics.: Mentor Graphics (n.d.)  
Amiq, Consulting: (n.d.)  
ChipVerify.: <https://www.chipverify.com/systemverilog/systemverilog-queues> (n.d.)  
Cummings, C.: Sunburst Design (n.d.-a)  
Cummings, C.: [www.sunburst-design.com](http://www.sunburst-design.com) (n.d.-b)  
Goel, P.: Retrieved from StackOverflow (n.d.)  
Guide, V.: Retrieved from [verificationguide.com](http://verificationguide.com) (n.d.)  
LRM, S.: IEEE 1800 SystemVerilog Language Reference Manual (2012)  
McGregor, M.: McGregor, Mike (KLA) (n.d.)  
Mentor: Mentor Graphics Coverage Cookbook. Mentor Graphics (n.d.)  
Methodology: Universal Verification: (n.d.)  
OVL: Open Verification Library. Open Verification Library (n.d.)  
Pro, V.: Verilog and SystemVerilog Resources for Design and Verification (n.d.-a)  
Pro, V.: Verilog and SystemVerilog Resources for Design and Verification (n.d.-b)  
Rich, Dave.: Mentor Graphics (n.d.)  
Sutherland, S.: SystemVerilog for Design (n.d.-a)  
Sutherland, S.: SystemVerilog for Design, Second Edition (n.d.-b)  
SystemVerilog\_LRM\_1800-2012: SystemVerilog LRM 1800-2012 (n.d.)  
SystemVerilog-LRM: IEEE 1800 Standard for SystemVerilog – 2017 (n.d.)  
testbench.: Retrieved from [www.testbench.in](http://www.testbench.in) (n.d.)  
Verification, U.: <http://www.learnuvmlibrary.com/>. Retrieved from <http://www.learnuvmlibrary.com/> (n.d.)  
Xilinx: FPGA Company (n.d.)

# Index

## 0-9, and Symbols

#-, 534–536	\$dumplimit, 819
##[m:n], 465	\$dumpoff, 819
##[m:n] -clock delay range, 468–474	\$dumpon, 819
##m, 465, 466	\$dumpports, 820
##m – clock delay, 466	\$dumpvars, 818
#=#, 534	\$error, 451, 773
\$assertfailoff, 548–549	\$falling_gclk, 532
\$assertfailon, 548–549	\$fatal, 451, 773
\$assertkill, 514	\$fclose, 792–796
\$assertnonvacuouson, 548–549	\$fdisplay, 761, 797
\$assertoff, 514	\$fell, 459
\$assertton, 514	\$fell_gclk, 534
\$assertpassoff, 548–549	\$fgetc, 800–802
\$assertpasson, 548–549	\$fgets, 800–802
\$assertvacuousoff, 548–549	\$finish, 759
\$bits : Expression size system function, 766	\$fmonitor, 761, 792, 797
\$bitstoreal, 764	\$fopen, 792–796
\$bitstoshortreal, 764	\$fread, 808–809
\$changed, 532	\$fscanf, 802–808
\$changed_gclk, 534	\$fstrobe, 761, 797
\$changing_gclk, 532	\$future_gclk, 532
\$countbits, 512–513, 771	\$fwrite, 761, 792, 797
\$countones, 511–512, 771	\$info, 451, 773
\$Display, 786	\$isunknown, 511, 771
\$dist_chi_square, 774	\$itor, 764
\$dist_erlang, 774	\$Monitor, 786
\$dist_exponential, 774	\$onehot, 510–511, 771
\$dist_normal, 774	\$onehot0, 510–511, 771
\$dist_poisson, 774	\$past, 461–465
\$dist_t, 774	\$past_gclk, 534
\$dist_uniform, 774	\$printtimescale, 761
\$dumpall, 819–820	\$q_add, 776
\$dumpfile, 817–818	\$q_exam, 777
\$dumpflush, 819	\$q_full, 777
	\$q_initialize, 776

\$q\_remove, 776–777  
 \$random, 773, 774  
 \$random and Probabilistic Distribution Functions, 773–775  
 \$readmemb, 809–812  
 \$readmemh, 809–812  
 \$realtime, 760  
 \$realtobits, 764  
 \$rising\_gclk, 532  
 \$root, 249  
 \$rose, 457–459  
 \$rose\_gclk, 534  
 \$rtoi, 763  
 \$sformat, 797–799  
 \$shortrealtobits, 764  
 \$signed, 765  
 \$sscanf, 802–808  
 \$stable, 460–461  
 \$stable\_gclk, 534  
 \$steady\_gclk, 532  
 \$stime, 760  
 \$Strobe, 786  
 \$swrite, 797–799  
 \$test\$plusargs, 814–817  
 \$time, 760  
 \$timeformat, 761–763  
 \$ungetc, 800–802  
 \$unsigned, 765  
 \$urandom() and \$urandom\_range(), 377–379  
 \$value\$plusargs, 814–817  
 \$warning, 451, 773  
 \$Write, 786  
 \$writememb, 813–814  
 \$writememh, 813–814  
 .Matched, 528–530  
 .Triggered, 525–530  
 [\*], 478  
 [\*m:n], 465, 476–483, 485, 486  
 [\*m], 465, 474–476  
 [+], 478  
 [=m:n], 465, 484–486  
 [=m], 465, 482–484  
 [->m:n], 465  
 [->], 486–487  
 [->m], 465  
 `Default\_nettpe, 842  
 `Define, 831–836  
 `Else, 837, 838  
 `Elsif, 837, 838  
 `Endif, 837, 838  
 `Ifdef, 837, 838  
 `Ifndef, 837, 838  
 `Resetall, 842  
 `Timescale, 839–842  
 `Undef, 836, 837  
 `Undefineall, 836, 837  
 |=>, 466  
 |->, 413, 463, 466

**A**

Abort properties, 545–548  
 Abstract class, 205–211  
 Accept\_on, 545–548  
 Active region, 443  
 Always @ (\*), 609–610  
 Always\_comb, 606–609  
 Always\_ff, 612–614  
 Always\_latch, 610–612  
 Always' procedural block, 603–605  
 Always property, 536–537  
 ANSI style *first* port rules, 239–240  
 ANSI style module header, 237–239  
 ANSI style *subsequent* port rules, 240–241  
 Antecedent, 432, 433  
 Application, 449  
 Argument binding by name, 737–738  
 Arithmetic operators, 292–295  
 Array locator methods, 92–98  
 Array manipulation methods, 92–103  
 Array ordering methods, 99–100  
 Array querying system functions, 766–768  
 Array reduction methods, 100–103  
 Arrays, 61–103  
 Assert #0, 426  
 Assert final, 426  
**Assertions**  
 clocking basics, 436–446  
 eventually, 538–539  
 expect, 530–532  
 methodology components, 417–422  
*not* operator, 508–509  
 operators, 465–510  
 s\_eventually, 538–539  
 severity levels, 451–452  
 system functions and tasks, 510–514  
 Assertions and OVL, 415  
 Assertions in static formal, 416–417  
 Assertion types, 422  
 Assign, 753–755  
 Assign and deassign, 753–755  
 Assigning, indexing and slicing of arrays, 72–74  
 Assignment operators, 289  
 Associative array, 83–92  
 class index, 85–86  
 string index, 85  
 wild card index, 84

- Associative array methods, 87–90  
Assume #0, 429  
Assume final, 429  
Asynchronous abort, 545  
Automatic functions, 728–730  
Automatic tasks, 718–724  
Automatic variable, 28  
Automatic *vs.* Static variable, 27–29
- B**  
Base class, 156–162  
Base class constructor, 169  
Bind, 452  
Binding, 453–455  
Binding properties, 452–455  
Binding properties to design ‘module’ internal signals, 454  
Bins, 564–568  
Bins’ filtering using the ‘with’ clause, 567–568  
Bins’ for transition coverage, 581–583  
Binsof, 588–591  
Binsof’ and ‘intersect’, 588–591  
Bits *vs.* Bytes, 10  
Bit-vector system functions, 771–773  
Bitwise binary AND (&) operator, 303  
Bitwise binary exclusive NOR operator, 304  
Bitwise binary exclusive OR (^) operator, 303  
Bitwise binary OR (|) operator, 303  
Bitwise Operators, 303–305  
Bitwise unary negation (~) operator, 304, 305  
Blocking statement, 530  
Blocking *vs.* non-blocking procedural assignments, 746–751  
Bounded queue, 107  
Break’ and ‘continue’, 662–663  
Break and return, 401–404
- C**  
Case statements, 637–652  
Cased and casez and do not care, 640–644  
Casez, 640–644  
Chain constructor, 169–172  
Checkers  
    illegal conditions, 696–698  
    in a package, 704  
    instantiation rules, 701–702  
    legal conditions, 695–696  
    rules for ‘formal’ and ‘actual’ arguments, 703  
Chip functionality Assertions, 418  
Chip interface Assertions, 418
- Class, 155–233  
    ‘const’ class properties, 217–221  
    extending parameterized class, 232–233  
    local members, 212–215  
    parameterized class, 225–233  
    parameterized class with static properties, 229–232  
    protected members, 215–217  
    type parameters, 227–229  
    value parameters, 226–227  
Class assignment, 184–185  
Class constructor, 169–172  
Class scope resolution operator, 221–225  
Clock delay range, 468–474  
Clock domain crossing (CDC), 514  
Clocking basics, 436–446  
Clocking basics – clock in ‘assert’, ‘property’ and ‘sequence’, 438  
Clocking block in an interface, 286–287  
Clocking blocks, 677–688  
Clocking blocks with interfaces, 682–685  
Code coverage, 327, 554–556  
Concatenation Operators, 310–311  
Concurrent assertion operators, 465  
Concurrent assertions, 429–435  
Concurrent assertions are multi-threaded, 447–448  
Conditional event control, 627–628  
Conditional operators, 308–310  
Consecutive repetition operator, 474–476  
Consecutive repetition range operator, 476–483, 485, 486  
Consequent, 432, 433  
Constant expression in ‘case’ statement, 644–646  
Const’ class properties, 217–221  
Constrained random test generation, 325–407  
Constrained random verification (CRV), 325  
Constraint blocks, 343–361  
Constraint\_mode () – control constraints, 370–372  
Constraints, 329–337  
Continuous assignment, 751–753  
Control constraints, 370–372  
Conversion functions, 763–765  
Conversion to/from signed/unsigned expression, 765  
Copying of dynamic arrays, 79–80  
Cover #0, 429  
Coverage options for ‘covergroup’ type, 598  
Coverage system tasks, functions and methods., 598–600  
Cover final, 429  
Covergroup, 558–559

Covergroup – formal and actual arguments, 568–569

Coverpoint, 560–564

Coverpoint using a function or an expression, 563–564

Cross coverage, 575–581

## D

Data hiding ('local', 'protected', 'const'), 211–221

Data types, 5–59

Deassign, 753–755

Deep copy, 189–198

Default argument values, 735–737

Default clocking block, 438–442

Default\_explicit\_clocking, 440, 441

Default port values, 245–246

Deferred ‘assume’, 429

Deferred ‘cover’, 429

Deferred immediate assertions, 422, 426–429

*Defparam*, 251–252

Difference between [=m:n] and [→m:n], 487–489

Difference between ‘sequence’ and ‘property’, 456–457

Disable iff, 449–451

Disable (property) operator, 449–451

Disable statement, 628–630

Disabling random variables, 367–370

Display tasks, 785–788

Do – while’ loop, 659

Downcasting, 194–198

Dynamic array of arrays, 80–83

Dynamic array of associative arrays, 91–92

Dynamic array of queues, 118–120

Dynamic arrays, 74–83

Dynamic cast, 196

Dynamic casting, 57–59

## E

Edge detection, 458–459

Edge sensitive and performance implication, 460

Embedding concurrent assertions in procedural block, 549–552

Encapsulation, 211–221

End event, 486

Endpoint of a sequence (.matched), 528–530

Endpoint of a sequence (.triggered), 525–530

Enumerated type methods, 32–36

Enumerated types, 30–38

Enumerated type with ranges, 36–38

Equality Operators, 298–301

Escape identifiers, 788–789

Event comparison, 627

Event data type, 47–51

Event sequencing – wait\_order, 50–51

Eventually, 538–539

Exercise, 492, 569

Expect, 530–532

Export’ tasks in a modport, 282–283

Extended class, 162–169

Extending parameterized class, 232–233

Extern, 221–225

External constraint blocks, 343–345

## F

File I/O system tasks and functions, 792–799

Final, 614–615

Final’ procedure, 614–615

First\_match, 505

First\_match complex\_seq1, 465

Followed by property, 534–536

Force - release, 755–757

Foreach, 352–355

Foreach’ Loop, 655–658

Forever’ loop, 660–661

Fork-join, 615–620

Fork - join\_any, 617–619

Fork - join\_none, 619–620

For’ loop, 652–653

Formal arguments, 448–449

Format specifications, 789–792

4-D unpacked array, 67–68

Functional coverage, 415, 554–556

control-oriented, 556

data-oriented, 556

performance, 562

Functional coverage methodology, 557

Function called as a statement, 725–726

Function name or “return” to return a

value, 726–727

Functions, 724–730

Functions in constraints, 356–357

Future global clocking sampled value functions, 532–533

Future sampled value functions, 532

## G

Gating expression, 462

Generate

conditional Construct, 825–829

loop constructs, 822–825

Get\_coverage, 599

Get\_inst\_coverage, 596, 599

Get\_randstate, 380–385

Glitch, 425, 427  
Global clocking, 534, 685–688  
Global clocking sampled value functions, 532–533  
Global variables, 25–30  
Global vs. Local variables, 25–27

## H

Hardware design and verification language (HDVL), 2

## I

If – else-if statements, 631–637  
Iff' and 'implies', 509–510  
If (expression) property\_expr1 else property\_expr2, 466, 509  
Ignore\_bins, 583–588  
Illegal\_bins, 588  
Illegal\_data\_dependency, 523  
Immediate assert, 425  
Immediate assertions, 423–429  
Immediate assume, 425  
Immediate cover, 425  
Implication and if-else, 351–352  
Implication operator, 432, 434–435  
Import' of a package, 146  
Import' tasks in a modport, 280–281  
Increment and decrement operators, 290–292  
Inheritance, 162–169  
Inheritance memory allocation, 167–169  
Initial' procedural block, 602–603  
In-line random variable control, 372–375  
Inside' operator, 321–323  
Integer data types, 5–10  
Integer match functions, 769  
Integral data type, 5  
Interface, 267–274  
    tasks and functions in an interface, 278–283  
Interprocess synchronization, 665–676  
Intersect, 500, 588–591  
Iterative Constraint (*foreach*'), 352–355

## J

Jump Statements, 662–663

## L

Legal\_data\_dependency, 523  
Let'  
    in immediate and concurrent assertions, 709–713

local scope, 706–707  
    with parameters, 707–708  
Level sensitive time control, 620–624  
Local members, 212–215  
Localparam, 253–254  
Local scope resolution (local::), 367  
Local variables, 520–525  
Logical operators, 301–303  
Longint, 6–8  
Loop statements, 652–661

## M

Mailboxes, 668–676  
Math Functions, 769–771  
Merging of named events, 625–627  
Modport, 274–278  
    'export' tasks in a modport, 282–283  
    'import' tasks in a modport, 280–281  
Module, 235–255  
Module header definition, 236–244  
Module instantiation, 247–250  
Module interface Assertions, 418  
Module parameters, 250–255  
Multiple implications, 552  
Multiple threads, 468  
Multiply clocked properties – clock resolution, 518  
Multiply clocked properties – ‘not’-operator, 517–519  
Multiply clocked properties – ‘or’-operator, 517  
Multiply clocked sequences and properties, 514–520  
Multi-threaded, 447–448

## N

Named event time control, 624–627  
Nested checkers, 694–695  
Nested implications, 552  
Nested modules, 249–250  
Net data types, 12–19  
Net types, 22–23  
Nexttime, 541–545  
Non-ANSI style module header, 241–244  
Non-blocking statement, 530  
Non-consecutive GoTo repetition operator, 486–487  
Non-consecutive repetition, 482–484  
Non-consecutive repetition range operator, 484–486  
Non-overlapping implication operator, 435  
Not operator, 508–509  
Not <property\_expr>, 466

**O**

Object handle, 161  
 Object oriented programming (OOP), 155  
 Observed region, 443  
 1-D packed & 1-D unpacked array, 65–67  
 1-D packed & 3-D unpacked array, 68–69  
 Open verification library (OVL), 415  
 Operators, 289–323  
     arithmetic operators, 292–295  
     assignment operators, 289  
     bitwise operators, 303–305  
     concatenation operators, 310–311  
     conditional operators, 308–310  
     equality operators, 298–301  
     increment and decrement  
         operators, 290–292  
     'inside' operator, 321–323  
     logical operators, 301–303  
     relational operators, 296–297  
     replication operators, 311–312  
     set membership operator, 321–323  
     shift operators, 307–308  
     streaming operators, 312–321  
     wildcard equality operators, 300–301  
 Overlapping implication operator, 435  
 OVL Library, 416

**P**

Packages, 145–153  
 Packed and npacked unions, 135–143  
 Packed and unpacked arrays as arguments to  
     subroutines, 74  
 Packed arrays, 61–72  
 Packed structure, 122–126  
 Packed union, 135, 142–143  
 Packing of bits, 314–318  
 Parallel blocks  
     fork-join, 615–620  
 Parameter dependence, 254–255  
 Parameterized class, 225–233  
 Parameterized class with static  
     properties, 229–232  
 Parameterized interface, 283–286  
 Parameterized mailbox, 675–676  
 Parameterized tasks and functions, 738–743  
 Parameterizing coverpoints in a class, 574–575  
 Pass by reference, 732–735  
 Pass by value, 730–732  
 Passing arguments by value or reference to  
     tasks and functions, 730–735  
 Passing values between productions, 404–407  
 Past global clocking sampled value  
     functions, 533–534

Past sampled value functions, 533

Performance, 409, 469  
 Performance implication assertions, 418  
 Polymorphism, 198–202  
 Post-randomization, 362–367  
 Preponed region, 443–446  
 Pre-randomize and post-  
     randomization, 362–367  
 Priority-case, 648–652  
 Priority-if, 636–637  
 Probabilistic distribution functions, 774–775  
 Procedural continuous assignment – assign  
     and deassign, 753–755  
 Procedural programming statements, 631–633  
 Productions, 391  
 Program, 257–265  
 Property, 456  
 Protected, 211–221  
 Protected members, 215–217  
 Pure virtual method, 205–211  
 Push' and 'pop' of a queue, 110

**Q**

Qualifying event, 479, 482–484, 486  
 Querying for coverage, 596–597  
 Queue management stochastic analysis  
     tasks, 775–783  
 Queue methods, 109–116, 118  
 Queue of queues, 118–120  
 Queue of systemverilog classes, 117–118  
 Queues, 105–120

**R**

Rand, 337–342  
 Randc, 337–342  
 Randcase, 389–391  
 Rand join, 397–400  
 Rand\_mode ( ) - disabling random  
     variables, 367–370  
 Randomization Methods, 361–367  
 Randomize, 372–375  
 Randomizing arrays and queues, 340–341  
 Randomizing object handles, 342  
 Random number generation system functions  
     and methods, 376–385  
 Random number generator (RNG), 376  
 Random production weights and if-else  
     statement, 393–395  
 Random stability, 385–388  
 Random variables (rand and randc), 337–342  
 Randsequence, 391–407  
 Reactive region, 443

- Reading data from a file, 800–814  
Real' data type conversion functions, 11–12  
Real data types, 10–12  
Real math functions, 769–771  
Reduction unary AND operator, 306  
Reduction unary exclusive OR operator, 306  
Reduction unary OR operator, 306  
Reject\_on, 545–548  
Relational operators, 296–297  
Repeat loop, 654–655  
Repeat production statement, 395–397  
Replication operators, 311–312  
Resolved *vs.* Unresolved type, 16  
Reusability, 414  
RTL Assertions, 418
- S**  
S\_always property, 536–537  
Sampled value, 444, 446  
Sampled value functions, 457–465  
Sampled variable, 444  
Sample( ) method, 591–596  
Sampling edge, 431, 436  
Sampling edge (clock edge), 442–443  
Scope visibility, 453–455  
Semaphores, 665–668  
Seq1 and seq2, 465, 495–498  
Seq1 intersect seq2, 465, 500  
Seq1 or seq2, 465, 498–500  
Seq1 within seq2, 465, 492  
Sequence, 456  
Set membership operator, 321–323  
Set\_inst\_name, 599  
Set\_randstate, 380–385  
S\_eventually, 538–539  
Severity levels, 451–452  
Severity system tasks, 773  
Shallow copy, 186–189  
Shift Operators, 307–308  
Shortint, 6–8  
Sig1 throughout seq1, 465  
Simulation control system tasks, 759, 760  
Simulation glitches, 426  
Simulation time system functions, 760, 761  
Slicing of arrays, 72–74  
S\_nexttime, 541–545  
Srandom( ), get\_randstate( ) and set\_randstate( ), 380–385  
Static and automatic functions, 728–730  
Static and automatic tasks, 718–724  
Static, automatic, local and global variables, 25–30
- Static casting, 51–57  
Static constraint, 336  
Static methods, 176–179  
Static properties, 172–176  
Static random variables, 339  
Static variable, 27–29  
Streaming operators (pack/unpack), 312–321  
String data type, 40–47  
String methods, 44–48, 50  
String operators, 41–43  
Strobe ( ) method, 597  
Structure as an argument to task or function, 130–131  
Structure as module I/O, 128–130  
Structure packed, 122–126  
Structures, 121–131  
Structure unpacked, 126–128  
Structure within a structure, 131  
S\_until, 539–541  
S\_until\_with, 539–541  
Super.new( ), 169–172  
Sync\_accept\_on, 545–548  
Sync\_reject\_on, 545–548  
Systemverilog class based coverage, 569–575  
Systemverilog functional coverage, 553–600  
Systemverilog 'interface, 267–287  
Systemverilog language, 1  
Systemverilog module, 235–255  
Systemverilog operators, 291  
Systemverilog processes, 601–630  
Systemverilog 'program, 257–265
- T**  
Tagged unions, 140–143  
Tasks and functions in an interface, 278–283  
Terminal production, 391  
Testbench races, 261–265  
This, 180–183  
3-D packed & 1-D unpacked array, 71–72  
3-D packed array, 63–65  
Throughout, 489–492  
Timescale System Tasks, 761–763  
Transition coverage, 581–583  
Tri0, 17–19  
Tri1, 17–19  
Triand, 17  
Trior, 17  
Tri-state Logic, 14  
2-D packed & 2D-unpacked array, 69–71  
2-D packed array, 62–63  
Typedef, 38–40  
Type parameters, 227–229

**U**

Unary Reduction Operators, 305–306  
Unbounded queue, 107  
Union, 133–143  
    packed and unpacked unions, 135–143  
    tagged, 140–143  
Unique-case, 648–652  
Unique0-case, 648–652  
Unique constraint, 348–350  
Unique-if, 633–635  
Unique0-if, 633–635  
Unpacked arrays, 61–72  
Unpacked structure, 126–128  
Unpacked unions, 136–140  
Unpacking of bits, 318–321  
Unresolved type, 16  
Until, 539–541  
Until\_with, 539–541  
Upcasting and downcasting, 194–198  
User defined type  
    typedef, 38–40  
Uwire, 15–16

**V**

Value change dump (VCD), 817–820  
Value change dump (VCD) file, 817–820  
Value parameters, 226–227  
Var, 23  
Variable and Net initialization, 23–25  
VHDL DUT binding with SystemVerilog  
    assertions, 455  
Virtual (abstract) class, 205–211  
Virtual methods, 203–205  
Void functions, 727

**W**

Wait fork, 622–624  
Wait\_order, 50–51  
Wand, 17  
Weighted Distribution, 345–348  
While loop, 658–659  
Wildcard bins, 583  
Wildcard equality operators, 300–301  
Wire' and 'tri, 13–15  
With function, 595  
Within, 492  
Wor, 17