

SystemVerilog for Design and Verification

Engineer Explorer Series

Course Version 21.10

Lecture Manual

Revision 1.0

© 1990-2022 Cadence Design Systems, Inc. All rights reserved.

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522.

All other trademarks are the property of their respective holders.

Restricted Print Permission: This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

The publication may be used solely for personal, informational, and noncommercial purposes;

The publication may not be modified in any way;

Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and

Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence customers in accordance with, a written agreement between Cadence and the customer.

Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Table of Contents

SystemVerilog for Design and Verification

Module 1	About This Course	3
Module 2	SystemVerilog Overview	10
Module 3	Standard Data Types and Literals	16
Module 4	Procedures Statements and Procedural Blocks	32
	Lab 1 Modeling a Simple Register	
	Lab 2 Modeling a Simple Multiplexor (Optional)	
Module 5	Operators	55
	Lab 3 Modeling a Simple Counter	
Module 6	User-Defined Data Types and Structures	67
	Lab 4 Modeling a Sequence Controller	
Module 7	Hierarchy and Connectivity.....	85
	Lab 5 Modeling an Arithmetic Logic Unit (ALU) (Optional)	
Module 8	Static Arrays	103
Module 9	Tasks and Functions	111
	Lab 6 Testing a Memory Module	
Module 10	Interfaces	129
	Lab 7 Using a Memory Interface	
	Lab 8 Verifying the VeriRISC CPU (Optional)	
Module 11	Simple Verification Features	151
Module 12	Clocking Blocks	168
	Lab 9 Using a Simple Clocking Block	

Module 13	Random Stimulus.....	182
	Lab 10	Using Scope-Based Randomization
Module 14	Basic Classes	197
	Lab 11	Using Classes
Module 15	Polymorphism and Virtuality.....	224
	Lab 12	Using Class Polymorphism and Virtual Methods
Module 16	Class-Based Random Stimulus	237
	Lab 13	Using Class-Based Randomization
Module 17	Interfaces in Verification.....	260
	Lab 14	Using Virtual Interfaces
Module 18	Covergroup Coverage.....	272
	Lab 15	Simple Covergroup Coverage
	Lab 16	Analyzing Cross Coverage (Optional)
Module 19	Queues and Dynamic and Associative Arrays (QDA).....	397
	Lab 17	Using Dynamic Arrays and Queues
Module 20	Introduction to Assertion-Based Verification (ABV)	316
Module 21	Introduction to SystemVerilog Assertions.....	326
	Lab 18	Simulating Simple Implication Assertions
	Lab 19	Sequence-Based Properties
Module 22	Direct Programming Interface (DPI).....	353
	Lab 20	Simple DPI Use
Module 23	Interprocess Synchronization.....	374
	Lab 21	Synchronizing Processes with a Mailbox
	Lab 22	Synchronizing Processes with a Semaphore
	Lab 23	Synchronizing Processes with Events (Optional)

Module 24	Next Steps	393
Appendix A	Verilog-2001 Summary	397
Appendix B	SystemVerilog Event Scheduler	430
Appendix C	Programs	434
Appendix D	Miscellaneous Features.....	442
Appendix E	SystemVerilog-2012 Features.....	458
Appendix F	DPI Reference.....	476



SystemVerilog for Design and Verification

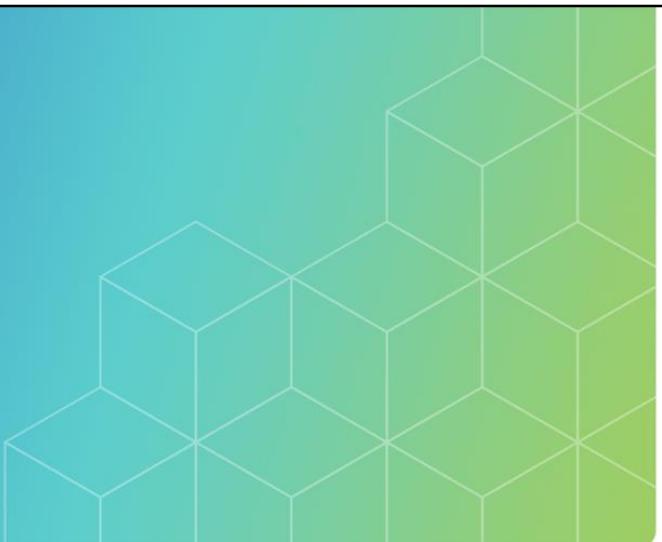
Engineer Explorer Series

Version 21.10

Estimated time: 5 Days

cadence®

This page does not contain notes.



About This Course

Module **1**

Revision **1.0**
Version **21.10**

Estimated Time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Course Prerequisites

Before taking this course, you need to have:

- Taken a Verilog language course
- A good working knowledge of Verilog

Knowledge of the following is useful, but not essential:

- An appreciation of Object-Oriented Modeling (OOM)
- Some familiarity with the Cadence® Xcelium™ simulator



This page does not contain notes.

Course Objectives

In this course, you

- Explore the new SystemVerilog features for design and verification:
 - Creating code using new standard and user-defined data types
 - Using new procedural statements and operators
 - Modifying design and testbench modules to use interfaces
 - Writing object-oriented verification code using classes
 - Creating stimulus using constrained randomization
 - Defining coverage to measure the effectiveness of stimulus
 - Adding dynamic array objects to testbench code
 - Writing assertions to check design behavior
 - Integrating C code into a SystemVerilog testbench



This page does not contain notes.

Course Agenda

- SystemVerilog Overview
- Standard Data Types and Literals
- Procedural Statements and Procedural Blocks
 - Lab: Modeling a Simple Register
 - Lab: Modeling a Simple Multiplexor (Optional)
- Operators
 - Lab: Modeling a Simple Counter
- User-Defined Data Types and Structures
 - Lab: Modeling a Sequence Controller
- Hierarchy and Connectivity
 - Lab: Modeling an Arithmetic Logic Unit (ALU) (Optional)
- Static Arrays
- Tasks and Functions
 - Lab: Testing a Memory Module
- Interfaces
 - Lab: Using a Memory Interface
 - Lab: Verifying the VeriRISC CPU (Optional)
- Simple Verification Features
- Clocking Blocks
 - Lab: Using a Simple Clocking Block
- Random Stimulus
 - Lab: Using Scope-Based Randomization
- Basic Classes
 - Lab: Using Classes
- Polymorphism and Virtuality
 - Lab: Using Class Polymorphism and Virtual Methods
- Class-Based Random Stimulus
 - Lab: Using Class-Based Randomization
- Interfaces in Verification
 - Lab: Using Virtual Interfaces
- Covergroup Coverage
 - Lab: Simple Covergroup Coverage
 - Lab: Analyzing Cross Coverage (Optional)
- Queues and Dynamic and Associative Arrays (QDA)
 - Lab: Using Dynamic Arrays and Queues
- Introduction to Assertion-Based Verification (ABV)
- Introduction to SystemVerilog Assertions (SVA)
 - Lab: Simulating Simple Implication Assertions
 - Lab: Sequence-Based Properties
- Direct Programming Interface (DPI)
 - Lab: Simple DPI Use
- Interprocess Synchronization
 - Lab: Synchronizing Processes with a Mailbox
 - Lab: Synchronizing Processes with a Semaphore
 - Lab: Synchronizing Processes with Events (Optional)

6 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Software and Licenses

For the software and licenses used in the labs for this course, go to:

https://www.cadence.com/en_US/home/training/all-courses/82143.html

If there is additional information regarding the specific software, it is detailed in the lab document and/or the README file of the database provided with this course.



This page does not contain notes.

Become Cadence Certified by Earning a Digital Badge



Digital badges indicate mastery in a certain technology or skill and give managers and potential employers a way to validate your expertise.

- Cadence Training Services offers digital badges for our popular training courses.
- Your digital badge can be added to your email signature or social media platforms like LinkedIn or Facebook.

Benefits of Cadence Certified Digital Badges

- Validate expertise
 - Expand career opportunities
- Professional credibility
 - Stand apart from your peers
- For more information, go to www.cadence.com/training or email es_digitalbadge@cadence.com.



How do I register to take the exam?

- Log in to our [Learning Management System](#) to locate the exam in your transcript.

How long will it take to complete the exam?

- Most exams take 45 to 90 minutes to complete. You may retake the exam multiple times to pass the exam.

How do I access and use the digital badge?

- After you pass the exam, you get a digital badge and instructions on how to place it on social media sites.

How is the digital badge validated?

- [Credly](#) validates the digital badge as issued to you by Cadence and includes the details of the criteria you completed to earn the badge.



8 © Cadence Design Systems, Inc. All rights reserved.

This page does not contain notes.

Icons Used in This Class



Best Practice



Language Syntax



Concept/
Glossary



Frequently Asked
Questions /
Quiz



Error Message



Problem & Solution



Quick Reference



Tool Command



How To



Lab List

Throughout this class, we use icons to draw your attention to certain kinds of information. Here are the icons we use, and what they mean.

This page does not contain notes.

SystemVerilog Overview

Module **2**

Revision **1.0**
Version **21.10**

Estimated time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Module Objectives

In this module, you

- Identify the purpose of SystemVerilog
- Analyze the segmentation of SystemVerilog constructs



This page does not contain notes.

What Is SystemVerilog?



SystemVerilog is a unified hardware design, specification, and verification language based on the Accellera SystemVerilog 3.1a extensions to the Verilog hardware description language.

It is a set of extensions to Verilog-2001, including the following:

- Added data types and relaxation of rules on existing data types
- Higher abstraction-level modeling features
- Language enhancements for synthesis
- Interfaces to model communication between design blocks
- Language features to enable verification methodologies
 - Assertions, constrained randomization, functional coverage
- Lightweight interface to C/C++ programs

Aim

- To define a concise, unified language for design and verification using a single simulation tool
 - Assuming tool support for enhanced testbench and verification features



SystemVerilog is the IEEE Std. 1800 “IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language”.

SystemVerilog includes the IEEE Std. 1364-2001 “IEEE Standard Verilog Hardware Description Language” by reference.

Several companies donated proprietary content to the Accellera EDA advocacy group, which integrated those Verilog extensions into SystemVerilog and championed the creation of the new IEEE standard.

Quick Reference Guide: SystemVerilog Language Extensions



SystemVerilog IEEE 1800

Literals	Arrays	Classes	Statements	Processes	Random	Clocking	Coverage	Interfaces
time string pattern	packed unpacked dynamic associative queue	new static this/super extends protected cast local	unique priority do while foreach return break continue final iff	always_comb always_latch always_ff join_any join_none wait fork static automatic const ref void	rand/randc constraint randomize() solve before dist with rand_mode constraint_mode randcase randsequence semaphore mailbox	clocking ##N	covergroup coverpoint cross wildcard sequence bins illegal_bins	interface virtual modport export import
Types logic bit byte shortint int longint shortreal void	Declarations const alias type var	Operators assignment wild equality tagged overloading						DPI export import context pure
								Program
Verilog-2001								
ANSI C style ports generate localparam constant functions			standard file I/O \$value\$plusargs `ifndef `elsif `line @*			** (power operator) configurations memory part selects variable part select		
Verilog-1995								
modules parameters function/tasks always @ assign			\$finish \$fopen \$fclose \$display \$write \$monitor `define `ifdef `else `include `timescale			initial disable events wait # @ fork-join		
+ = */ % >> <<								

13 © Cadence Design Systems, Inc. All rights reserved.



The IEEE Std. 1364 standardized in 1995 was previously the proprietary Verilog hardware description language.

The 2001 update to the Verilog standard added several convenience features, still primarily targeted at hardware design.

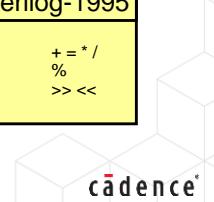
Accellera proposed an update to the Verilog standard that added many design convenience features and more than tripled the verification features. This slide illustrates just how much SystemVerilog adds to Verilog. To hopefully minimize confusion, Accellera requested and obtained the new IEEE standard number 1800 for these new features.



Quick Reference Guide: SystemVerilog Segmentation

			Verification Extensions
Classes and Object-Oriented Design Dynamic Data Structures Scope and Class Based Constrained Randomization Intraprocess Synchronization Direct Programming Interface (DPI)	Assertions Coverage Clocking Blocks Program Blocks	Increase Verification Efficiency!	
			Design Subset
Relaxed Datatype Rules Static Data Types Interfaces Specialized Procedures	Procedural Statements Connectivity Enhancements Convenience Features	Express more with less code! Increase synthesis quality!	
ANSI C style ports generate localparam constant functions	standard file I/O \$value\$plusargs 'ifndef `elsif `line @*	(* attributes *) configurations memory part selects variable part select	multi dimensional arrays signed types automatic ** (power operator)
modules parameters function/tasks always @ assign	\$finish \$fopen \$fclose \$display \$write \$monitor 'define 'ifdef 'else 'include 'timescale	initial disable events wait # @ fork-join	wire reg integer real time packed arrays 2D memory
modules parameters function/tasks always @ assign		modules parameters function/tasks always @ assign	+ = * / % >> <<

14 © Cadence Design Systems, Inc. All rights reserved.



Obviously, most SystemVerilog enhancements are for verification, have no hardware implementation and therefore are not synthesizable. These are the SystemVerilog verification extensions and include classes, coverage, randomization and assertions.

However, there are several SystemVerilog enhancements that are intended to fix common Verilog synthesis issues and which have hardware implementation-defined by the Language Reference Manual (LRM). This is the design subset of SystemVerilog intended for RTL code, and includes relaxed and enhanced data types, additional procedural statements and connectivity enhancements.

Obviously, there are many features that can be used in both verification and design. There is a large “grey area” of language features for which synthesizability and implementation are not defined by the LRM but depend on the synthesis tool you are using. There are even constructs that are definitely synthesizable with a known implementation in one form but vendor-specific in another. This can affect the portability of design code in that some constructs may be synthesizable in one tool, but not in another, or may be synthesizable, but implemented differently in different tools.

Normally, a company will make a decision as to whether portability and implementation are important and refine their existing RTL coding guidelines accordingly.

Module Summary

SystemVerilog has a lot of new content, including:

- Nearly 100 new keywords

SystemVerilog is a substantial upgrade on Verilog:

- Based heavily on Verilog-2001

There is a huge learning curve for both designers and tool vendors.

- Parts of the language are still being refined
- Tools may not support some language constructs
- Most users will adopt SystemVerilog incrementally

This course covers the most useful, the most frequently used, and the most widely supported features of the language.



Many design teams have not yet fully adopted the Verilog-2001 standard. As SystemVerilog is based on Verilog-2001, such teams will need to learn about Verilog-2001 as well as SystemVerilog.

Having many new keywords means that existing Verilog designs may not compile with SystemVerilog compilers. For example, if you have a signal named *do*, then your code will not compile in SystemVerilog because *do* is a SystemVerilog keyword.

However, SystemVerilog does not take features away from Verilog users. Verilog-2001 features are still available and work the same as earlier.

Standard Data Types and Literals

Module **3**

Revision **1.0**
Version **21.10**

Estimated time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Module Objectives

In this module, you

- Use the new `logic` data type
- Analyze the relaxed rules for data types in SystemVerilog
- Examine two-state data types and their uses
- Differentiate the changes between time literals and timing specification



This page does not contain notes.

What Is a Datatype?



A datatype is a set of values (2-state or 4-state) that can be used to declare data objects or to define user-defined data types.

- The Verilog datatypes have 4-state values: (0, 1, Z, X).
- SystemVerilog adds 2-state value types based on `bit`:
 - Has values 0 and 1 only.
 - Direct replacements for `reg`, `logic` or `integer`.
 - Greater efficiency at higher-abstraction level modeling (RTL).

Newly introduced SV 2-state datatypes

Type	Description	Sign
<code>bit</code>	Single bit, Scalable to vector	Default unsigned
<code>byte</code>	8-bit vector or ASCII character	Default signed
<code>shortint</code>	16-bit vector	Default signed
<code>int</code>	32-bit vector	Default signed
<code>longint</code>	64-bit vector	Default signed



All Verilog data types except `real` are 4-state logic. 4-state logic is essential for gate-level modeling and for initialization at RTL, but simulating everything in 4-state logic can add significant performance overhead.

Therefore SystemVerilog adds a 2-state `bit` type, containing only 0 and 1 values, with the intention of enhancing simulation performance at the higher abstraction levels.

As with the 4-state register type `integer`, SystemVerilog defines a number of predefined `bit` types of various widths – `shortint` is 16 bits, `int` is 32 bits, and `longint` is 64 bits, on all platforms.

Objects of the `bit` type are by default `unsigned`, and objects of the predefined types are, by default, `signed`.

SystemVerilog also defines a 32-bit floating-point type, `shortreal`, to complement the 64-bit Verilog `real` type.

What Is a 'logic'?



The keyword `logic` defines that the variable or net is a 4-state data type.

- The Verilog `reg` data type is potentially confusing:
 - It can synthesize to a net or register depending on the usage.
- SystemVerilog redefines register data types as variable (`var`).
- SystemVerilog also defines the 4-state type as `logic`.
 - `var logic` (variable of `logic` type) replaces `reg`.
 - The `var` keyword is optional.

Verilog

```
reg clock;
reg [15:0] b_reg;
reg [7:0] mem8x32 [0:31];
```

SystemVerilog

```
logic clock;
logic [15:0] b_reg;
logic [7:0] mem8x32 [0:31];
```



SystemVerilog changes the terminology for data types, but in a way that is backward compatible with Verilog.

Verilog has two forms of data type: net (`wire`) and register (`reg`). However, the term *register* is misleading. A synthesis tool may infer either combinational or sequential logic for a Verilog register type, depending on how it is used. Therefore, SystemVerilog uses the term `variable` (or `var`) to refer to Verilog register types.

SystemVerilog also defines the name `logic` for the basic 4-state Verilog type (0, 1, X, Z).

Therefore, the Verilog declaration `reg clk;` becomes `var logic clk;` in SystemVerilog. The default SystemVerilog data type for `logic` is `variable`, making the `var` keyword optional. Therefore the SystemVerilog form becomes `logic clk;` So `logic` becomes a direct replacement for `reg`.

You can also specify the `logic` type for net data types like `wire`. So Verilog `wire clk;` becomes `wire logic clk;` in SystemVerilog. However, SystemVerilog assumes that all net data types are `logic`, making the `logic` keyword optional. Therefore, the SystemVerilog form becomes `wire clk;`

Example: Usage of 2-State and 4-State Types

```
bit [7:0] twostate;
logic [7:0] fourstate;
...
twostate <= fourstate;
```

- 2-state variables initialize to 0
 - Hides failure to initialize design

- 4-state variables initialize to X
 - Exposes failure to initialize design

4-state to 2-state

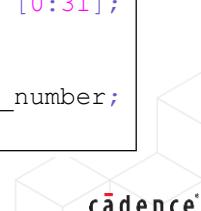
0	0
1	1
Z	0
X	0



Avoid mixing 2-state
and 4-state types

Usage of 2-state
types in TestBench

```
bit [7:0] test_number;
bit [7:0] mem8x32 [0:31];
int i;
...
mem8x32 [0] = test_number;
mem8x32 [1] = i;
```



Although 2-state types are more efficient for simulation, they have some issues for RTL design.

Remember that the x and z values in 4-state logic are used for more than modeling unknown and tristate values. X is used for un-initialized logic (not yet assigned) and z for un-driven logic (declared but never assigned). Hence 4-state types always initialize to x at the start of the simulation, allowing the detection of un-initialized code.

2-state types initialize to 0 at the start of the simulation; hence un-initialized code cannot be so easily detected.

SystemVerilog allows 2- and 4-state types to be easily mixed, although this can be dangerous. If a 4-state type is assigned to a 2-state type, any x or z value is converted to 0. This can hide un-initialized or un-driven values. The conversion occurs with no warning or error, and so mixing 4-state and 2-state types is generally not a good idea.

Due to the initialization and conversion issues with 2-state types, they are rarely used in RTL code. However, they can be useful for testbench code and for the definition of the clock and reset variables which should not need x or z values.



Quick Reference Guide: Verilog Data Type Rules

- Verilog has strict data type rules:
 - **Variables** (registers) are assigned values in procedural blocks.
 - **Nets** are driven by continuous assignments, module inputs, module instance outputs, or primitive instances.
- These lead to the following connectivity characteristics:
 - Module inputs are always **nets**.
 - Module outputs are **variables** if driven by a procedural block, or **nets** in all other cases.
 - Connections to the input ports of a module instance are **variables if driven by a procedural block, or nets in all other cases**.
 - Connections to the output ports of a module instance are always **nets**.
 - Connections to bidirectional **iout** ports are always **nets**.

21 © Cadence Design Systems, Inc. All rights reserved.



Remember the Verilog rules:

- You assign to register types (`integer`, `real`, `reg`, `time`) only in procedural blocks.
- You drive net types by continuous assignments or from outputs of modules or primitives.

This means that module input ports must connect internally to nets, module output ports must connect externally to nets, and module inout ports must connect both internally and externally to nets. Only module output ports can connect internally to variables.

The next page shows an example using the Verilog data type rules.

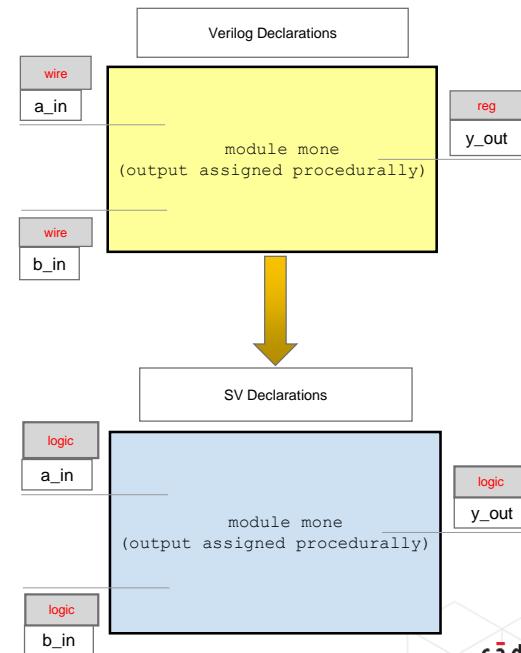
Relaxation of Datatype Rules



IEEE 1800-2012 6.3.1

SystemVerilog relaxes the rules for a **variable** by:

- Assigning a SystemVerilog **variable**:
 - In any number of `initial` or `always` blocks:
 - As in Verilog currently
 - From a single continuous assignment
 - From a single module `out` port
 - From a single primitive output
- Thus, you can declare most design signals to be **variable**.
 - As the `var` keyword is optional, you can simply declare most signals as type `logic`:
 - `logic my_var;`



22 © Cadence Design Systems, Inc. All rights reserved.

cadence®

To ease confusion about when to use a Verilog register type and when to use a net type, SystemVerilog relaxes the data type rules. A SystemVerilog variable can be driven from a continuous assignment or a module or primitive output, as long as there is only **one** driver of that form. Therefore, in most situations, you can simply use the `logic` keyword instead of `reg` or `wire`.

The next page contains an example using the relaxed SystemVerilog data type rules.

Verilog Data Type Rules: Example

```
module mone (output reg y_out,
             input wire a_in, b_in);
  always @(a_in or b_in)
    y_out = a_in && b_in;
endmodule
```

y_out is reg in mone – assigned in a procedural block

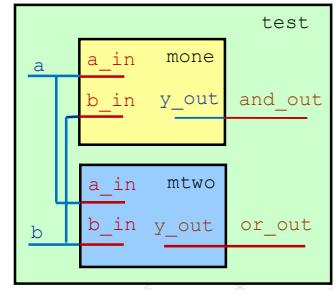
```
module mtwo (output wire y_out,
             input wire a_in, b_in);
  assign y_out = a_in || b_in;
endmodule
```

y_out is wire in mtwo – assigned by a continuous assignment

```
module test;
  reg a, b;
  wire and_out, or_out;
  mone u1 (and_out, a, b);
  mtwo u2 (or_out, a, b);
  initial begin
    a = 0;
    b = 0;
    ...
  end
endmodule
```

and_out, or_out are wire in module test – instance outputs

a, b are reg in module test – assigned in an initial procedural block



23 © Cadence Design Systems, Inc. All rights reserved.

Here, we are using fully defined Verilog-2001 ANSI C syntax for module port declarations for clarity.

In both modules `mone` and `mtwo`, the input ports `a_in` and `b_in` must be declared as net data types.

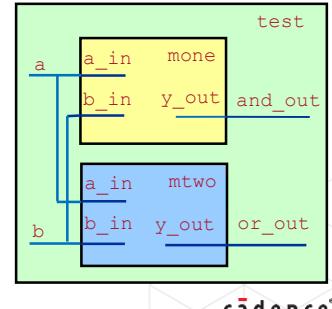
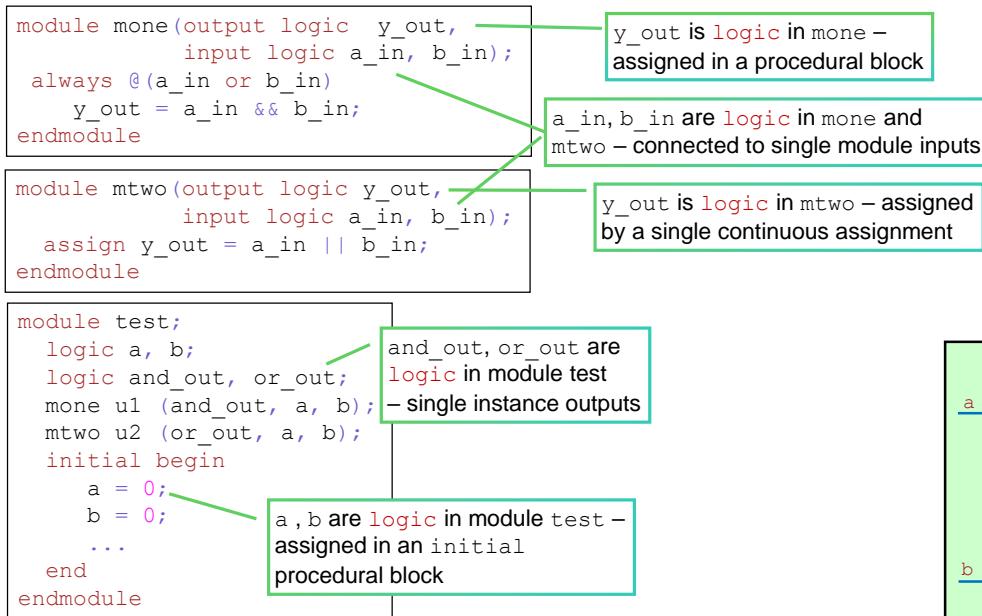
In module `mone`, the output `y_out` must be declared as a register data type as it is assigned from the `always` procedural block.

In module `mtwo`, the output `y_out` must be declared as a net data type as it is driven from the continuous `assign` statement.

In module `test`, `a` and `b` must be declared as register data types as they are assigned from the initial procedural block, but `and_out` and `or_out` must be declared as net data types as they are driven from the instance output ports.

Therefore, a connection like `a -> a_in`, or `y_out -> and_out` actually changes data type as it crosses the module boundary.

Relaxed Data Type Rules: Example



24 © Cadence Design Systems, Inc. All rights reserved.

Using the relaxed data type rules of SystemVerilog, our example becomes easier to read and write.

In both modules `mone` and `mtwo`, the input ports `a_in` and `b_in` can now be declared as register data types (`logic`) as they are both driven from single input ports of the module.

In module `mtwo`, the output `y_out` can now be declared as a register data type (`logic`) as it is driven from a single continuous assign statement.

In module `test`, `and_out` and `or_out` can now be declared as register data types (`logic`) as they are driven from single output ports of the module instances.

We can simply use `logic` everywhere in this example.

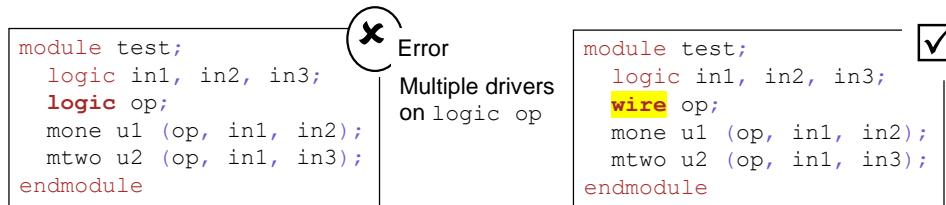
Therefore, a connection like `a -> a_in` or `y_out -> and_out` no longer changes data type as it crosses the module boundary.

How to Overcome Variable Assignment Restrictions



Although SystemVerilog relaxes the rules on data type declaration, there are still some restrictions on assignments to variables. These restrictions aim to prevent multiple drivers on a single variable.

- You cannot combine procedural assignments with continuous assignments or module output drivers on the same variable.
- You cannot have multiple continuous assignments or multiple output ports drive the same variable.
- Only net types can have multiple drivers.



Although SystemVerilog relaxes the rules on data type declaration, there are still some restrictions on assignment to variables. These restrictions aim to prevent multiple drivers on a single variable.

If a variable is assigned from a procedural block, then it cannot also be driven from a continuous assign statement or from a module output port.

A variable cannot be driven by multiple assign statements or multiple module output ports or by a combination of assign and port drivers.

Only net data types (like `wire`) are allowed to have multiple drivers.

The intention is that signals with single drivers are declared as variables, and net types are reserved solely for connections with multiple drivers only.

However, you still have the problem of driving a variable from multiple procedural blocks. SystemVerilog has a different solution for this problem.

Assignments to a Variable from Multiple Procedures



Verilog allows assignments to a variable from multiple procedures. How do you avoid this in SystemVerilog?

- Special procedural blocks (`always_ff`, `always_comb`, `always_latch`) allow only a single driver to variables.
 - See the Procedures and Procedural Statements module.

```
logic op;  
  
always @(sel, a, b)  
  if (sel)  
    op = a;  
  else  
    op = b;  
  
always @(sel2, c, d)  
  if (sel2)  
    op = c;  
  else  
    op = d;
```



```
logic op;  
  
always_comb  
  if (sel)  
    op = a;  
  else  
    op = b;  
  
always_comb  
  if (sel2)  
    op = c;  
  else  
    op = d;
```



Error

always_comb
allows only a
single driver on
logic op.

26 © Cadence Design Systems, Inc. All rights reserved.



SystemVerilog has a set of new procedural blocks for RTL code. These allow the designer to clearly identify how they intend the procedural block to be synthesized. For example, `always_comb` is an `always` block that should synthesize to combinational logic.

Another feature of these new procedural blocks is that they enforce a “single driver per variable” policy. If a variable is assigned from one of these procedural blocks, then it is a compilation error to drive the same variable from any other procedural block, or `assign` statement or module output.

The Procedures and Procedural Statements module describes these new procedural blocks in more detail.

What Is an Unsized Literal?



An Unsized literal is a number (integer or real) with a base specifier but no size specification.

- Size and value need not fit the target.
 - Value is padded or truncated to fit the target.
- Verilog-1995 extends to 32 bits:
 - With 0 if leftmost bit is 0 or 1.
 - With leftmost bit if Z or X.
- Verilog-2001 extends to full size of expression.
- New SystemVerilog single-bit literal syntax:
 - Fills whole target with that single value.
 - Avoids issues with incorrect literal sizing.
 - Allows entire vector to be set to 1.

// Sized literal
// <size>'<base><value>

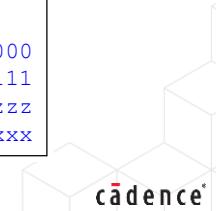
logic [5:0] databus;

databus = 6'b0; // 000000
databus = 6'b1; // 000001
databus = 6'bz; // zzzzzz
databus = 6'bx; // xxxxxx
databus = 4'bx; // 00xxxx

SystemVerilog

// Unsized literal
// '<value>
logic [5:0] databus;

databus = '0; // 000000
databus = '1; // 111111
databus = 'z; // zzzzzz
databus = 'x; // xxxxxx



27 © Cadence Design Systems, Inc. All rights reserved.

A Verilog literal has three parts: a size (default value 32 in Verilog-1995), a base (default decimal), and a value (required).

Verilog-1995 extends an unsized literal to 32 bits. If the leftmost bit is the high impedance value or the unknown value, it extends to 32 bits with that value. It extends beyond 32 bits with zeros (with the normal padding rules for vector assignment where the target is smaller than the source).

Verilog-2001 extends the leftmost high impedance value or unknown value to the size of the enclosing expression.

SystemVerilog has a new syntax that allows single-bit literals that are both unsized and unbased.

When you assign an unsized literal, SystemVerilog fills all bit positions of the target with the specified single-bit 4-state value.

This makes it very easy to set a logic vector to, for example, all ones, all high impedance, or all unknowns.

What Are Time Literals?



Time Literals are numbers written in integer or fixed-point format, followed without a space by a time unit (fs ps ns us ms s).

- SystemVerilog provides a time literal construct that:
 - Explicitly specifies time values.
 - Allows more control over time literals.

- Format:
 - Integer or fixed-point value.
 - Immediately followed by time unit:
 - With no space between the value and unit.
 - Having units of (fs ps ns us ms s).
 - Scaled to the current time unit.
 - Rounded to the current precision.
 - Also having a 1step delay value:
 - One simulation precision unit.

42ns
3.14ps
1step

Only 1step is allowed
Never 2step, etc.



You write a time literal as an integer or fixed-point number, followed immediately by a time unit (fs ps ns us ms s) without an intervening space character.

The 1step delay refers to the simulation step, that is, the simulation precision.

Time literals allow delay values to be explicitly defined and so isolated from the current time unit, traditionally specified with a `timescale compiler directive. However, SystemVerilog still rounds the time literal to the current time precision. This implies that you must have a current time unit and precision specified for the scope where the time literal appears.

Example: Time Units in Assignments

```
`timescale 1ns / 100ps      // time unit 1ns, precision 100ps
module testmodule;
  logic avar, sel, bvar, cvar, clk;

initial
begin
  #20ns sel = 0;           // blocking assign
  #5.18ns sel = 1;         // here rounded to 5.2ns
  bvar = #1step cvar;      // blocking assign using 1step
end

always @(posedge clk)
  #5.1ns avar <= avar + 1; // nonblocking assign
endmodule
```

You can use a time literal or `1step` anywhere you would use a delay

In this example, `#1step` advances simulation for 100ps



You can use a time literal anywhere you would use a unitless integer or real number to represent time.

This example uses time literals to specify procedural and intra-assignment delays.

Although the time literal isolates a delay value from the current time unit, the delay is still scaled by the current time precision. In the example above, the `#5.18ns` delay is scaled to `#5.2ns` as the precision is 100ps (0.1ns).

timeunit and timeprecision



IEEE 1800-2012 22.7

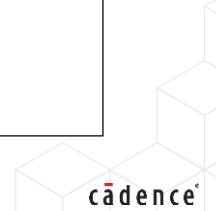
- Equivalent to `timescale, but without file order compilation dependency.
- timeunit and timeprecision are:
 - Declarations, not directives.
 - Placed inside the module.
 - Must be the first statements of the module.
 - Only visible in the design element where they are declared and any nested units.

```
module testbench;
  timeunit 1ns;
  timeprecision 100ps;
  logic status;
  initial begin
    #10ns status = 1'b1;
    #10 status = 1'b0;
  end
endmodule
```

```
module testbench2;
  timeunit 1ns/100ps;
  logic status;

  initial begin
    #10ns status = 1'b1;
    #10 status = 1'b0;
  end
endmodule
```

30 © Cadence Design Systems, Inc. All rights reserved.



The timeunit and timeprecision declarations are equivalent to the timescale compiler directive (`timescale), but without the file-order dependency of compiler directives. timeunit and timeprecision must be the first declarations in the module or design element. The declarations are visible only within that unit declaration and within any further nested declarations.

Syntax: **timeunit <literal><units>** ;
 timeprecision <literal><units> ;

Where:

literal ::= 1, 10, 100

units ::= s ms us ns ps fs step

- step advances time by the simulation time precision.

The current time unit is defined using the following highest to lowest precedence:

- 1st – The timeunit of the current interface, module, package or program definition.
- 2nd – The timeunit inherited from the enclosing interface or module definition.
- 3rd – The timescale (`timescale) directive of the current compilation unit.
- 4th – The timeunit declaration of the current compilation unit.
- 5th – The implementation-specific default time unit.

Quiz



Which data type can be used for the majority of SystemVerilog variables?

logic



In which design situations would you need to use nets?

Where there is more than one driver, for example tristates and bidirectionals.



Which logic set is intended for modeling at higher-abstraction levels?

2-state logic



What is the issue with using 2-state data types with RTL code?

2-state logic initializes to 0 and so hides initialization problems in RTL code. Also assigning 4-state to 2-state variables converts X and Z values to 0.



Which declarations allow more control over timing?

timeunit and timeprecision



This page does not contain notes.



Procedural Statements and Procedural Blocks

Module **4**

Revision **1.0**
Version **21.10**

Estimated time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Module Objectives

In this module, you

- Use new SystemVerilog procedural statements and enhancements to existing constructs
- List enhancements for addressing common Verilog synthesis issues
- Use synthesizable procedural blocks



This page does not contain notes.

Repeated Block Identifier at Block End



For any named block, SystemVerilog allows you to repeat the name after the block end, join, or join variant keyword.

Verilog-2001 allows begin...end and other code blocks to be named.

SystemVerilog allows a matching name to be specified with the block end.

It is allowed for the following code blocks:

- module...endmodule
- begin...end
- task...endtask
- function...endfunction
- fork...join (all forms)
- primitive...endprimitive
- interface...endinterface

You can label *any* statement.

Verilog-1995

```
module mytest;
initial
begin : blocka
...
begin : blockb
...
end
end
endmodule
```

SystemVerilog

```
module mytest;
initial
begin : blocka
...
begin : blockb
...
end : blockb
end : blocka
endmodule : mytest
```



This page does not contain notes.

Block Item Declarations in Unnamed Blocks



IEEE 1800-2012 A.2.8

In Verilog-2001, only named blocks can declare local variables.

In SystemVerilog, unnamed blocks can declare local variables.

- The variable is visible in the block where it is declared and in any nested blocks.
- Hierarchical references to the variable cannot be used.

```
initial begin          // unnamed block
    integer i;           // local declaration in unnamed block
    i = 0;
    while (i <= 10) begin // unnamed block
        $display("i:%d", i);
        if (i < 5) begin   // unnamed block
            $display("Number is:%d", i);
        end
        i = i + 1;
    end
end
```

35 © Cadence Design Systems, Inc. All rights reserved.



Verilog-1995 did not allow variables to be declared in procedural blocks or local scopes within a block.

Verilog-2001 allowed local variables to be declared within blocks, but only if the block was explicitly named.

SystemVerilog allows local variables to be declared in unnamed blocks. These variables are visible in the unnamed block and any blocks nested below it, but as the block has no name, the variables are not accessible through hierarchical references.

Variable Declarations in `for` Loop Statements



IEEE 1800-2012 12.7.1

Verilog

```
// loop variable declared outside loop
integer i;
initial begin
    for (i=0; i<10; i=i+1)
        ...
initial begin
    for (i=0; i<8; i=i+1)
        ...
int: Good use
for 2-state types
```

You can declare a `for` loop variable within the `for` statement.

- Variables are visible only in the loop.
- The same identifier name can be safely used in multiple (un-nested) loops.
- Good use of 2-state data types as loop variables.

SystemVerilog

```
initial begin
    // loop variable declared in loop
    for (int i=0; i<10; i=i+1)
        ...
initial begin
    for (int i=0; i<8; i=i+1)
        ...
    ...
```

Variable 'i' visible
to for loop onlyTwo loops, two
independent variables

36 © Cadence Design Systems, Inc. All rights reserved.

One benefit of allowing variables in unnamed blocks is that you can now declare a `for` loop variable within the `for` statement, removing the need to separately declare the variable outside the loop. This has the added benefit of avoiding issues where a single loop variable is accidentally or deliberately used in multiple, overlapping loops.

In fact, you can declare multiple variables in the loop, separately initialize each variable, and separately reassign each variable. Thus, SystemVerilog `for` loops can be a function of more than one loop variable.

foreach Loop



IEEE 1800-2012 12.7.3

This loop iterates over all the elements of an array.

Loop variable characteristics:

- Does not have to be declared.
- Are read only.
- Only visible inside loop.

Use multiple loop variables for multidimensional arrays.

- Equivalent to nested loops.

Useful for initializing and array processing.

Equivalent SV code

```
for (int i=7; i>=0; i=i-1)
    intarr[i] = 7-i;
```

```
int intarr [7:0];
...
foreach (intarr [i])
    intarr[i] = 7-i;
```

Iterates left bound to right bound

Equivalent SV code

```
int arr2d [7:0] [2:0];
...
foreach (arr2d [k, l])
    arr2d[k][l] = k*l;
```

```
for (int k=7; k>=0; k=k-1)
    for (int l=2; l>=0; l=l-1)
        arr2d[k][l] = k*l;
```

Two loop variables for a 2D array creates nested loops

37 © Cadence Design Systems, Inc. All rights reserved.



The SystemVerilog foreach loop iterates over all the elements of an array. It is equivalent to a for loop, which iterates over the full width of the array from left bound to right bound. This construct is useful for simple loop-based array initialization or processing.

The foreach loop variable is automatically declared (and typed) as part of the foreach statement. As a local variable, it is visible only within the foreach statement and any nested blocks. In a change to normal Verilog behavior, this variable cannot be written but only read.

If the array is multidimensional, you can declare a loop variable for each dimension. This creates nested loops. Loop variables are mapped to dimensions in order of cardinality (see the *IEEE1800 Language Reference Manual*). In the example above, k iterates from 7 to 0, and l from 2 to 0. Also, higher cardinality dimensions change more rapidly, so in the example above, k is set at 7 while l iterates from 2 to 0, then k decrements to 6 while l changes, etc.

Syntax:

```
foreach ( array_identifier [ loop_variables ] )
    statement(s)
```

do ... while Loop



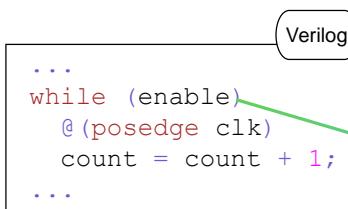
IEEE 1800-2012 12.7.4

- The `while` loop executes a group of statements until expression becomes false.
- expression is checked at the beginning.

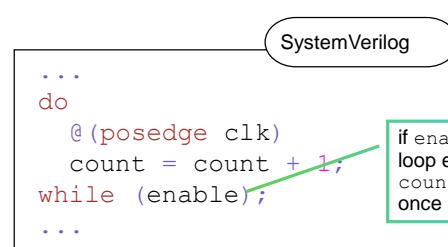


IEEE 1800-2012 12.7.5

- The expression is checked after statements execute.
- The statement block executes at least once.
- This makes certain loop functions easier to create.



if enable false on loop entry;
count not incremented



if enable false on loop entry;
count incremented once

38 © Cadence Design Systems, Inc. All rights reserved.



The Verilog `while` loop executes statements in the loop while the condition is true. The condition is checked before the loop is entered, and if the condition is false at this point, no loop statements are executed.

The SystemVerilog `do...while` loop executes the loop statements once before the condition is tested. Therefore even if the condition is false, the loop statements are executed once. This form of the `while` loop makes certain functionality easier to create.

Syntax:

```
do  
  statement_or_null  
  while ( expression ) ;
```

Jump Statements **break** and **continue**



IEEE 1800-2012 12.8

SystemVerilog adds the **break** and **continue** keywords to control execution of loop statements.

- **break**
 - Terminates the execution of loop immediately.
 - Usually under conditional control.
- **continue**
 - Jumps to the next iteration of a loop.
 - Usually under conditional control.
- Also used in **for**, **while** and **do-while** loops.

```
repeat (8) begin
    data = {data[6:0], data[7]};
    if (data[7])
        break;
end
...
```

```
foreach (data [i]) begin
    if (data[i])
        continue;
    count = count + 1;
end
```

39 © Cadence Design Systems, Inc. All rights reserved.



You can use **break** and **continue** statements only within loop statements (**do**, **for**, **foreach**, **repeat**, **while**).

break terminates the current loop, i.e., it forces a jump to the end of the loop. The **break** is usually conditional. This **break** example rotates data to the left until the rightmost bit is 1.

continue starts the next iteration of the loop, i.e., it forces a jump to the beginning of the loop. Again, **continue** is usually conditional. This **continue** example counts the number of zeros in the data vector. The **foreach** loop iterates through the data bits and jumps over the increment statement if the current data bit is 1.

Use of these statements may make your code more readable than using branching statements or the Verilog **disable** statement.



Quick Reference Guide: Verilog case Statement

Synthesis Pragmas	Explanation	Example
parallel_case	<ul style="list-style-type: none"> parallel_case attribute directs the synthesis tool to remove inferred priority logic. Commonly used to remove priority encoders from gate-level implementation. Generally discouraged. 	<pre>(* synthesis parallel_case *) case (parc) 0: op = a; 1,2: op = b; 3: op = c; endcase</pre>
full_case	<ul style="list-style-type: none"> full_case informs the synthesis tool to treat outputs as don't care assignments for all unspecified case choices. It doesn't work since it looks at only case item expressions and not their associated statements. Adding the default statement nullifies its usage. Generally discouraged. 	<pre>(* synthesis parallel_case, full_case *) case (fulc) 0,1: op = a; 2: op = b; default: op = c; endcase</pre>

40 © Cadence Design Systems, Inc. All rights reserved.



Verilog case statement branches are prioritized. The simulator checks the branches in the order in which they appear; branches can overlap and there need not be a branch for every possible value of the case expression. The synthesis tool must initially produce hardware that represents this priority structure, though it can later optimize out the priority structure if it is not needed. As the designer, you may know that in real operation, the case match items are mutually exclusive. You can use the parallel_case attribute to direct the logic synthesis tool to not build the priority structure.

As the designer, you may know that in real operation, certain case expression values cannot occur. If you do not specify match items for those values and do not specify default assignments, the synthesis tool will, for combinational logic, infer a latch to hold the existing result values for those unmatched case expression values. You can use the full_case attribute to direct the logic synthesis tool to produce binary result values for those unmatched case expression values, rather than to infer a latch. Use of the full_case attribute should never be necessary, as you can always simply add a default branch.

If incorrectly applied, these directives can create incorrect hardware. As the directives are ignored by simulation tools, this incorrect behavior may only be detected (if at all) in a post-synthesis netlist.

SystemVerilog priority case



IEEE 1800-2012 12.5.3

- Modifier to the `case` statement
- **Synthesis:** Equivalent to the `full_case` attribute
- **Simulation:** Compile-time/Run-time violation report (probably warning) if no case item expression {can / does} match the case expression
 - Overlapping branches are permitted
- Can also apply to `casex`, `casez`

```
priority case (fullc)
  0:      op = a;
  0,1,2 : op = b;
  3:      op = c;
endcase
```



```
priority case (1'b1)
  en_a : op = a;
  en_b : op = b;
  en_c : op = c;
endcase
```

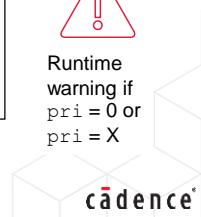
Runtime warning if one enable not active

```
priority casez (pri)
  3'b1???: op = a;
  3'b01?: op = b;
  3'b001: op = c;
endcase
```



Runtime warning if pri = 0 or pri = X

41 © Cadence Design Systems, Inc. All rights reserved.



The `priority case` modifier replaces and enhances the `full_case` attribute. SystemVerilog performs runtime checks to verify that for every execution of the case statement, a matching branch is found. Hence, the full case behavior of the case statement can be tested in simulation.

Note: If simulation is insufficient and a legal value that does not have a matching branch is never applied to the priority case, then a runtime warning is not received. For example, in the `priority casez` example, if `pri = 0` was a legal value, but never applied in simulation, a runtime warning would not be seen and the issue would not be detected.

SystemVerilog unique case



IEEE 1800-2012 12.5.3

- Modifier to the `case` statement.
- **Synthesis:** Equivalent to the `full_case` and `parallel_case` attributes for synthesis.
- **Simulation:** Compile-time/Run-time violation report (probably warning) if no case item expression {can / does} match the case expression or multiple case item expressions {can / do} match the case expression.
- Defines `case` branches as complete and mutually exclusive.
 - One branch must be executed.
 - Runtime warning if no branch found for a case value.
 - No overlapping allowed.
 - Runtime warning if multiple branches found for a case value.
- Can also apply to `casex`, `casez`.
 - Beware: more likely to be overlapping.

```
unique case (fullc)
  0:      op = a;
  0,1,2 : op = b;
  3:      op = c;
endcase
```



Runtime warning
when fullc=0

```
unique case (1'b1)
  en_a : op = a;
  en_b : op = b;
  en_c : op = c;
endcase
```



Runtime warning
if multiple enables active

```
unique casez (pri)
  3'b1???: op = a;
  3'b?1?: op = b;
  3'b??1: op = c;
endcase
```



Runtime warning
if more than one bit=1

42 © Cadence Design Systems, Inc. All rights reserved.



The `unique case` modifier replaces and enhances the `parallel_case` attribute. SystemVerilog performs runtime checks to verify that for every entry into this statement, one and only one matching branch is found.

Note: The simulation may report that a case statement is not unique (for example, when `fullc` has the value `3'bxxx`) even though for synthesis purposes the case is full and parallel.

SystemVerilog priority if



IEEE 1800-2012 12.4.2

Modifiers can be applied to `if` statements with the same effects.

`priority if` implies full logic:

- Branch conditions are checked sequentially.
- Compile-time/Run-time violation report (probably warning) if no branch {can be / is} taken. First match executes:
 - Use an unconditional `else?`

```
priority if (en_a)
    op = a;
else if (en_b)
    op = b;
else if (en_c)
    op = c;
```



Runtime warning
if one enable
not active

43 © Cadence Design Systems, Inc. All rights reserved.



The `priority` and `unique` modifiers can also be applied to `if` statements. An `if` statement can be made complete by appending an unqualified `else` with a null statement (`else ;`). A null statement is simply a semicolon.

It is important to understand that SystemVerilog treats a `unique if` very much like a `unique case` in that it evaluates each conditional expression independently of the other conditional expressions.

SystemVerilog unique if



IEEE 1800-2012 12.4.2

Unique if implies parallel logic:

- Branch conditions are tested simultaneously (priority structure is ignored). Exactly one branch must match.
- Compile-time/Run-time violation report (probably warning) if no branch {can be / is} taken.
- Compile-time/Run-time violation report (probably warning) if multiple branches can be taken.

```
unique if (ctrl >= 3'b100)
    op = a;
else if (ctrl <= 3'b100)
    op = b;
else
    op = c;
```



Runtime warning
if ctrl == 3'b100

This page does not contain notes.

Conditional Event Control with `iff` Qualifier



IEEE 1800-2012 A.6.5

The `iff` keyword qualifies a procedural event control.

The event expression triggers only if the `iff` condition is true.

- For example, the block executes when `avec` changes and `enable` is 1.
- Block not triggered when `enable` changes.

`iff` has precedence over `or`:

- Can add parenthesis for clarity.

Limited use in RTL code.

- Latches and gated clocks.

More useful for embedded events in verification stimulus code.

```
always @(avec iff enable == 1)
y <= avec;
```

Creates Latch in Synthesis

```
always @(avec)
if (enable)
y <= avec;
```

Hardware Equivalent

Typical RTL Usage

```
always @(posedge clk iff (gate == 1)
      or posedge rst)
  if (rst)
    data_out <= 8'h00;
  else
    data_out <= bvec;
```

Typical TB Usage

```
...
foreach (payload [i])
  // send payload when not suspended
  @ (negedge clk iff (!suspend))
    data_out <= payload[i];
...

```



45 © Cadence Design Systems, Inc. All rights reserved.

You can add an “if and only if” (`iff`) qualifier expression to any event, net, or variable in an event expression. The event guard applies only to the immediately preceding identifier.

In the top example, the `enable` signal guards the `avec` signal. Transitions of `enable` have no effect. Transitions of `avec` trigger block execution only when `enable` is 1. The expression is evaluated when `avec` changes, and not when `enable` changes.

Note that `iff` has precedence over `or`. This can be made clearer by the use of parentheses.

Use for `iff` in RTL is limited to modeling latches and gated clocks.

`iff` is more useful in verification code, particularly in embedded event expressions for stimulus code. In the last example, data can only be driven into the design on the negative edge of `clk` when the DUT output `suspend` signal is low. Using the `iff` guard makes the code much simpler.

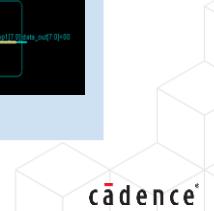


Quick Reference Guide:

Modeling Hardware Elements for Synthesis Using Verilog

Hardware Element	Explanation	Example	Schematic Illustration
Combinational	Combinational block can be modeled using an always statement.	<pre>always @(a or b or sel) if (sel == 1) op3 = a; else op3 = b;</pre>	
Latch	A Latch can be modeled using an always statement.	<pre>always @(sel or a) if (sel) op2 <= a;</pre>	
Register	Edge sensitive storage device can be modeled using an always statement.	<pre>always @ (posedge clk or posedge rst) if (rst) op1 <= 8'h00; else op1 <= data_in;</pre> <p>1. Synthesizability is controlled by sensitivity list and coding style. 2. Simulator has no idea what user intends. </p>	

46 © Cadence Design Systems, Inc. All rights reserved.



A Verilog `always` block can synthesize to combinational, latched, or sequential logic, depending upon the sensitivity list and your coding style. The simulator does not know what you intend, so it cannot verify that your block matches your intentions.

SystemVerilog adds implementation-specific procedural blocks (`always_comb`, `always_latch`, `always_ff`). These blocks reduce design ambiguity by clearly indicating the hardware intent for a procedural block. Simulation, formal checking, linting, synthesis, equivalence checking and other downstream tools can have a consistent specification.

Combinational Blocks with `always_comb`



IEEE 1800-2012 9.2.2.2

SystemVerilog adds a specialized procedural block for modeling combinational logic:

- Implied, complete sensitivity list.
- Any variable assigned in an `always_comb` cannot be assigned by another procedure.
- Cannot contain further blocking timing or event control.
- Automatically executed once at time 0 without waiting for an event.
 - After all `initial` and `always` blocks have executed.
 - Ensures outputs are consistent with inputs.
- Tools may issue warnings if the block does not infer combinational logic.

```
always_comb
  if (sel == 1)
    op = a;
  else
    op = b;
```

 Error

```
logic op;

always_comb
  if (sel)
    op = a;
  else
    op = b;

always_comb
  if (sel2)
    op = c;
  else
    op = d;
```

47 © Cadence Design Systems, Inc. All rights reserved.



The `always` combinational (`always_comb`) procedural block provides special functionality for combinational logic:

- It infers a sensitivity list that includes every variable read by the procedural block.
- It prohibits its assigned variables from being assigned in any other procedural block. (With these processes, it is illegal for a variable to be written to by more than one process, even when these processes are in different modules, interfaces or test programs)
- It cannot contain any embedded timing or event control.
- It is initially run once at time zero after all `initial` and `always` blocks have run to suspension.
This is to ensure that the outputs of the block are consistent with any changes to the inputs.

Normally, an `always` block will be executed only at time 0 if there is an event on a signal in its sensitivity list. In addition, the execution order of `always` and `initial` blocks is indeterminate. A testbench `initial` block may apply an initial value to a design input that does not trigger an event (e.g., assignment of 0 to a 2-state type). Therefore, the outputs of a combinational block may not be consistent with the input values at time 0. Use of an `always_comb` fixes this initialization inconsistency by automatically executing the `always_comb` after `initial` and `always` blocks at time 0.

Software tools can perform additional checks to warn whether behavior within an `always_comb` procedural block does not represent combinational logic, e.g., whether it infers latches.

Comparing Constructs `always @*` and `always_comb`



`always @*`

- Verilog-2001 event control feature



`always_comb`

- SystemVerilog procedural block

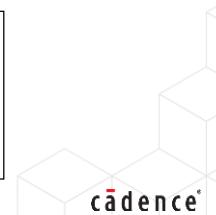
- Can include timing and additional event controls.
- Can assign to variables which are assigned elsewhere.
- Triggered at time 0 with other blocks only if event on sensitivity list.
- Only sensitive to changes in the arguments of a called function.

```
always @*
  if (sel == 1)
    op = a;
  else
    op = b;
```

- Cannot contain any timing or event controls.
- Cannot assign to variables which are assigned elsewhere.
- Automatically triggered at time 0 after `always` and `initial` blocks.
- Sensitive to changes in any input to a called function.

```
always_comb
  if (sel == 1)
    op = a;
  else
    op = b;
```

48 © Cadence Design Systems, Inc. All rights reserved.



Verilog-2001 added a wildcard sensitivity list for combinational `always` procedures. However, this is only an enhancement for the event expression; it does not prevent the `always` block from containing other event controls or timing, from driving variables that are assigned elsewhere nor trigger execution of the block at time 0.

Also, the wildcard sensitivity has a limitation. If the `always` block contains a function call, then only the arguments of the function are added to the sensitivity list. If the function reads a variable that is not passed via the argument list (i.e., reads the variable by side effects), then that variable is not included in the wildcard sensitivity list.

The SystemVerilog `always_comb` block includes the extra features which prevent additional event or delay expressions in the block; it prevents multiple drivers on assigned variables and triggers the execution of the block at the start of simulation. In addition, the complete sensitivity list of `always_comb` includes all variables read by a function call, including those read by side effects.

Therefore, in every way, the `always_comb` block is far better for combinational synthesis code than the `always` block.

Latch Blocks with `always_latch`



IEEE 1800-2012 9.2.2.3

SystemVerilog adds a specialized procedural block for modeling latched logic:

- Implied, complete sensitivity list.
- Variables assigned in an `always_latch` cannot be assigned by another procedure.
- Cannot contain further block timing or event control.
- Automatically executed once at time 0 without waiting for an event:
 - After all `initial` and `always` blocks have executed.
 - Ensuring outputs are consistent with inputs.
- Tools can issue warnings if the block does not infer latched logic.

```
always_latch  
if (gate == 1)  
    op <= a;
```

```
always_latch  
if (en1)  
    op <= c;  
  
always_latch  
if (en2)  
    op <= d;
```

Error



49 © Cadence Design Systems, Inc. All rights reserved.

The `always_latch` (`always_latch`) procedural block provides special functionality for latched logic, similar to an `always_comb` block.

- It infers a sensitivity list that includes every variable read by the procedural block.
- It prohibits its assigned variables from being assigned in any other procedural block.
- It cannot contain any embedded timing or event control.
- It is initially run once at time zero after all `initial` and `always` blocks have run to suspension.
This is to ensure that the outputs of the block are consistent with any changes to the inputs.

However, the best feature of the `always_latch` is that it clearly documents that a designer intends to create latched logic.

Software tools can perform additional checks to warn whether the behavior within an `always_latch` procedural block does not represent latched logic.

Register Blocks with always_ff



IEEE 1800-2012 9.2.2.4

SystemVerilog adds a specialized procedural block for modeling registered logic:

- Variables assigned in `always_ff` cannot be assigned by another procedure.
- Contains one and only one event control.
- Cannot contain any block timing.
- Tools may issue warnings if the block does not infer registered logic.

```
always_ff @ (posedge clk or posedge rst)
  if (rst)
    op <= 1'b1;
  else
    op <= ip;
```

50 © Cadence Design Systems, Inc. All rights reserved.



The `always` flip-flop (`always_ff`) procedural block requires one and only one event control preceding the block body.

- It prohibits its assigned variables from being assigned in any other procedural block.
- It cannot contain any additional embedded timing or event control.

Software tools can perform additional checks to warn whether the behavior within an `always_ff` procedural block does not represent sequential logic.

Module Summary

In this module, you learned how to

- Write block names that can also appear at the block end, which enhances readability
- Declare local variables in unnamed blocks that are especially useful as for and foreach loop variables
- Write the full_case and parallel_case attributes that can be replaced for proper case statement synthesis
- Set up runtime checks with the event control iff qualifier
- Analyze always_comb, always_latch, always_ff which can be used to prohibit multiple signal drivers, better initialization, and no embedded events or timing



This page does not contain notes.

Procedural Blocks: Quiz

What is wrong with these examples?
How could you correct them?

//ONE
logic [2:0] myvec, tmp;

always@ (myvec) begin
 tmp = myvec>>1;
 unique case (tmp)
 3'b000 : \$display("zero");
 3'b001 : \$display("one");
 3'b010 : \$display("two");
 endcase;
end

No case branch
for tmp=3'b011

//TWO
always_ff
begin
 #(period/2) clk <= 0;
 #(period/2) clk <= 1;
end

Block timing inside
an always_ff

//THREE
always @ (posedge clk iff rst == 0)
// synchronous reset
if (rst)
 data_out <= 8<92>h00;
else
 data_out <= data_in;

Procedure never
triggered when rst=1,
so never reset

//FOUR
always_comb
if (ctrl == 1)
 op <= a;

Does not infer
combinational logic

This page does not contain notes.

priority and unique: Quiz



```
module mod1 (input bit [2:0] data);
  always @(data)
    priority casez (data)
      3'b00?: $display("0 or 1");
      3'b0?: $display("2 or 3");
    endcase
endmodule
```

Runtime warnings when
data = 4 to 7
or any data bit is unknown



```
module mod2 (input [2:0] data);
  unique @(data)
    unique case (data)
      0,1: $display("0 or 1");
      2,3: $display("2 or 3");
      4,5: $display("4 or 5");
    endcase
endmodule
```

Runtime warnings when
data = 6 to 7 or any data bit
is unknown



```
module mod3 (input clk, rst, en,
              output reg [7:0] cnt);
  always @ (posedge clk or negedge rst)
    unique if (!rst)
      cnt <= 0;
    else if (en)
      cnt <= cnt + 1;
endmodule
```

Runtime warnings when:
• incomplete: (rst==1 and en==0)
• incomplete: (rst==x or en==x)
• overlap: (rst==0 and en==1)

Inappropriate use of priority and
unique can cause problems.



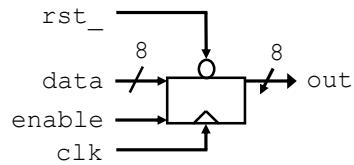
This page does not contain notes.



Labs

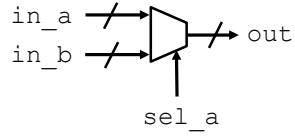
Lab 1 Modeling a Simple Register

- Model a simple register using logic data types, timeunit and timeprecision statements, and an always_ff procedural block.



Lab 2 Modeling a Simple Multiplexor (Optional)

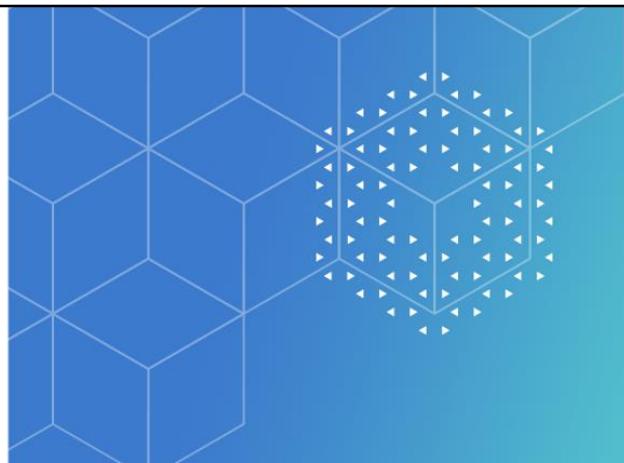
- Model a simple scalable multiplexor using an always_comb procedural block and the unique case construct.



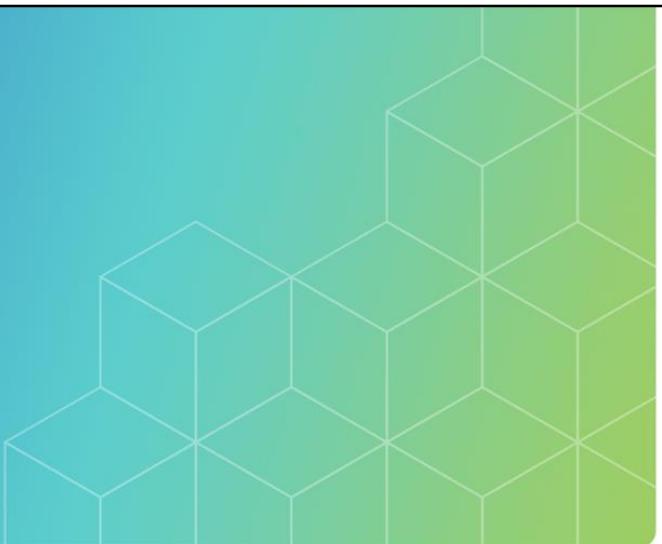
54 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



Operators



Module **5**

Revision **1.0**
Version **21.10**

Estimated time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Module Objectives

In this module, you

- Analyze new SystemVerilog assignment operators
- Use wildcard equality and inequality operators
- Examine the `inside` operator



This page does not contain notes.

What Are Assignment Operators?



Operators that join an operation along with a blocking assignment to the first operand of the operator.

```
initial begin
  a = 1;
  b = 2;
  a += b; // same as a = a + b, so, a = 3
  ...

```

Symbol	Usage	Meaning	Description
<code>+=</code>	<code>a += b</code>	<code>a = a + b</code>	add
<code>-=</code>	<code>a -= b</code>	<code>a = a - b</code>	subtract
<code>*=</code>	<code>a *= b</code>	<code>a = a * b</code>	multiply
<code>/=</code>	<code>a /= b</code>	<code>a = a / b</code>	divide
<code>%=</code>	<code>a %= b</code>	<code>a = a % b</code>	modulus
<code>&=</code>	<code>a &= b</code>	<code>a = a & b</code>	logical and
<code> =</code>	<code>a = b</code>	<code>a = a b</code>	logical or
<code>^=</code>	<code>a ^= b</code>	<code>a = a ^ b</code>	logical xor
<code><<=</code>	<code>a <<= b</code>	<code>a = a << b</code>	left shift logical
<code>>>=</code>	<code>a >>= b</code>	<code>a = a >> b</code>	right shift logical
<code>>>>=</code>	<code>a >>>= b</code>	<code>a = a >>> b</code>	right shift arithmetic
<code><<<=</code>	<code>a <<<= b</code>	<code>a = a <<< b</code>	left shift arithmetic

57 © Cadence Design Systems, Inc. All rights reserved.



Assignment operators are blocking assignments. Therefore, they are suitable for use only in RTL combinational logic, temporary variables in RTL sequential code, or testbench and stimulus.

Pre- and Post-Increment/Decrement Operators



IEEE 1800-2012 11.4.2

These combine a blocking assignment and an increment or decrement operator:

- Pre-form (`++a`, `--a`) adds or subtracts and then uses new value.
- Post-form (`a++`, `a--`) uses a value and then adds or subtracts.
- Operators can be used as:
 - A standalone statement.
 - Part of an expression.

Statement
`--total;`

As Statement
`for (int i=0; i < 7; i++)`
...

As Expression
`initial begin`
 `b = 1;`
 `a = b++; // post a=1, b=2`
 `a = ++b; // pre a=3, b=3`
 `a = b--; // post a=3, b=2`
 `a = --b; // pre a=1, b=1`
`end`



Pre- and post-increment and decrement operators are blocking assignments. Therefore, they are only suitable for use in RTL combinational logic, temporary variables in RTL sequential code, or testbench and stimulus.

Wildcard Equivalence Operator

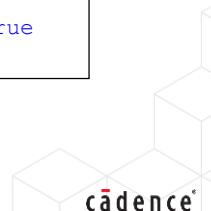


IEEE 1800-2012 11.4.6

- Allows bits to be defined as a “wildcard”:
 - Don’t-care – like casex
 - An X, Z or ? in the *right* operand matches any value in the left
 - Asymmetric – only right side can have wildcard bits
 - Also wild inequality ($!=?$)
- Remember that Verilog has two (in)equality operators:
 - Logical (in)equality ($==$)
 - Identity (in)equality ($===?$)

```
a = 4'b0101;
b = 4'b01XZ;

if (a == b)           // unknown
  ...
if (a === b)          // false
  ...
if (a ==? b)          // true
  ...
if (a !=? b)          // false
  ...
if (a ==? 4'b?1?1)    // true
  ...
...
```



The wildcard equality operator ($==?$) and the wildcard inequality operator ($!=?$) treat unknown (X) and high-impedance (Z) values in the given bit position of the right operand as “don’t-care” bits to ignore when doing the comparison. These operators return a 1-bit result that can be 0, 1 or unknown.

Remember that you can use the Z and question mark (?) characters interchangeably in a Verilog 4-state literal.

Set Membership Operator



IEEE 1800-2012 11.4.13

- inside does a comparison:
 - True if expression value is contained within a value list.
 - List values can be variables (including arrays), ranges.
 - List values can have wildcards.
 - List values can overlap.
- For nonintegral expressions, it uses the equality operator (==):
 - Result is 1'b0, 1'b1
- For integral expressions, it uses a wild equality operator (==?):
 - Result is 1'b0, 1'b1, 1'bX

```
if ( a inside {2'b01, 2'b10} )
...
// equivalent to:
if ((a==2'b01) || (a==2'b10))
```

```
int bar [1:0] = '{5,6};
if ( a inside {bar, [0:2]} )
...
// equivalent to:
if (a inside {bar[1], bar[0], 0, 1, 2})
```

```
if ( a inside {2'b0?} )
...
// equivalent to
if ( a inside {2'b00, 2'b01, 2'b0X, 2'b0Z})
```

60 © Cadence Design Systems, Inc. All rights reserved.



The set membership (inside) operator performs a comparison between a left-side expression and a right-side list of values. The right-side list can utilize wildcards, value ranges and the values can overlap.

Range values must be expressed in the form [lower_bound : higher_bound]. Either bound can be \$ indicating the minimum or maximum value for the expression.

For nonintegral expressions, SystemVerilog performs an equality match operation, which returns 1 or 0.

For integral expressions, SystemVerilog performs a wild equality match operation, which is an asymmetric match. Therefore, the result can be 0, 1 or unknown.

Syntax:

- expression inside { value_range { , value_range } }**
- value_range ::= expression | [expression : expression]**

Example: Using `inside` Operator with the `case` Statement

- You may want to execute statements when a variable or expression is within a range of values.
- `casez` can be used if range limits are on bit boundaries.
- `casez` is not useful if the range is not related to bit positions.
- For arbitrary range limits, the `inside` operator should be used.

```
bit[2:0] a;

casez (a)
  3'b00?:$display("0 or 1");
  3'b0???:$display("2 or 3");
endcase
...
```

Non bit boundary range

```
bit[3:0] a;

case(a) inside
  0,2    : $display("0 or 2");
  [3:5]   : $display("3 or 4 or 5");
  1,[6:7]: $display("1 or 6 or 7");
  default: $display("value>7");
endcase
...
```

Can mix range and discrete values



The set membership (`inside`) operator performs a comparison between a left-side expression and a right-side list of values. The right-side list can utilize wildcards, value ranges and the values can overlap.

Range values must be expressed in the form `[lower_bound : higher_bound]`. Either bound can be `$` indicating the minimum or maximum value for the expression.

For nonintegral expressions, SystemVerilog performs an equality match operation, which returns 1 or 0.

For integral expressions, SystemVerilog performs a wild equality match operation, which is an asymmetric match. Therefore, the result can be 0, 1 or unknown.

Syntax:

- `expression inside { value_range { , value_range } }`
- `value_range ::= expression | [expression : expression]`



Quick Reference Guide: Operator Precedence and Associativity

Operator	Associativity	Precedence
() [] :: .	Left	Highest
+ - ! ~ & ~& ~ ^ ~^ ~^ ~+ ~-	Right	
(unary)		
**	Left	
* / %	Left	
+ - (binary)	Left	
<<>> <<<>>>	Left	
< <= > >= inside dist	Left	
== != === !== ==? !=?	Left	
& (binary)	Left	
^ ~^ ~^ (binary)	Left	
(binary)	Left	
&&	Left	
	Left	
? : (conditional operator)	Right	
->	Right	
= += -= *= /= %= &= ^= = <<= >>=	None	
<<< >>= := :/ <=		
{ } {{}}	Concatenation	Lowest

62 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Array Assignment Patterns



IEEE 1800-2012 10.9.1

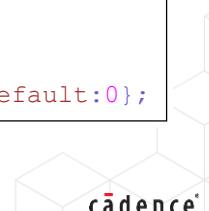
- Define a list of values for an assignment to:
 - Array elements
 - Structure fields (see later)
- It is equivalent to individual assignments:
 - Un-sized literals are allowed
- Pattern “keys” can be used:
 - Array element index
 - default keyword
- There must be a pattern value for every array element:
 - Either explicitly or using default

```

int arr1 [3:0];
arr1 = '{0,1,2,3};
/* equivalent to
arr1[3] = 0;
arr1[2] = 1;
arr1[1] = 2;
arr1[0] = 3;
*/
arr1 = '{3:1, default:0};
/* equivalent to
arr1[3] = 1;
arr1[2] = 0;
arr1[1] = 0;
arr1[0] = 0;
*/
// initialize mem to 0
reg [15:0] mem [0:1023] = '{default:0};

```

63 © Cadence Design Systems, Inc. All rights reserved.



You can make pattern assignments to unpacked arrays and unpacked structures (structures described in the User-Defined Data Types module).

Assignment patterns are an aggregate assignment, equivalent to an individual assignment to each array element. It is a compilation error if the number of values in the pattern is more than or fewer than the number of elements in the target array.

The assignment pattern starts with an acute accent ' , as opposed to a Verilog compilation directive, which begins with a grave accent ` .

Assignment pattern syntax is very similar to a concatenation, but unlike concatenation, individual values do not need to be sized. As the construct expands to individual assignments, SystemVerilog pads or truncates each assignment as needed.

You can make pattern assignments by position or by key, but not by both in the same assignment. For arrays, the key can be an element index or the default keyword, which assigns the default value to all elements not otherwise specified in the pattern.

Use of the default key is the only situation in which it is allowed to have fewer values in the pattern than elements in the array.

For structures, the key can be a field name, a field type, or the default keyword. See the User-Defined Data Types module for more information.

Module Summary

In this module, you learned how to

- Use blocking only assignment operators for combinational logic or verification code
- Employ pre- and post-increment/decrement operators, particularly useful in `for` loop constructs
- Assignment patterns (plus enhancements!):
 - Useful for multidimensional arrays
 - Also essential for structures (see next module)
- Use Wild equality/inequality operators
- Use Set membership (`inside`) operator



This page does not contain notes.

Quiz



What is the value of each expression or the output of each conditional branch?

OUTPUT

branch one
branch four

```
initial begin
    a = 4'b00xx;
    b = 4'b0001;
    c = 4'b0100;

    b++;
    c += b;
    c >= b;

    if (b ==? a)
        $display("branch one");
    else
        $display("branch two");

    if (c inside {4'b0110, 4'b0010} )
        $display("branch three");
    else
        $display("branch four");
    ...
}
```

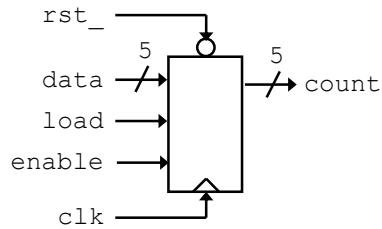
This page does not contain notes.



Lab

Lab 3 Modeling a Simple Counter

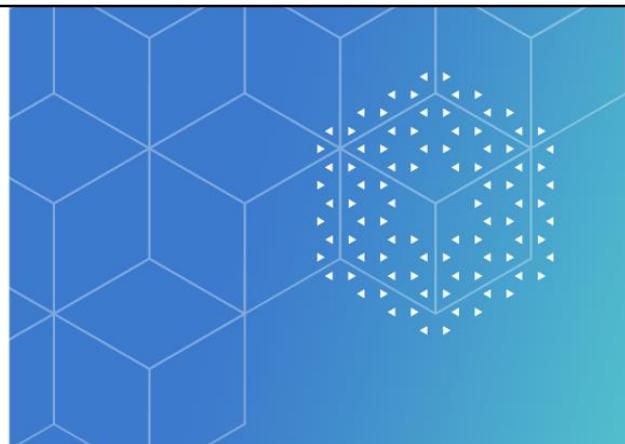
- Model a simple counter using SystemVerilog constructs



66 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



User-Defined Data Types and Structures

Module **6**

Revision **1.0**
Version **21.10**

Estimated time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Module Objectives

In this module, you

- Name type declarations in SystemVerilog with `typedef`
- Create types with user-defined, enumerated values
- Use structures to create arrays of different data types
- Identify the use of packages



This page does not contain notes.

Primitive Types and User-Defined Types



- **Primitive Data Types:** Built-in types not constructed from other data types.
- **User-Defined Types:** SystemVerilog's data types can be extended with user-defined types using `typedef`.

- User-defined data types are:
 - Types named with `typedef`.
 - Types constructed from other types, such as vectors and arrays.
 - Enumerate (enum) types for user-defined value sets.
 - Structure (struct) types to bundle multiple variables into one object.
 - union types to store different data types in the same space.

```
logic [7:0] vec8_t;  
bit     data;  
int    digit;
```

Primitive Data Types Usage

```
typedef logic [7:0] vec8_t;  
...  
vec8_t vec_1, vec_2;
```

`typedef` is used to give a user-defined name to existing 'logic' data type

The named data type can be used now to declare other variables

SystemVerilog divides types into two groups:

- A primitive data type is a built-in type that is not constructed from other types.
- Any other data type is user defined. This includes types that you name with the `typedef` keyword and types you construct from other types, such as arrays, enumerations, structures and unions.

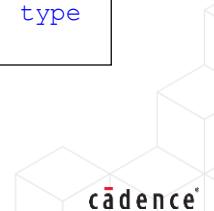
User-Defined Types: Examples

- You can declare and name a type using `typedef`:
 - Verilog standard types
 - SystemVerilog user-defined types
- The type name can be used anywhere, the type declaration is legal:
 - For example: variable or port declaration
- You must declare `typedef` before the name is used.
- The declaration must be visible in the scope where the name is used.
- An un-named type is known as an anonymous type.

```
typedef logic [7:0] vec8_t;  
  
input vec8_t op1;  
vec8_t busa, busb;  
function vec8_t checksum;  
...  
...
```

```
// typed (named) type  
typedef logic [15:0] vec16_t;  
vec16_t wordvar;
```

```
// anonymous (un-named) type  
logic [16:0] wordvar;
```



A `typedef` declares and names a user-defined type, allowing the name to be used in the declaration of variables, ports and other types. You must declare a `typedef` before you use the name, either:

- Inside any declarative scope (for example, a module), which makes it visible only within that scope, and any nested declarative scopes.
- In the compilation unit. (See the Hierarchy and Connectivity module for more details.)
Or
- Inside a package, which makes it visible in all design elements that import the package. (See the Hierarchy and Connectivity module for more details.)

What Is an Enumerated Type?



An enumerated type is a set of integral named constants.

- It is a type with user-defined values:
 - Value names must be unique in the current scope.
- Declared with the enum keyword:
 - Named or anonymous.
- Enumerated types are strongly typed.
- Should only be directly assigned:
 - A named value of the type.
 - A variable of the same type.
- Naming the type (typedef) permits typecasting:
 - `type_name' (value)`

Not Legal Direct Assignment
of Integral value
to enum variable is not allowed

```
// named enum type
typedef enum {idle, start,
             pause, done} state_t;
state_t state, next_state;

// anonymous enum type
enum {idle, start,
       pause, done} astate;

...
state = idle;           Legal
state = next_state;     Legal
state = 2'b00;          Legal
state = 2;               Legal
state = state_t'(2);    Legal
...
```



An enumerate type is a user-defined type where the user defines the value set of the type. A SystemVerilog enumerate type is declared with the keyword `enum` followed by a list of values for the type. The value names must be unique in whichever scope the enumerate type is used. A common policy is to use uppercase names for values to ensure uniqueness.

Enumerate type declarations can be typed (with `typedef`) or anonymous. There are significant benefits to naming the enumerate type as we shall see.

SystemVerilog enumerate types are strongly typed. An enumerate variable can be assigned only from a variable of the same type or from an enumeration value of its type.

To assign any other value requires a static cast to the enumerated type. A cast can only be made using the name of the enumerate type. Casting the value to the enumerated type does not check the validity of the value (see following slides).

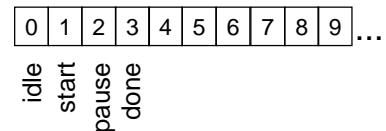
Traditionally, Verilog is not a strongly typed language. Therefore some compilers allow the direct assignment of an integral value to an enumerated type variable, but generate a warning.

Tip: Always use a `typedef` to declare your enumerate types.

Enumerated Type Values

- Default base type of an `enum` is `int`.
- Default values increment from zero in declaration order.
- An enumerate variable can hold any value from the base type:
 - Even if there is no matching enumerate value name.
 - Casting does not check value.
- An enumerated type is the base type with mnemonics for specific values.
- An enumerate variable in an expression is replaced by its base type value.

```
typedef enum {idle, start,
              pause, done} state_t;
// idle=0; start=1
// pause=2; done=3
```



```
state_t state;
int aint;
initial begin
    state = state_t'(2); // pause
    state = state_t'(8); // no name

    aint = state * 2;    // 16
    ...

```

72 © Cadence Design Systems, Inc. All rights reserved.



By default, the base type of an enumerate type is `int`.

By default, value encodings for enumerated types are controlled by the order in which they are declared. The first value is represented by 0, second by 1 and so on.

An enumerate type can be considered as its base type (default `int`) with names (mnemonics) for specific values. `state` can be assigned (via casting) any `int` value.

Therefore:

```
state = state_t'(8);
```

is legal, even though there is no named value in the enumerate declaration corresponding to the integral value 8.

When you use an enumerate variable in an expression, SystemVerilog automatically converts the enumeration value to its base integral value and uses the integral value to evaluate the expression. If you wish to assign the expression result back to the enumerate variable, you must use an explicit cast:

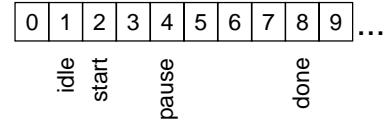
```
state = state_t'(state *2);
```

How to Assign Enumerate Value Encodings Explicitly



Explicit encoding can be included in an enumerated type declaration, either typed or anonymous which allows one-hot, grey or other encodings to be defined for enumerated values.

```
typedef enum {idle = 1,
              start = 2,
              pause = 4,
              done = 8} state_t;
```

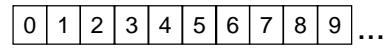


1. You can define explicit encoding for values:
 - Allows one-hot coding, etc.

2. You can mix explicit and implicit value encoding:
 - Implicit values increment from previous explicit value.

3. Value encodings must be unique:
 - Compilation error is reported if encodings overlap.

```
enum {S0=0, S1, S2=6, S3} encs;
```



```
enum {P0=0, P1, P2=1, P3} bad_encs;
```

P2 duplicates P1 encoding

Explicit encoding can be included in an enumerated type declaration, either typed or anonymous. This allows one-hot, grey or other encodings to be defined for enumerated values.

For 2-state types like the default *int* base type, you can use only the binary values (no Z or X).

You can mix explicit and implicit value encoding in the same enumerated type declaration. For any value without an explicit encoding, the encoding is simply an increment of the encoding for the previous enumerate value; hence for the variable *encs*, S1 has the encoding 1, and S3 = 7.

It is a compilation error if two or more values have the same encoding; e.g., for variable *badencs*, both P1 and P2 have an encoding of 1, P1 by implicitly following the encoding for P0, and P2 by explicit declaration.

Note: For these encodings to be reflected in hardware, the synthesis tool must support explicit encoding. If, for example, you want to optimize the encoding of a state variable, it may be better to allow the synthesis tool to choose its own encoding based on your synthesis goals.

Enumerated Type Value Name Sequences

- You can use a range notation for simple value name sequences.
 - You can mix ranged and unranged values.

Equivalent to

```
typedef enum {S[2]} seqa_t;
```

```
typedef enum {S0, S1} seqa_t;
```

Equivalent to

```
typedef enum {go, R[3:5], stop} seqb_t;
```

```
typedef enum {go, R3, R4, R5, stop} seqb_t;
```



You can specify a range of enumeration elements. This automatically creates names for the elements and assigns incremental values to the names.

Assigning Enumerate Base Type Explicitly



IEEE 1800-2012 6.19

- You can specify the enumeration base type:
 - Base type int by default
- This allows you to constrain the enumerate value range.
- The default value encoding will match the specified base type.
- Explicit value encoding must match type and (if explicit) length.



An explicit base type makes enumerates easier and safer to use.

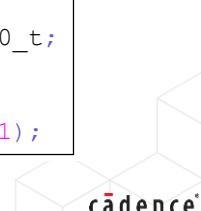
```
typedef enum bit[1:0]
  {idle, start, pause, done} base_t;
// idle = 2'b00, start = 2'b01
// pause = 2'b10, done = 2'b11
```

00	01	10	11
idle	start	pause	done

Enum Declaration with explicit base type and implicit encoding

```
typedef enum bit[2:0]
  {idle = 3'b000,
   start = 3'b001,
   pause = 3'b010,
   done = 3'b100} onehot0_t;
onehot0_t state;
...
state <= onehot0_t'(3'b001);
```

75 © Cadence Design Systems, Inc. All rights reserved.



The base type of an enumerated type defaults to `int`. As we have seen, this means a default enumerate variable can be assigned any value in the `int` range (32 bits).

You can define an explicit base type for the enumerate. This allows you to constrain the size of the base type and hence limit the range of values that can be assigned into the type by casting.

You can declare base types of the atomic integer types (`byte`, `shortint`, `int`, `longint`, `integer`, `time`) or integer vector types (`bit`, `logic`, `reg`).

If you define an explicit base type, then any explicit value encodings must match the base type and size. In the example, for type `onehot0_t`, as the base type is of size 3, the explicit encodings must all be sized to length 3. Truncation and padding assignment rules do not apply when defining explicit encodings. This is to ensure every value has a unique encoding.

Enumerated Base Type vs. Enum Initial Value

- The initial value of an enum variable depends on its base type:
 - 0 for default int type and any explicit 2-state type.
 - X for an explicit 4-state type.
- You can explicitly encode a value with X:
 - Typically, to indicate an illegal value.

```
enum bit[2:0]{BONE, BTWO} var_b;
initial begin
  $display("value %b", var_b);
  $display("name %s", var_b.name());

```

value 000
name BONE

```
enum logic[2:0] {LONE, LTWO} var_1;
initial begin
  $display("value %b", var_1);
  $display("name %s", var_1.name());

```

value xxx
name

Empty string

```
enum logic[2:0] {LX=3'bX, LONE=3'b001, ...} var_1;
initial begin
  $display("value %b", var_1);
  $display("name %s", var_1.name());

```

value xxx
name LX

76 © Cadence Design Systems, Inc. All rights reserved.



The initial value of an enumerate variable depends on the base type of the enumerated declaration.

- For the default int base type, the initial value is 0, which would match the default encoding of the first enumeration value. Therefore, by default, the variable initializes to the first enumerate value.
- For any explicit 2-state type (e.g., bit array), the initial value is 0, which would match the default encoding of the first enumeration value.
- For an explicit 4-state base type (e.g., logic array), the initial value is all bits unknown ('x), which would match only an explicitly encoded value.

In the second example, the initial value of enumerate var_1 is 3'bxxx. As this value does not match the encoding for any of the declared values, the name() method returns an empty string.

With a 4-state explicit base type, you can encode a value with unknowns ('x). This could be used, for example, to indicate that the variable is invalid and detect sampling problems. However, an explicit unknown value will cause problems if the variable is intended to be synthesized.

Quick Reference Guide: Enumerated Type Access Methods



Method	Description
first()	Returns first value
last()	Returns last value
next (N)	Returns next Nth value from current
prev (N)	Returns previous Nth value from current
num()	Returns number of values
name()	Returns string equivalent of value

```
typedef enum {SA, SB, SC, SD} state_t;
state_t st = st.first();
...
repeat(5) begin
    $display ("%s = %0d", st.name(), st);
    if (st == st.last())
        $display ("-----");
    st = st.next();
end
...
```

Iterates over defined values only

SA = 0
SB = 1
SC = 2
SD = 3

SA = 0

```
state_t op;
...
op = op.first();
repeat (op.num())
begin
    ...
    op = op.next();
end
...
```

Iterate over every value of an enumerate



77 © Cadence Design Systems, Inc. All rights reserved.

This page does not contain notes.

What Is a Structure?



A structure represents a collection of data types that can be referenced as a whole, or the individual data types that make up the structure can be referenced by name.

- Declare using the `struct` keyword.
- Use the “dot” notation to access individual fields.
- You can make pattern assignments to structures:
 - Ordered or keyed:
 - Not both in the same pattern.
 - Keys can be by name, type, default or a mix of these.
- You can declare arrays of structures.
- Structures can be nested.

```
typedef struct {
    logic      id, par;
    logic[3:0] addr;
    logic[7:0] data;
} frame_t;

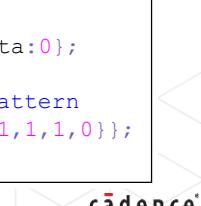
frame_t f1, two_frame[1:0];
logic[7:0] data_in;
...
// individual field access
f1.id = 1'b1;
data_in = f1.data;

// ordered assignment pattern
f1 = '{1'b1, 1'b1, 4'h0, 8'hff};

// named assignment pattern
f1 = '{id:0, par:1, addr:0, data:0};

// nested ordered assignment pattern
two_frame = '{'{0,0,0,255}, '{1,1,1,0}};
```

78 © Cadence Design Systems, Inc. All rights reserved.



A structure is an array of elements that can be of different data types. It can be used to group complex data information, which would be otherwise spread over several separate variables into one object.

The individual fields of the structure can be accessed from a structure variable using the dot notation.

By default, the fields of a structure are stored and must be accessed separately. Therefore, the easiest way to load a structure is using a pattern assignment. The assignment must be made to every field but can be either ordered or keyed. With structures, the key can include assignment by type as well as assignment by name and default (see next slide).

Structures are just like any other type in SystemVerilog, and therefore you can declare arrays of structures or structures containing fields that are of another structure type (nested structures).

The next slide describes pattern assignments in more detail.

How to Assign Values to Structure Elements



Structure can use either ordered or keyed assignment patterns for assigning values to structure elements.

Order of precedence:

1. Name
 - Assign fields by name.
2. Type
 - Assign remaining fields of a specific type.
3. Default
 - Assign remaining fields using default.
 - Value must be type compatible with all remaining fields.

All fields must have an assignment.

```
typedef struct {
    logic      id, par;
    logic[3:0] addr;
    logic[7:0] data;
} frame_t;
frame_t f1;

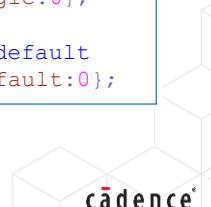
// assignment by order
f1 = '{1'b0, 1'b0, 12, 8'hff};

// assignment by name
f1 = '{data:8'hff, addr:12, par:0, id:1};

// assignment by name and type
f1 = '{data:8'hff, addr:12, logic:0};

// assignment by name, type & default
f1 = '{data:8'hff, logic:1, default:0};
```

79 © Cadence Design Systems, Inc. All rights reserved.



Assignment patterns can use either ordered or keyed assignments but cannot use both in the same pattern.

When using an assignment pattern, all fields must be assigned a value, either explicitly or using the *default* key.

Key precedence is by (highest to lowest):

- Name, using the field name.
- Type, selecting all fields of the specified type.
- Default, which applies to all remaining unassigned fields. All fields to which the default value applies must be of a type compatible with the default value specified.

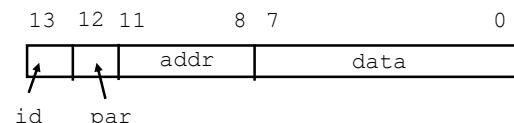
What Is a Packed Structure?



Packed Structure fields are packed together in memory without gaps and can be used as a whole with arithmetic and logical operators.

- Structures can be packed.
- All fields must be “packable”:
 - Integral values only.
 - Any field that is array or structure must also be packed.
- A packed structure can be treated as a one-dimensional vector.
 - Can still use “dot” notation to access individual structure fields.
- If any field is 4-state, then the whole structure is stored as 4-state.
 - Conversions are done automatically upon read/write.

Avoid mixing 2-state and 4-state types



```
typedef struct packed {
    logic     id, par;
    logic[3:0] addr;
    logic[7:0] data;
} frame_t;

frame_t f1, f2;

logic [3:0] addrbuf;
initial begin
    addrbuf = f1.addr;
    addrbuf = f1[11:8];
    f2 = f1 >> 2;

    f1 = 8'haa;           //14'h00aa
    f1 = 16'h5555;        //14'h1555
    ...

```

All fields are 4-state types

Same field

Padding

Truncation



80 © Cadence Design Systems, Inc. All rights reserved.

By default, a structure's fields are stored and assigned individually. However, a structure can be stored, and accessed, as a single one-dimensional array by using the keyword `packed`.

There are some limitations on the field types for a packed structure. Only integral fields can be packed. Multidimensional arrays are allowed if they are also packed (see module on static arrays) and any nested structure fields must also be packed.

A packed structure can be assigned, read, sliced, shifted and operated upon as if it were a single one-dimensional array. There are two ways to access the fields of a packed structure. You can still use the dot notation to access a field, but you can also slice the structure.

For example, `f1.addr` and `f1[11:8]` both access the same information.

In general, you should avoid mixing 2-state types like `bit` and 4-state types like `logic` in a structure declaration. Mixing the types means the entire structure is stored as a 4-state type, but automatic conversion to 0 or 1 is carried out when a 2-state field is accessed. If you wrote X into a 4-state field and shifted the value into a 2-state field, the packed structure will still be storing an X in the 2-state field, but it would be read as a 0.

New Type Declaration Regions



IEEE 1800-2012 26.0

- You can declare ports of user-defined types, but you must declare types before you use them:
 - How do modules see the type definition?
- SystemVerilog has two new name spaces:
 - Packages
 - Compilation Unit Scope (CUS):
 - To be avoided (see later for details)
- Both regions are described in more detail in a later module...

```
package mytypes;
  typedef enum {start,done} mode_t;
  ...
endpackage : mytypes
```

Package declaration

```
module mtwo import mytypes::*; (
  input logic [7:0] out,
  output mode_t mode);
  ...

```

Package imported in module header

```
typedef enum {start, done} mode_t;

module mone (
  input mode_t mode,
  output logic [7:0] out);
  ...

```

Declaration in CUS

Not Recommended

81 © Cadence Design Systems, Inc. All rights reserved.

You must declare types before you attempt to use them. This presents a problem when you need to declare ports of a user-defined type using Verilog-2001 syntax. To resolve this, SystemVerilog adds two new declarative regions which can be used for sharing type declarations:

- The package is a new design element for shared declarations. The package is like a module – it is usually declared in its own file and must be separately compiled before any other modules or packages which reference the package. You reference declarations from the package by using `import` statements or by using resolved pathnames.
- The Compilation Unit Scope is a declarative region immediately before the module definition in a SystemVerilog file. The Compilation Unit Scope should generally be avoided if possible.

The Hierarchy and Connectivity module explores both mechanisms in more detail.

Module Summary

In this module, you learned about

User-defined data types (`typedef`):

- Lets you define a type that is used throughout the design
 - Example: Modifying bus widths without using parameters

Enumerations (`enum`):

- Name states in a state machine or op-codes in an instruction set
- Easier and safer than using parameters or `'define`

Structures (`struct`):

- A vector of different data types, referenced by a single name
- Makes for more readable code, with less typing
- Simplifies the passing of data across module/task/function boundaries

Packed data types:

- Allows bit-vector operations across packed structs



This page does not contain notes.

Quiz



Which of these statements in the code snippet are incorrect? How could you correct them?

```
typedef enum logic[1:0] {up=2'b00, stay=2'b01, down=2'b11} cstate_t;  
cstate_t cstate;  
  
typedef struct {int tag;  
               cstate_t mode;  
               logic[7:0] data;} tagdata_t;  
tagdata_t tdata;  
  
typedef enum {idle, go, stop, stay, reset} mechstate_t;  
  
logic [7:0] slice;  
  
initial begin  
    tdata.mode = reset;  
    tdata = '{tag:4, data:4, default:4};  
    slice = tdata[7:0];  
    cstate = cstate + 1;  
    cstate = cstate_t'(2'b10);  
    slice = cstate + 1;  
end
```

Duplicate definition for stay

Unknown value reset in cstate_t

Invalid value 4 for mode field

Cannot slice unpacked structure

Cannot assign int to cstate_t



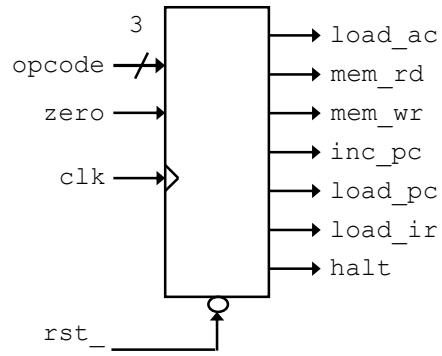
This page does not contain notes.

Lab



Lab 4 Modeling a Sequence Controller

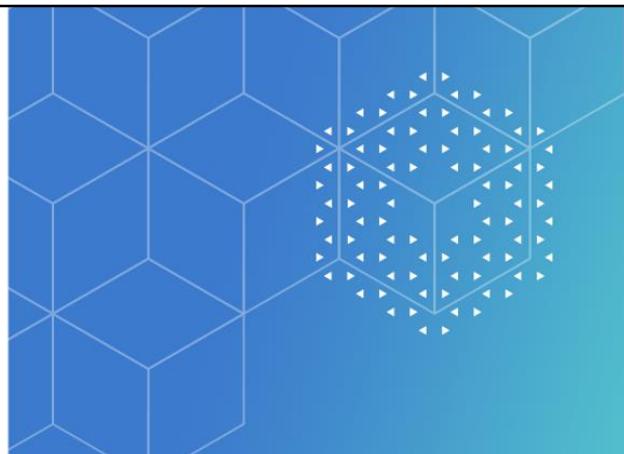
- Model a simple FSM using SystemVerilog constructs including enumerate data types, enumeration methods and packages.



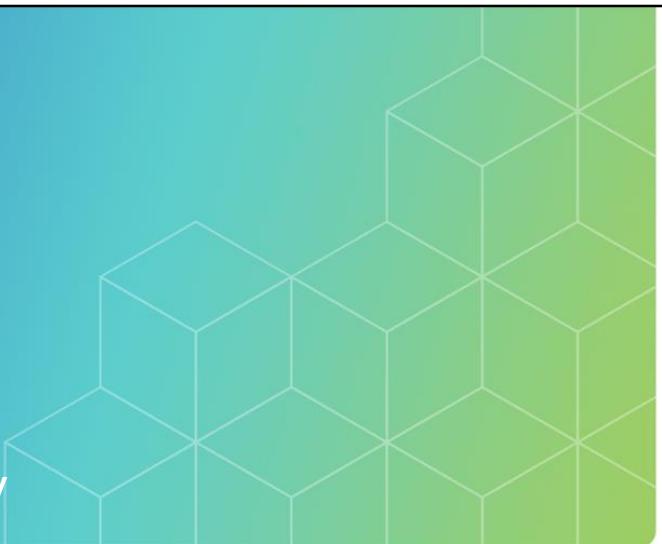
84 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



Hierarchy and Connectivity



Module 7

Revision 1.0
Version 21.10

Estimated time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Module Objectives

In this module, you

- Use implicit .name and .* port connections in module instantiations
- Define and reference declarations in packages
- Identify the issues involved with the compilation unit scope



This page does not contain notes.



Quick Reference Guide: Verilog Port Connections

Ordered Port Connections

Ordered port connections, which have the risk of incorrectly ordered connections

Ordered

```
counter c1 (clk, rst, ld, data, cnt);
```

Badly ordered

```
counter c2 (clk, ld, rst, data, cnt);
```

SystemVerilog adds two options to simplify port connections:

- .name (dot-name)
- .* (dot-star)

```
module counter (
    input logic clk, rst, ld,
    input logic [7:0] data,
    output logic [7:0] cnt);
    ...
endmodule
```



For Verilog, you can use implicit ordered port connections or explicitly named port connections.

Ordered port connections map the signals of the instantiation to the ports of the module in the order of declaration, the first signal to the first port, etc. This runs the risk of listing the signals in the wrong order. Detecting badly ordered port connections in simulation can be very difficult.

Named port connections explicitly identify which signal is mapped to which port, greatly reducing the risk of incorrect connections. Named port connections are usually the preferred option but are very verbose.

SystemVerilog adds two options to simplify named port connections, referred to as dot-name and dot-star.

Implicit Port Connection: .name (dot-name)



IEEE 1800-2012 23.3.2.3

- Where signal and port names match, just use name once:

```
.clk = .clk(clk)
```

This is as safe, and not as verbose, as a named connection.

- It can be mixed with a named connection:
 - For ports where names do not match.
- Signals used in .name must be explicitly declared.

```
module count (
  input logic clk, rst, ld,
  input logic [7:0] data,
  output logic [7:0] cnt,
  output logic val);
  ...
endmodule
```

.name port connection

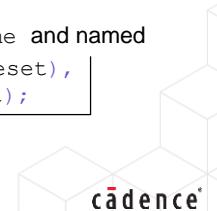
```
count c4 (.clk, .rst, .ld,
          .data, .cnt, .val);
```

Named equivalent

```
count c5 (.clk(clk), .rst(rst), .ld(ld),
          .data(data) .cnt(cnt), .val(val));
```

.name and named

```
count c6 (.data, .clk, .rst(reset),
          .ld(load), .cnt);
```



Where the net or variable name matches the port to which it is connected, SystemVerilog allows you to just use the name once (dot-name). This is equivalent to a named connection but is easier to write.

For example, where a variable `clk` is connected to a port `clk`, instead of writing the named connection `.clk(clk)` SystemVerilog allows you to simply write `.clk`.

You can use a dot-name port connection only if the name, size and type of the port match that of the connecting net or variable.

Dot-name can be mixed with full-named connections in the same module instance to make connections where the port name and signal name do not match.

A SystemVerilog dot-name port connection is equivalent to a Verilog named port connection with the following exceptions:

- Signals used in a dot-name connection must be explicitly declared. In Verilog, an undeclared identifier defaults to a scalar `wire`. It is not uncommon to use such implicit net declarations in Verilog module instantiations; however, these implicit declarations cannot be used for dot-name connections.
- Dot-name does not allow explicit or implicit typecasting between the port and the connected signal, not even padding or truncation. However, there are two exceptions:
 - It automatically converts between 2-state and 4-state types of the same length.
 - It automatically converts between a net and a variable of the same length.

Implicit Port Connection: . * (dot-star)



IEEE 1800-2012 23.3.2.4

- Where all signal and port names match, just use . *
- It automatically connects ports to signals of the same name.
- There is safety of named connection without verbosity:
 - Can lose some port list readability.
- It can be mixed with a named connection:
 - For ports where names do not match.
- Signals used in . * must be explicitly declared.

```
module count (
  input logic clk, rst, ld,
  input logic [7:0] data,
  output logic [7:0] cnt,
  output logic val);
  ...
endmodule
```

.* port connection

count c7 (. *);

↓

Dot-name equivalent

count c7 (.clk, .rst, .ld,
.data, .cnt, .val);

.* and named

count c9 (. *, .rst(reset), .ld(load));

cadence®

89 © Cadence Design Systems, Inc. All rights reserved.

Where all net or variable names match the port to which they are connected, SystemVerilog allows you to simply write . * (dot-star). This is equivalent to a named connection for every port, but is far easier to write, if much harder to read.

You can use a dot-star port connection only if the name, size and type of the port match that of the connecting net or variable.

Dot-star can be mixed with full named connections in the same module instance to make connections where the port name and signal name do not match.

The dot-star connection is subject to the same rules as the dot-name connection. In particular, signals used in a dot-star connection must be explicitly declared. In Verilog, an undeclared identifier defaults to a scalar `wire`. It is not uncommon to use such implicit net declarations in Verilog module instantiations; however, these implicit declarations cannot be used for dot-name connections.

Rules for Using .name and .*

- In an instantiation:

- Ordered connections with .name or .* cannot be mixed.
- .name and .* connections cannot be mixed.
- Named and .name connections can be mixed.
- Named and .* connections can be mixed.

- Named connections are required for:

- Port/signal name mismatches.
- Port/signal width mismatches.
- Unconnected ports.

Ordered and dot-star

```
count c10 (clock, reset, .*);
```

Dot-name and dot-star

```
count c11 (.clk, .rst, .*);
```

Dot-name and named

```
count c12 (.data, .clk, .cnt, .val,  
.rst(reset), .ld(load));
```

Dot-star and named

```
count c13 (*,  
.ld(load),  
.data(data[15:8]),  
.val());
```



Restrictions on SystemVerilog port connections include the following:

- You cannot use Verilog-ordered connections with any other kind of port connection in the same instance.
- You cannot mix a dot-star connection with dot-name connections in the same instance. The dot-name connections would be redundant, but this is still a compilation error.
- You can use Verilog-named port connections with either SystemVerilog dot-name connection or SystemVerilog dot-star connection.

Verilog-named connections are still essential to connect ports where names or widths are not matched or where the port is unconnected or connected to an expression of signals.



Port Connections and Their Advantages

Dot-Star Advantages	Dot-Name Advantages	Named Advantages
<ul style="list-style-type: none"> Connecting hierarchy is much easier: <ul style="list-style-type: none"> Much less verbose than any of the alternatives As safe as a named port connection Exceptions to matching port and signal names clearly stand out It is very useful for top-level block connections and testbenches: <ul style="list-style-type: none"> Block and testbench variable names can match port names 	<ul style="list-style-type: none"> Connecting hierarchy is much more likely to be correct: <ul style="list-style-type: none"> Less connection debugging Port-to-signal association is retained: <ul style="list-style-type: none"> Parent and child both have port list 	<ul style="list-style-type: none"> Needed where port and variable names do not match: <ul style="list-style-type: none"> Low-level netlists Structural designs Multiple instances of a module

91 © Cadence Design Systems, Inc. All rights reserved.



A dot-star connection has both benefits and drawbacks.

Your code is much less verbose and promotes the use of the same signal names across hierarchy boundaries, which is a good design practice. When used with named connections, they also clearly highlight where the signal name and port name do not match, which makes the code more readable.

However, the dot-star connection no longer contains the instance port names, which affects both readability and design debug.

The dot-name connection has the best of both worlds – the safety of a named connection but less verbose coupled with the visibility of the port list.

New Type Declaration Regions



New Type Declaration Regions are spaces used to share parameters, data, type, task, function, sequence, property, and checker declarations among multiple SV modules, interfaces, programs and checkers.

SystemVerilog has two new options for sharing declarations:

1. Compilation Unit Scope:

- To be avoided – see next slide

2. Packages

Verilog uses the `include compiler directive to share declarations among modules that can cause compilation issues.

Using mode_t declaration of package in module

```
package mytypes;
    typedef enum {start,done} mode_t;
endpackage : mytypes

module mone(...);
    import mytypes::*;
    mode_t mode,
    ...

```

Compilation Unit Scope

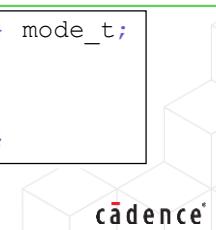
Using mode_t declaration of CUS in module

```
typedef enum {start, done} mode_t;

module mone(
    input mode_t mode,
    output logic [7:0] out);
```



Not Recommended



In Verilog, we use the `include compiler directives to allow declarations to be shared among multiple modules. The compiler replaces the directive with the referenced code. This can cause issues; for example, an included file may compile in one location but not another. To resolve these issues, SystemVerilog adds two new declarative regions which can be used for sharing declarations:

- The package is a new design element for shared declarations. The package is like a module – it is usually declared in its own file and must be separately compiled, before any other modules or packages which reference the package. You reference declarations from the package by using import statements or by using resolved pathnames.
- The Compilation Unit Scope is a declarative region immediately before the module definition in a SystemVerilog file. The Compilation Unit Scope should generally be avoided if possible.

1. Compilation Unit Scope (CUS)

- New scope for declarations is:
 - Outside of design elements (modules, interfaces, etc.).
- Default scope is a single file:
 - Tool options can change this:
 - For example: defining scope as multiple files compiled at same time.
- Scope is unnamed:
 - Cannot make hierarchical references to declarations.
- Local declarations override declarations in CUS:
 - Use `$unit::` to access CUS declarations.

```
modules.sv
typedef enum {start, done} mode_t;
localparam eseg = 7'b11111100;

module mone (input mode_t mode,
              output logic [7:0] out);
  ...
endmodule : mone

module mtwo (input logic [7:0] out,
              output mode_t mode,
              output [6:0] oa, ob);
  localparam eseg = 7'b00111111;
  ...
  assign oa = eseg;           // local
  assign ob = $unit::eseg;    // CUS
endmodule : mtwo
```



Scope of compilation unit is tool-specific and can change between compile sessions. Local declarations override CUS declarations.

The Compilation Unit Scope (CUS) is a new scope for declarations. Declarations are placed inside a file but outside of a design element (module, package, interface, etc.). The scope of declarations in a CUS are, by default, restricted to the file in which they are declared. By declaring several design elements in the file, the CUS declarations can be shared between the design elements. The SystemVerilog standard also requires a compiler to be able to extend the visibility of a CUS declaration to all files compiled at the same time. Vendors may also provide other options.

A CUS is equivalent to an anonymous package. As it is anonymous, there is no name for referencing the scope, so you cannot use a hierarchical pathname to reference any CUS declaration.

A CUS type declaration is visible to any design elements defined in the same compilation unit scope. Local declarations within the design element can override those in the CUS by using the same identifier. This follows the normal Verilog rules that declarations within the most local scope take precedence over all others. You can explicitly access a CUS declaration by using a scope resolution operator (`:::`) prefixed with dollar-unit (`$unit`). This allows a design element to access a CUS declaration that is otherwise overridden by a local declaration.

By default, Cadence® tools will treat each file list in a compile run as a separate CUS. Multiple compile runs (followed by a separate elaboration step) will create multiple CUSSs. There is also the simulator option `-scu`, which puts each individual file in a separate CUS.

2. Packages

- New design element similar to a module:
 - Must be compiled separately.
 - Must be compiled before elements that reference the package.
- They contain declarations to be shared between elements:
 - Types, variables, subroutines...
- You can import package declarations:
 - Explicit – specifically named.
 - Implicit – all using wildcard (*).
- Or directly access a declaration using the resolution operator (::):
 - Does not require import.

```
package mytypes;
  typedef enum {start,done} mode_t;
endpackage : mytypes
```

Explicit import into CUS

```
import mytypes::mode_t;
module mone (input mode_t mode...);
```

Wildcard import

```
module mone (...);
  import mytypes::*;
  mode_t mode,
  ...
```

Resolved name

```
module mone (...);

  mytypes::mode_t mode,
  ...
```



94 © Cadence Design Systems, Inc. All rights reserved.

The package is a design element. A design element is a top-level simulation building block that is typically declared in a separate file and must be separately compiled. Verilog has three design elements – module, configuration and primitive. SystemVerilog adds package, interface and program.

A package allows you to share declarations between multiple design elements. You can place almost any declaration – types, variables, tasks and functions – within a package.

You can reference packages from within other design elements – modules, interfaces, programs, and other packages. There are three ways to reference the declarations from a package:

- An explicit import allows you to reference selected package declarations.
- An implicit wildcard import allows you to reference all the package declarations.
- You can also reference a declaration using the package name and the scope resolution operator.

The first example explicitly imports the mode_t type.

The second example implicitly imports the entire contents of the package.

The last example uses a resolved pathname to the package declaration and does not import the declaration. Any other references to the package declarations must also use a resolved name in the absence of an import statement. Resolved names are sometimes used in addition to package imports for clarity.

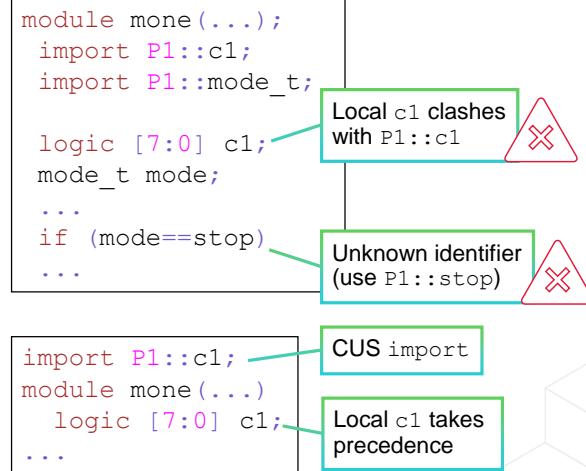
What Is an Explicit Import?



An explicit import only imports the symbols specifically referenced by the import.

- Explicit import directly loads declaration into the module.
- Declaration must be unique in the current scope:
 - Compilation error if local or other explicitly imported declarations have same name.
- Only the symbols specifically referenced are imported.
- An explicit import in a CUS follows compilation unit rules:
 - Local declarations override CUS declaration.
 - Even for an explicit import.
 - Compilation Unit Scope should be avoided if possible.

```
package P1;
  localparam int c1 = 10;
  typedef enum {start,stop} mode_t;
endpackage : P1
```



95 © Cadence Design Systems, Inc. All rights reserved.

cadence®

An explicit import directly loads a package declaration into the design element as if it was declared in the design element. As such, if the design element can have visibility of another declaration using the same identifier in the same scope, either through a local declaration or another explicitly imported package declaration, then it is a compilation error.

An explicit import only makes the identifiers in the import statement visible. For example, if you explicitly import an enumerated type, that does not automatically also import the enumeration value names. However, if you explicitly import a structure type, the field names are visible in the design element.

Explicit import statements in the Compilation Unit Scope (CUS) place those type declarations in the Compilation Unit Scope. These declarations are then subject to the normal rules for CUS declarations. Specifically, those local declarations can override the imported package declarations, even if the package declarations are explicitly imported. Also the extent of the import visibility can be changed with compilation options. For these reasons, use of the Compilation Unit scope should be avoided if possible.

What Is a Wildcard Import?



A wildcard import allows **all** identifiers declared within a package to be imported provided the identifier is not otherwise defined in the importing scope.

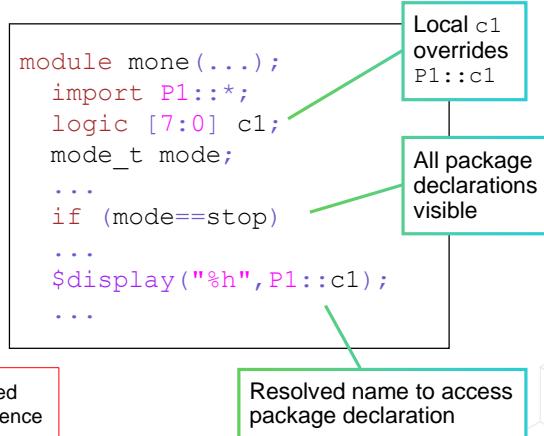
```
package P1;
  localparam int c1 = 10;
  typedef enum {start,stop} mode_t;
endpackage : P1
```

Wildcard import makes declaration candidate for import:

- Local or explicitly imported declarations can override wildcard imported declarations.
- Package declarations still visible through resolved name.



Local or explicitly imported declarations take precedence over wildcard imports.



96 © Cadence Design Systems, Inc. All rights reserved.

cadence®

An implicit wildcard import makes the package types candidates for import. It imports each type only when the type is used. If the design element can have visibility of another declaration using the same identifier, either through a local declaration or an explicitly imported package declaration, and the other declaration is visible before the identifier is used, then the wildcard imported declaration is overridden.

However, declarations from the package are still visible even if overridden by local or explicitly imported declarations, by using a resolved name.

A wildcard import makes all the declarations in the imported package visible, including enumerate type values.

Import and Export Statements

- The import is a statement that is equivalent to a data declaration.
- Declarations are only visible from the import onwards.
- Fewer visibility issues.
- Better readability.
- Imports do not chain:
 - Importing P1 into P2, and then P2 into mone does not make P1 declarations visible in mone.
 - P1 must be separately imported into mone...
 - ...or add an export to P2 to make P1 visible as part of P2.
 - export P1::*;
 - Rarely used.



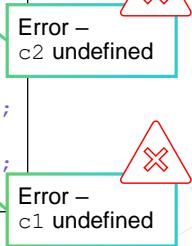
Place imports at the beginning of the design element.

```
package P1;
  int c1 = 5;
  ...
endpackage : P1
```

```
package P2;
  import P1::*;
  int c2 = c1;
  ...
endpackage : P2
```

Could add
export P1::*;

```
module mone(...);
  logic [7:0] d = c2;
  import P2::*;
  initial begin
    $display("%0d", c2);
    $display("%0d", c1);
  ...
endmodule
```



cadence

97 © Cadence Design Systems, Inc. All rights reserved.

The import statement is a declaration. You can place an import in any design element, such as an interface, module, program or other package, but not within a class scope.

Imported declarations are only visible from the point of the import statement onward, so you must import a declaration before you attempt to use it. It is a good tip to place all your imports at the top of the module, for better readability and fewer visibility issues.

In the example, the first reference to c2 fails because it comes before the import statement.

Another issue with imports is that they do not chain. In the example, package P2 contains an import for package P1, which allows it to access the value of c1. However, when we import P2 into the module mone, we only see the declarations from P2, not P1. Therefore, the reference to c1 fails. To see declarations from P1 in module mone, package P1 must be separately imported into the module.

An alternative is to make the declarations imported from P1 visible as part of the P2 package by adding an export statement to P2, although exports are rarely used in practice.

export P1::*;

The export can be wildcard or explicit. A special wildcard export form exports all declarations imported into a package:

export *::*;

Ambiguity and Resolved Names



How to resolve ambiguous references due to: "Multiple Declarations in separate packages using same name."

Solutions

- Try to avoid duplicate names
- Structure packages carefully
 - Only required declarations visible
- Use resolved names
- Use explicit imports

98 © Cadence Design Systems, Inc. All rights reserved.

```
package P1;
  typedef enum {start,done} mode_t;
  int c1 = 5;
  ...
endpackage : P1
```



```
package P2;
  int c1 = 8;
  ...
endpackage : P2
```

Solution 1

```
module mone(...);
  import P1::*;

  logic [7:0] d = c1;
  logic [7:0] e = P2::c1;
  ...

```

Error – ambiguous

OK – resolved

Solution 2

```
module mone(...);
  import P1::*;
  import P2::*;

  import P2::c1;

  logic [7:0] d = c1;
  ...

```

Explicitly importing c1 from package you want to always infer

OK – resolved

It is an error to import the same identifier more than once. For wildcard imports, the error occurs when you actually attempt to use the identifier.

In this example, packages P1 and P2 both declare the identifier c1. Wildcard importing of the contents of both packages gives a compilation error only upon the first use of c1.

An error would also occur if both c1 declarations were explicitly imported.

There are several solutions to this issue:

- Avoid duplicate identifiers where possible.
- Partition your packages carefully to separate out declarations that clash. It is generally better to have more, smaller packages rather than a few larger ones. This allows better control of declaration visibility.
- Use resolved names to explicitly identify which package declaration is visible.
- Explicitly import only the parts of each package that you need.

Importing Packages in the Module Header



How to import Package declarations that may be used in ANSI C module port lists...?

- import inside module is insufficient.

Several Solutions

- Use resolved name:
 - Inefficient for multiple declarations.
- Move import to Compilation Unit Scope:
 - But we're trying to avoid using this scope.
- Import package as part of the module header:
 - Imported before parameter and port lists.
 - Import clause must be terminated with a semicolon.

```
package P1;
  typedef enum {start,stop} mode_t;
endpackage : P1
```

```
module mone (
  input mode_t mode,
  ...);
  import P1::*;
  ...
endmodule
```

Error – mode_t undefined

```
module mone import P1::*;
  input mode_t mode,
  ...;
  ...
endmodule
```

import in module header

import must include semicolon

If you are using package declarations in an ANSI C module port or parameter list, then using an import statement inside the module is insufficient. The import is compiled after the package declaration use and so the declarations cause compilation errors.

There are several solutions to this issue:

- Use resolved names for all package declarations, although this is inefficient if you are using multiple declarations from the package.
- Move the import statement to the Compilation Unit Scope. This means the import statement is subject to the normal rules of this scope, specifically in that the visibility of the scope can be modified with compiler options. Therefore we should avoid this option if possible.
- Packages can also be imported as part of a module header. This makes all the package items visible throughout the module and these package items can also be used in parameter or port declaration of the module. This option can also be used with other design elements such as interfaces and programs.

The module import must come before the ANSI C port or parameter list of the module and must be terminated with a semicolon, as if you were executing an import statement.

Module Summary

In this module, you learned how to

Do Connectivity Changes

- New options make it easier to build hierarchy
- dot-name connection gives best tradeoff on safety, verbosity and readability

Use Compilation Unit Scope

- Definitions external to module
- Rarely (if ever) needed

Use Packages

- Define types and subroutines external to modules
- Definitions reusable throughout a design hierarchy
- Good for system constants, common user-defined types and functions like printing out error messages

100 © Cadence Design Systems, Inc. All rights reserved.



You should restrict your use of the compilation unit scope. The actual scope can be tool-dependent and not necessarily identical for each compilation session.

Use of packages places related declarations in one file that you can separately compile and then import into the design element where needed.

One situation where you might want to use the compilation unit scope is for Verilog-2001 port declarations that use user-defined types. You can import just those type declarations from a package into the compilation unit scope just before the port declarations. Other port declarations would do the same, thus restricting the type definition itself to one package rather than potentially multiple compilation units.

Quiz



Find the problems in the following module top code and correct them.

Import P2 to define mytype2

Redundant

```
import P1::load;
module top;
    import P1::*;
    mytype2 data;
    logic [7:0] count, cnt;
    logic clk, rst, ld, load;
    typedef logic [7:0] mytype1;

    inc U1 (clk, ld, rst, data[7:0], cnt);
    inc U2 (.clk, .ld, .rst, .data, .cnt);
    inc U3 (*.);
    inc U4 (*.*, .count);
    inc U5 (*.*, .data(data[7:0]));
    ...
endmodule
```

Multiple drivers on cnt variable
for U1, U2, U3, and U5

```
package P2;
    typedef logic [15:0] mytype2;
endpackage : P2
```

```
package P1;
    import P2::*;
    typedef enum {one, two} mytype1;
    localparam int load = 10;
endpackage : P1
```

```
module inc (input logic clk, rst, ld,
            input logic [7:0] data,
            output logic [7:0] cnt);
    ...
endmodule
```

rst, ld order swapped

No port count

data size mismatch for
U2, U3 and U4

101 © Cadence Design Systems, Inc. All rights reserved.



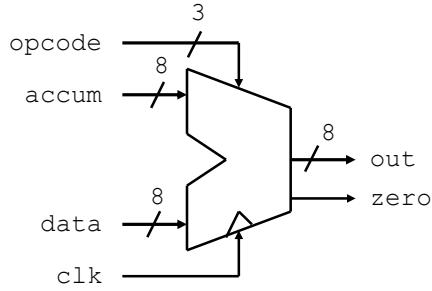
This page does not contain notes.



Lab

Lab 5 Modeling an Arithmetic Logic Unit (ALU) (Optional)

- Model a simple ALU using SystemVerilog constructs including enumerate data types, enumeration methods and packages.



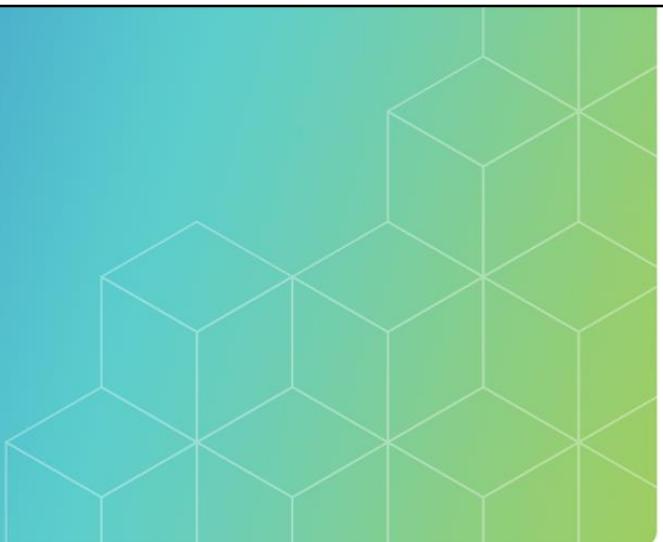
102 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



Static Arrays



Module 8

Revision 1.0
Version 21.10

Estimated time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Module Objectives

In this module, you

- Examine array enhancements in SystemVerilog
- Analyze the difference between packed and unpacked arrays
- Explore array-querying functions



This page does not contain notes.

Arrays



IEEE 1800-2012 7.4

- SystemVerilog allows arrays of any type including:
 - Events, structures, and enumerates
- Standard, Verilog-like arrays are defined as unpacked:
 - Index declared after name
 - Elements stored separately

Assignment patterns can be used to assign unpacked arrays.

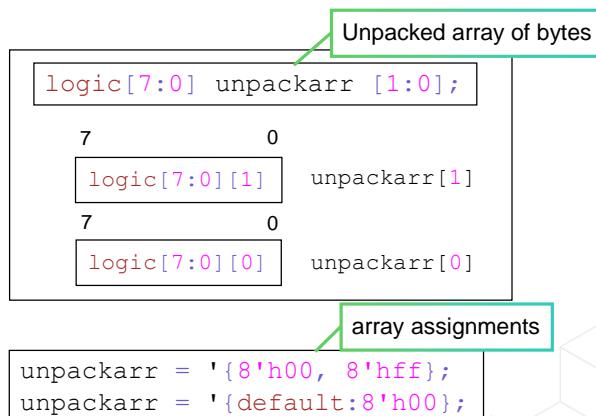
```
int inta [2:0][1:0];
```

2D array of int

```
typedef enum bit [1:0]
  {S0, S1, S2, S3} seq_t;
```

```
seq_t seq5 [4:0];
```

enum array


105 © Cadence Design Systems, Inc. All rights reserved.

Verilog-2001 allows arrays of any number of dimensions of any type except events.

SystemVerilog allows arrays of any number of dimensions of any type, including events.

Traditional Verilog multidimensional arrays are defined as unpacked by SystemVerilog. Unpacked means that each element of the array is stored separately, and operations or accesses over multiple elements are not possible. Unpacked arrays are distinguished by having the index defined after the array identifier.

Traditional Verilog single-dimensional arrays are defined as packed by SystemVerilog. Packed means that the elements of the array are stored as one item, allowing operations or accesses over the full width of the element, as well as operations or accesses on each individual element.

This means that standard vectors, such as `logic [7:0] avec`, are defined as packed.

Pattern assignment is the easiest method to access multiple unpacked array dimensions. You can make pattern assignments by order or by key but not by both in the same assignment. The key can be the element index, and you can also use the default keyword to assign a value to all elements which are otherwise unassigned.

Unpacked Multidimensional Arrays



The term unpacked array is used to refer to the dimensions declared after the data identifier name.

- Unpacked dimensions are stored separately.
- You can read and write any slice from one element to a whole array:
 - Must match size, layout and type.
 - Indexing precedence: left to right.
- Assignment patterns are useful for loading unpacked arrays:
 - Keyed or ordered.
 - Can be nested.
 - Can also contain expressions, for example: repetition.

```

int inta [2:0][0:1];
int intb [1:3][2:1];
int intc [15:0][0:255];
...
inta = intb;
inta[2] = intb[3];
inta = '{'{2,1},'{5,4},'{3,2}}';
// inta[2][0] = 2
// inta[2][1] = 1
// inta[1][0] = 5
...
intb = '{3{'{1,2}});
// intb[1][2] = 1
// intb[1][1] = 2
// intb[2][2] = 1
...
intc = '{default:100};
// intc[0][0] = 100
...
  
```

106 © Cadence Design Systems, Inc. All rights reserved.

cadence®

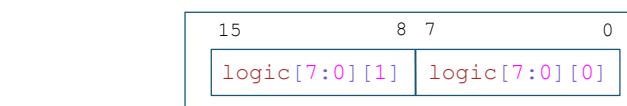
This page does not contain notes.

Packed Arrays



A packed array is a mechanism for subdividing a vector into subfields, which can be conveniently accessed as array elements.

- Packed arrays:
 - Index declared before name
 - Stored as a single vector
 - Can be operated upon and accessed as a single vector
 - Indexing precedence from left to right
- Only single-bit data types can be packed: `bit`, `logic`, `reg`:
 - And recursively, other packed objects of these types
- Packed arrays obey assignment rules for padding and truncation.
- A packed array is equivalent to a vector with predefined bit fields.



Packed array of bytes

```
logic[1:0] [7:0] packarr;
logic[7:0] short = 8'hAA;
logic[23:0] long = 24'h555555;
...
packarr = 16'h0;
packarr = packarr << 1;
packarr = ~packarr;

packarr = short; //16'h00aa
packarr = long; //16'h5555
```

SystemVerilog also has the concept of packed arrays. In a packed array, elements are stored together as a single vector, allowing operations and access to the whole array, as well as individual elements. For example, a packed array can be inverted, shifted, or assigned as a single vector while still having access to the individual elements.

The packed dimensions are declared before the array identifier. Therefore, traditional Verilog one-dimensional vectors are packed.

You can pack only the single-bit types (`bit`, `logic`, `reg`), and recursively, other packed objects (`array`, `struct`) of those types.

A packed array obeys the traditional Verilog rules for truncation and padding when assigned to or from an array of different lengths.

A packed array is essentially just a vector with predefined fields.

You can use the packed array name as an integer in any expression in which an integer may appear. Remember that a packed array is by default unsigned and all multiple bit integer types are by default signed.

Packed arrays are generally synthesizable as they are defined of synthesizable type (`bit`, `logic`, `reg`) and have a defined hardware interpretation (single vector).

Mixed Packed and Unpacked Arrays

- Arrays can have both packed and unpacked dimensions:
 - Like Verilog memories.
 - Any number of dimensions.
- Mixed arrays can be declared with `typedef`:
 - Unpacked dimensions placed after type name.
- Precedence for indexing is like the following:
 - Unpacked dimensions left to right...
 - ..then packed dimensions left to right.
- Generally, only fully packed arrays are synthesizable.

Verilog memory – unpacked array of packed bytes

```
reg [7:0] mem [0:255];
```

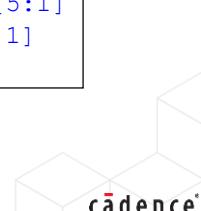
```
typedef logic [7:0] mem_t [0:255];
mem_t mem1;
```

2 3 1

Indexing order

```
bit [3:0] [5:1] v1 [7:0];
// v1      = array of 2D packed arrays
// v1[3]   = 2D packed array[3:0][5:1]
// v1[3][2] = 1D packed array[5:1]
// v1[3][2][1] = scalar bit
```

108 © Cadence Design Systems, Inc. All rights reserved.



SystemVerilog allows arrays to have both packed dimensions (declared before the array identifier) and unpacked dimensions (declared after the array identifier).

Therefore, traditional Verilog memories are interpreted as a one-dimensional unpacked array of one-dimensional packed arrays.

```
reg [7:0] mem [0:255];
```

Elements in the packed dimensions can be accessed either individually or all at once. Elements in the unpacked dimensions can only be accessed individually.

Mixed packed and unpacked array types can be declared with `typedef`, as can purely packed or unpacked arrays. With mixed arrays, the unpacked dimensions are placed after the type name.

When indexing a mixed array, precedence is from left to right through the unpacked dimensions first, followed by a left to right through the packed dimensions.

Generally, only fully packed arrays are synthesizable as they have a direct hardware interpretation (single vector).



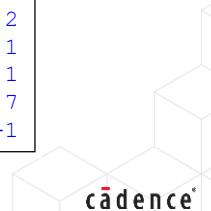
Array Querying System Functions

Multidimension Queries	\$dimensions	Number of dimensions in the array
	\$unpacked_dimensions	Number of unpacked dimensions in array
Single Dimension Queries	\$left	Left bound
	\$right	Right bound
	\$low	Lowest bound
	\$high	Highest bound
	\$increment	1 if \$left >= \$right, else -1
	\$size	Number of elements

These methods return an `int` value, and are applicable to all arrays.

```
typedef logic [7:0] mem_t [1:1024];
mem_t mem ;
...
$display($dimensions(mem));           // 2
$display($unpacked_dimensions(mem_t)); // 1
$display($left(mem));                // 1
$display($high(mem_t,2));            // 7
$display($increment(mem));           // -1
```

109 © Cadence Design Systems, Inc. All rights reserved.



Array querying system functions return an `int` value and apply to all array types, including fixed-size integer types (equivalent to one-dimensional packed arrays). You can apply them to any array identifier and to any static array type.

Syntax:

`$dimensions | $unpacked_dimensions (identifier | type)`

Array dimension system functions (from slowest varying = 1)

`<function> (identifier | type [, dimension_expr=1])`

`<function> one of { $left $right $low $high $increment $size }`

- `$dimensions` returns the number of dimensions in the array: 1 for strings and simple bit vectors or 0 for any other type.
- `$unpacked_dimensions` returns the number of unpacked dimensions of an array or 0 for any other type.
- `$left` returns the left declared bound of the dimension.
- `$right` returns the right declared bound of the dimension.
- `$low` returns the lowest declared bound of the dimension.
- `$high` returns the highest declared bound of the dimension.
- `$increment` returns 1 if the left bound is greater than or equal to the right bound and -1 otherwise.
- `$size` returns the number of elements in the dimension.

Dimensions are numbered from number 1 in the order of index precedence.

Quiz



What are the values of the following?

```
logic [3:0][7:0] parr;  
...  
parr[0] = 6;  
parr[1] = 7;  
parr[2] = 8;  
parr[3] = 9;  
...
```

```
typedef enum bit [1:0] {A, B, C, D} mytype;  
  
mytype [6:0] pnum; // packed enum array  
...  
// packed array concatenation assignment  
pnum = {D, D, C, C, B, B, A};  
...
```

What is the hex value of:

parr	:	09080706
parr[0]	:	6
parr[0][5]	:	0

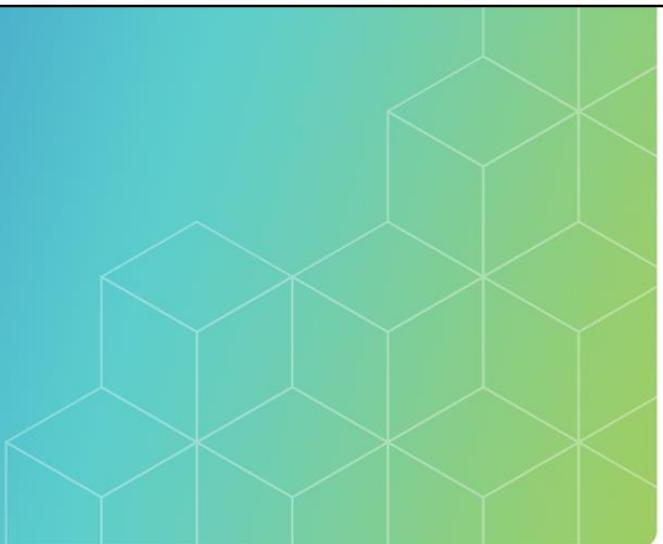
What are the hex and enum (where applicable) values of:

pnum	:	3e94, DDCCBBA
pnum[2]	:	1, B
pnum[4:3]	:	a CC
pnum[5][1]	:	1 n/a

This page does not contain notes.



Tasks and Functions



Module **9**

Revision **1.0**
Version **21.10**

Estimated time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Module Objectives

In this module, you

- Create functions with output arguments
- Use void functions
- Apply default argument values and named argument passing
- Review the issues with the default "pass-by-value" argument operation
- Examine the SystemVerilog "pass-by-reference" option for arguments



This page does not contain notes.



Verilog-2001 Tasks and Functions Review

Tasks

- Any number of input, output or inout arguments – can be ANSI C style
- May contain timing: @(...), #delay, wait

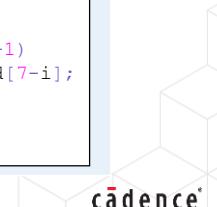
```
task write_mem (
    input [7:0] waddr, wdata,
    output [7:0] indata, addr);
begin
    indata <= wdata;
    addr <= waddr;
    write <= 1'b0;
    #5 write <= 1'b1;
    #10 write <= 1'b0;
end
endtask
```

Functions

- One or more input arguments only – can be ANSI C style
- No timing allowed – function must complete in zero time
- Returns single value – vector, real or integer

```
function [7:0] flip(
    input [7:0] word);
    reg [7:0] tempword;
    integer i;
begin
    for (i=7; i>=0; i=i-1)
        tempword[i] = word[7-i];
    flip = tempword;
end
endfunction
```

113 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



Verilog-2001 Static and Automatic Subroutines

Verilog-1995 subroutines are static:

- Only one copy exists.
- Concurrent calls can overwrite each other's arguments and variables.

Verilog-2001 added automatic subroutines:

- New copy for every call.
- Concurrent calls each have their own set of arguments and variables.
- Argument assignment must be blocking.

```
function automatic [63:0] factorial;
  input [7:0] n;
begin
  if (n == 1)
    factorial = 1;
  else
    factorial = n * factorial(n-1);
end
endfunction
```

Verilog-2001

```
task automatic count_clocks;
  input [31:0] edges;
begin
  repeat(edges)
    @(posedge clk);
end
endtask

initial
begin
  count_clocks(10);
  ...
end

always @(posedge trig)
begin
  count_clocks(6);
  ...
end
```

Verilog-2001

114 © Cadence Design Systems, Inc. All rights reserved.

cadence

This page does not contain notes.

SystemVerilog Optional begin / end and Named Ends



IEEE 1800-2012 13.3

- You can omit the `begin` and `end` keywords.
- You can repeat the subroutine name with `endtask` or `endfunction`.
 - SystemVerilog verifies that it is the same name.

```
function [7:0] flip
    (input [7:0] word);
    logic [7:0] tempword;
//begin
    for (int i=0;i<=7;++i)
        tempword[i] = word[7-i];
    flip = tempword;
//end
endfunction : flip
```

SystemVerilog

```
task write_mem (
    input [7:0] waddr, wdata,
    output [7:0] indata, addr);
//begin
    indata <= wdata;
    addr <= waddr;
    write <= 1'b0;
#5 write <= 1'b1;
#10 write <= 1'b0;
//end
endtask : write_mem
```

SystemVerilog

115 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Void Functions



IEEE 1800-2012 7.4

- You can declare functions that do not return a value.
 - Define return type as `void`.
- You call a void function as a statement:
 - All restrictions of functions still apply.
- `void` can also be used to discard a function return value:
 - By casting the return type to `void`.

```
function void printerr (input int errs);
  if (errs == 0)
    $display ("No Errors");
  else
    $display ("%0d Errors", errs);
endfunction
...
initial begin
  ...
  printerr(status);
  ...

```

```
function bit myfunct(...);
  ...
endfunction
...
initial begin
  void' (myfunct(...));
  ...

```

116 © Cadence Design Systems, Inc. All rights reserved.



In Verilog, a function must return a value even if this value is redundant or superfluous. SystemVerilog allows a function not to return a value by declaring the return type of the function as `void`. Such a function is called a void function.

As the function no longer returns a value, it cannot be called as part of an expression or assignment. Therefore, a void function is called as a statement, like a task. Apart from this difference, a void function follows all the normal rules for a SystemVerilog function.

`void` can also be used in another capacity with functions. If you call a function which does have a return value, and you wish to ignore this value, you can cast the return value to the `void` type.

```
void' ( nonVoidFunctionCall(...) )
```

This allows a function which does return a value to be called as a statement.

Voiding a function return value needs to be used with care, because a return value usually contains important information, such as the success or failure of an operation, which should be checked.

Function Output Arguments



IEEE 1800-2012 13.4

- You can provide `input`, `output`, and `inout` function arguments.
 - Argument default is `input` of type `logic`.
 - `output/inout` arguments let functions return multiple values.
 - `output/inout` arguments let `void` functions return values.
- Function becomes a general-purpose synthesizable subroutine.

```
function [7:0] addcarry (input [7:0] a, b,
                        output carry);
    {carry, addcarry} = a + b;
endfunction
```

```
logic [7:0] a, b, sum;
logic cry;

initial begin
    sum = addcarry(a, b, cry);
    ...

```

`cry` is an output

```
// a and b default to inputs
function void add (integer a, b,
                    output integer sum);
    sum = a + b;
endfunction

always @ (a or b)
    // void function call
    add (a, b, sum);
```



117 © Cadence Design Systems, Inc. All rights reserved.

In SystemVerilog, a function can have `output` and `inout` arguments as well as `input`. Output arguments are passed out when the function returns.

To remain compatible with Verilog, an undefined argument direction defaults to `input` and an undefined argument type defaults to `logic`.

Output arguments allow nonvoid functions to return more than one value – the return type of the function plus any output arguments. In the examples above, the function `addcarry` returns two values, one by the output argument `carry` and the other via the function name `addcarry`.

Output arguments also allow `void` functions to return values.

You can use functions with `output` or `inout` arguments only within procedural blocks and not in continuous assignments.

The purpose of these changes is to remove the one key advantage of a task over a function – the ability to return multiple output arguments. Now functions can return multiple values; there is no longer any reason to use tasks in synthesizable code.

Default Argument Values

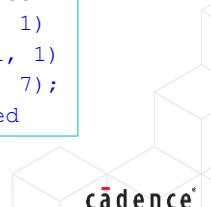
- You can specify default values for subroutine arguments.
 - SystemVerilog uses default values for those not passed – with rules!
 - Must have unambiguous mapping of actuals to formals.
 - It is an error for a formal parameter to **not** have an unambiguous value.

Error:
 'j' gets value
 of 2 and 'k' is not
 defined

```
// default argument values in task definition
task read (int j = 0, int k, int data = 1);
. . .
endtask

int val = 21;

// invocation of task with default arguments
read ( , 5);      // equivalent to (0, 5, 1)
read (2, val);    // equivalent to (2, 21, 1)
read ( , 5, 7);   // equivalent to (0, 5, 7)
read (2);         // ERROR - k not defined
```



Subroutines can have large numbers of input arguments, and some of these arguments may be mapped to the same literal value for the majority of calls. Consider, for example, a debug switch on a task, which is usually disabled, but when enabled reports additional information about the task execution. In such cases, it may be convenient to define default values for these arguments, allowing the argument to be omitted in the task call when the default is used, but included in the call if the default value must be overridden.

SystemVerilog allows a subroutine declaration to specify a default value for each singular argument that is not an output port. You can use default values only with ANSI-C-style port declarations. When you call the subroutine, you can omit any arguments that have default values. If necessary, you can use commas as placeholders for default arguments, although this is not very readable.

When actual arguments are mapped to formals in the call, every input or inout argument must have an unambiguous value, either through an explicit mapping or a default value.

Argument Binding by Name

- You can bind subroutine arguments by name instead of by position.
 - Similar to port connection by name.
 - Simplifies calling subroutines having many arguments.
 - Simplifies calling subroutines having default arguments.

```
task read (int j = 0, int k, int data = 1);
. . .
endtask

int val = 21;

// invocation of task with default arguments and name passing
read (.j(4), .k(val), .data(7));
read (.j(2), .k(val));           // equivalent to (2, 21, 1)
read (.k(3));                  // equivalent to (0, 3, 1)
read (.data(7), .j(4), .k(3));
```



SystemVerilog allows you to bind subroutine argument actuals to their formals by name, similar to Verilog named port mapping in module instantiations. This is especially useful with default values, to avoid using commas as position placeholders.

Optional Argument List

- You can omit passing arguments if all arguments have default values.
- If there are no required arguments, you can even omit the parentheses for:
 - Tasks calls.
 - Void function calls.
 - Non-recursive calls to class methods.

```
// default value for every argument
task read (int j = 0, int k = 2, int data = 1);
    ...
endtask

// invocation of task with default arguments
read();    // equivalent to (0, 2, 1)
read;      // also legal in SV
```



If you are calling a void function or class function method that has no arguments, or calling a task, void function or class method for which all arguments have default values, SystemVerilog allows the parentheses for the call to be omitted. For a directly recursive nonvoid function method call, you need to either include the parentheses or hierarchically qualify the function called. This is to differentiate between the recursive function call and the function name variable.

Explicit Return: `return`



IEEE 1800-2012 13.4.1

- Executing a `return` statement immediately exits a subroutine.
 - Allows easier structuring of subroutines
- In a function, you can include an expression to return a function value.

```
function integer mult (input integer num1, num2);
    if ((num1 == 0) || (num2 == 0)) begin
        $display("Zero multiply");
        return 0;
    end
    else
        mult = num1*num2;
endfunction
```

```
task printstatus (input int errs);
    if (errs == 0) begin
        $display("Zero Errors");
        return;
    end
    else begin
        $display("%d Errors", errs);
        case (errs)
            ...
        endcase
    end
endtask
```

121 © Cadence Design Systems, Inc. All rights reserved.



SystemVerilog adds an exit explicitly for subroutines using the `return` statement. For a function, the `return` statement can provide an explicit return value function, and this value overrides any other value previously assigned to the function name.

If the subroutine contains output or inout arguments, you should remember to assign these before executing the `return` statement. Unassigned output or inout arguments will have either default initialization values, or for static subroutines, values from any previous calls.

Argument Passing by Value



For the code shown here, will this work as expected?

- This is the default method of argument passing.
- It is the same as in Verilog-2001, but can also pass new data types: arrays, structures, unions, classes, etc.
- Values copied *in* on call:
 - Subroutine then unaware of input changes.
- Values copied *out* on return:
 - Environment meanwhile unaware of output changes.
- Issue for testbench tasks.

Formal arguments mapped to actuals in call

```

logic req;
logic ack;
logic [7:0] data;

task cpu_drive_bad
    (input logic [7:0] write_data,
     input logic data_valid,
     output logic data_read,
     output logic [7:0] cpu_data);

#5 data_read = 1;
wait(data_valid == 1);
#20 cpu_data = write_data;
wait(data_valid == 0);
#20 cpu_data = 8'hzz;
data_read = 0;

endtask : cpu_drive_bad
...
cpu_drive_bad(8'hff, ack, req, data);
...

```

Input changes not detected

Only last output updates seen

122 © Cadence Design Systems, Inc. All rights reserved.

cadence

SystemVerilog allows any new or user-defined types as argument types.

For argument passing, Verilog supports only the pass-by-value mechanism:

- Inputs are sampled and passed into the subroutine at the start of the call.
 - Calls to a static subroutine all share the same local copy.
 - Calls to an automatic subroutine each make their own local copy.
- Outputs are passed out of the subroutine at the end of the call.
- Changes to input arguments are not seen inside the subroutine during the lifetime of the call.
- Changes to output arguments are not seen outside the subroutine during the lifetime of the call.

In the example above, the assignment of `data_read` to 1 is not seen on `req` until the end of the call. Likewise, any changes in `ack` are not seen on `data_valid` after the beginning of the call. The task will hang on one of the two `wait` statements depending on the value of `ack` sampled at the beginning of the call.

Solution 1: Variable Access by Side-Effect

- This is a traditional solution to the argument passing issue:
 - Task directly reads/writes variables.
 - Only literals are passed as arguments.
- Variables must be of same scope as task declaration:
 - Task cannot be declared in a package.
 - Reuse of task is limited.
- It is used extensively for class methods:
 - See later in presentation.

```

logic req;
logic ack;
logic [7:0] data;

task cpu_drive_side
  (input logic [7:0] write_data);

  #5 req = 1;
  wait(ack == 1);
  #20 data = write_data;
  wait(ack == 0);
  #20 data = 8'hzz;
  req = 0;
endtask : cpu_drive_side
...
cpu_drive_side(8'hff);
...

```

123 © Cadence Design Systems, Inc. All rights reserved.



The traditional solution to the argument passing issue is to use side-effects. Here the variables driven and read by the task are not passed as arguments, but are accessed directly. Only literals (in this case `write_data`) are passed as arguments. The variables accessed by side-effects must be declared in the same scope as the task declaration. These are not necessarily the variables from the scope of the task call.

There are some restrictions to this solution.

- The task cannot be reused for a different set of such nets or variables than those used in the declaration. This can lead to duplication.
- The task must be declared local to the variables it accesses, preventing the task from being declared in a package, for example.

These restrictions can be avoided through the use of interfaces (see later). The side-effect approach is also extensively used in class methods, where subroutines are declared local to the variables (or class properties) they access (see later).

Solution 2: Argument Passing by Reference

- Use pass by reference by declaring arguments using `ref`:
 - Use `ref` instead of parameter direction.
 - Only variables not nets.
- It creates a link between actual and formal arguments:
 - Changes in inputs can be seen in task.
 - Changes in outputs are immediately updated.
- Task must be declared as automatic.

```
task automatic cpu_driver_ref
  (input logic [7:0] write_data,
   ref logic data_read,
   ref logic data_valid,
   ref logic [7:0] cpu_data);
  #5 data_valid = 1;
  wait (data_read == 1);
  #20 cpu_data = write_data;
  wait (data_read == 0);
  #20 cpu_data = 8'hzz;
  data_valid = 0;
endtask : cpu_driver_ref
...
cpu_drive_ref(8'hff, req, ack, data);
```

All output updates seen

Input changes detected

Formal arguments mapped to actuals in call

SystemVerilog supports pass-by-reference (`ref`) for variable arguments. `ref` cannot be used for net arguments – they can only be passed by value.

Pass by reference passes a handle to the actual argument object rather than its value. The subroutine directly accesses the variable whose reference is passed. Changes to input arguments are immediately seen inside the subroutine call. Changes to output arguments are immediately seen outside the subroutine call.

`ref` arguments can only be used with automatic subroutines.

Using 'const' with Reference Arguments



IEEE 1800-2012 13.5.2

- It can be used only with automatic (not static) subroutines:
 - Risk of outdated reference.
- Without direction, arguments may be written in error:
 - Use `const ref` to prevent the subroutine from modifying argument.
- It is also useful for efficiency gains:
 - When arguments occupy large amounts of memory.

```
task automatic create_crc
  (ref logic [255:0] payload [1:0]
  ...
  ...
  payload = '0;
```

Should the data be modified?

```
task automatic create_crc
  (const ref logic [255:0] payload [1:0]
  ...
  ...
  payload = '0;
```

`const argument`
cannot be modified



125 © Cadence Design Systems, Inc. All rights reserved.



You can hierarchically access the arguments of a static subroutine throughout the simulation. The static reference would be valid only while the referenced variable exists. For this reason, SystemVerilog does not permit passing static arguments by reference. You must in general be careful not to use a reference that might be invalid, for example, in an automatic task that consumes time and returns after the calling process goes out of scope.

With the use of the `ref` keyword, the direction information of the argument is lost. This may lead to inadvertently updating arguments meant only as inputs. To prevent an argument from being updated, declare it as `const ref`.

When the arguments are large, it can be undesirable to pass arguments by value. *Pass by value* creates multiple copies of the same data, and is thus inefficient. Passing large arguments by reference does not create additional copies every time the subroutine is called. This can be much more efficient.

Module Summary

In this module, you learned how to

- Use following features that describe more functionality with less code:
 - return statements
 - Default argument values
- Apply below features for reduced potential syntax and functional errors:
 - Function output arguments
 - Argument passing by name
 - Argument passing by reference
- Use more efficient subroutine implementation:
 - Passing large or complex arguments by reference (`ref`) instead of value should reduce memory usage
- Use `void` functions with output arguments inside synthesizable code



This page does not contain notes.

Quiz



What is the type of the function that returns no data?

void



What is the default type of a function output argument?

logic



Under what circumstances can you call a subroutine and provide no arguments?

If all arguments have a default value



Compare subroutine argument pass-by-value and pass-by-reference.

pass-by-value – Copies value upon invocation and return

pass-by-reference – Subroutine accesses original object



How do you prevent a subroutine from writing an argument passed by reference?

Declare the formal parameter to be *const ref*



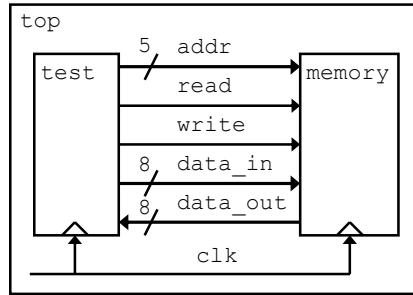
This page does not contain notes.



Lab

Lab 6 Testing a Memory Module

- Model stimulus tasks to verify a memory module using subroutine enhancements.



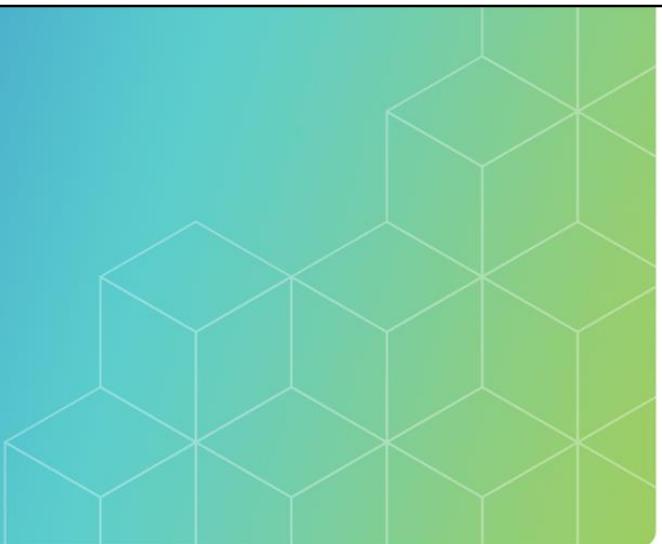
128 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



Interfaces



Module **10**

Revision

1.0

Version

21.10

Estimated time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Module Objectives

In this module, you

- Investigate the benefits and capabilities of interfaces
- Create simple interfaces
- Refine simple interfaces with ports and parameters
- Define variable direction and visibility with *modports*
- Define interface methods – tasks/functions declared in interfaces



This page does not contain notes.

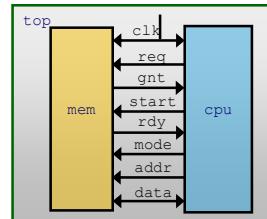
Verilog Hierarchical Connections



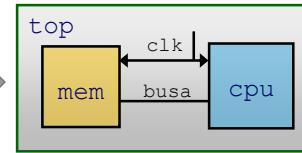
One Verilog hierarchical connection requires 5 declarations:

- Two port declarations in modules `mem` and `cpu`
- Signal declaration in `top`
- Signal added to each instantiation of `mem` and `cpu`

Problem: Creating and maintaining multiple connections is tedious.



Using Interface



```
module memory(
    input logic clk, req, start,
    logic [1:0] mode,
    logic [7:0] addr,
    inout wire [7:0] data,
    output logic gnt, rdy);
    ...
endmodule
```

```
module cpucore(
    input logic clk, gnt, rdy,
    inout wire [7:0] data,
    output logic req, start,
    logic [7:0] addr,
    logic [1:0] mode);
    ...
endmodule
```

```
module top;
    logic req, gnt, start, rdy;
    logic clk = 0;
    logic [1:0] mode;
    logic [7:0] addr;
    wire [7:0] data;

    memory mem(.clk, .req, .start,
               .mode, .addr, .data, .gnt, .rdy);

    cpucore cpu(.clk, .gnt, .rdy, .data,
                .req, .start, .addr, .mode);
    ...

```

131 © Cadence Design Systems, Inc. All rights reserved.



To add a single connection between two modules in one level of hierarchy, you:

- Add a port to each module.
- Declare a net or variable in the parent module.
- Map the net or variable to the port of each module instance.

A standard communications bus may be composed of many signals, which may be connected to multiple modules. Creating and maintaining such a bus connection can be difficult in Verilog.

SystemVerilog supports interfaces. In its simplest form, an interface is a separately declared and named group of signals. All connections associated with a specific interface are declared and maintained in one place.

More complex interfaces can have their own ports through which they can share external nets and variables with the connected modules. They can be parameterized to allow, for example, different bus widths, and can contain subroutines to convert abstract data communications to low-level signal transitions.

Declaration of the bus ports and signals is time consuming and error prone. Maintaining bus connections is also difficult, particularly if the bus is connected to many separate modules. Modifying one signal connection requires you to:

- Modify the port list of every module that connects to the bus.
- Modify every instantiation of the changed modules.
- Modify signal declarations in the parent module.

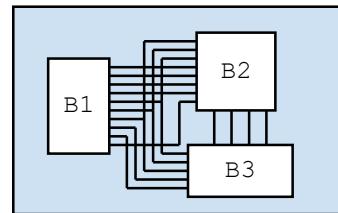
A typical module may have many port connections, representing perhaps several buses. In such cases, it can be difficult to remember which ports belong to which bus.

In this example, a connection between the `memory` and `cpucore` modules is composed of the `req`, `gnt`, `start`, `rdy`, `mode`, `addr` and `data` signals. Each of these connections is declared in the `memory`, `cpucore` and `top` modules.

What Is an Interface?

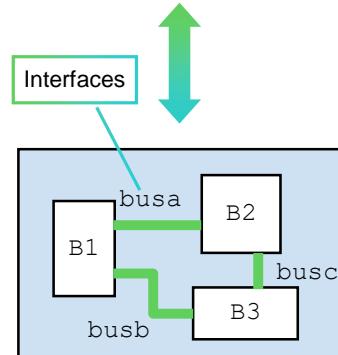


A SystemVerilog interface is, at its most basic level, simply a bundle of external nets and/or variables normally shared by one or more modules by virtue of port connections.

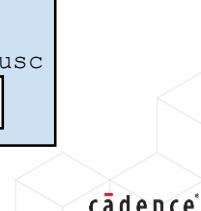


SystemVerilog provides the interface construct to:

- Encapsulate communication between hardware blocks.
- Provide a mechanism for grouping together multiple signals into a single unit that can be passed around the design hierarchy.
- Enable abstraction in RTL design.

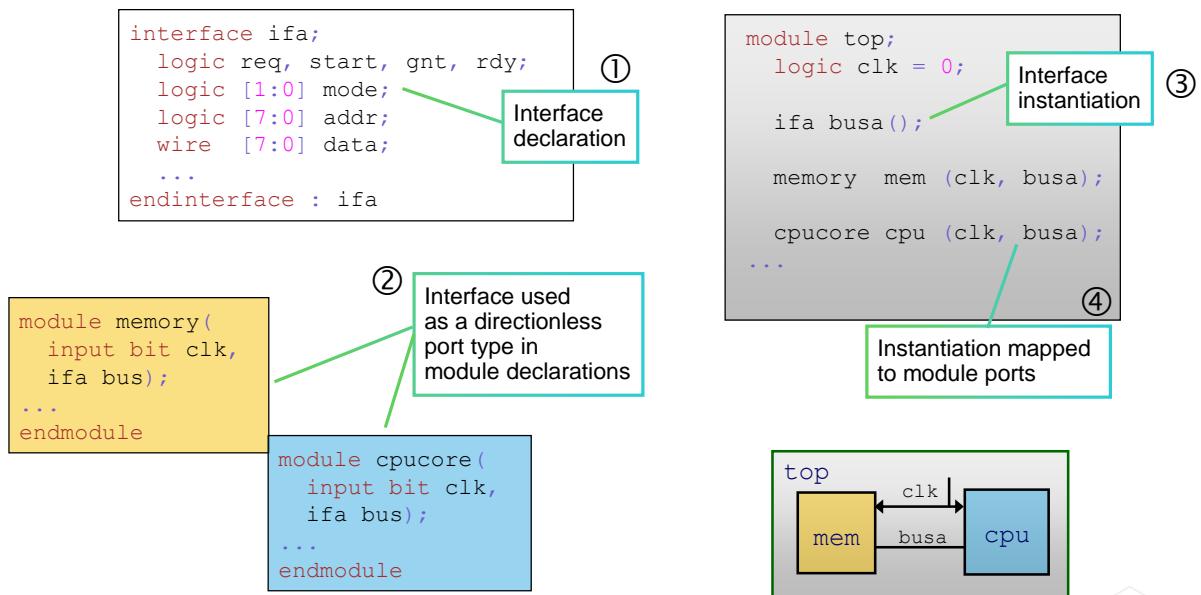


132 © Cadence Design Systems, Inc. All rights reserved.



An interface is a new kind of design block that captures inter-module communication in one place. Rather than declare many ports and signals at each hierarchy level and connect them, you can declare the signals in one interface and declare ports of that interface type, which when connected to the interface, automatically make all those individual signal connections for you.

Example with Interfaces



133 © Cadence Design Systems, Inc. All rights reserved.



The previous example can be created using interfaces as follows:

1. Define all the bus signals in an interface. This is a module-like design element. The interface is declared in a separate file and must be compiled separately.
2. The interface name becomes the type for an interface port in the module declaration. This port will contain all the signals defined in the interface. The port does not have a direction defined as it contains both `input`, `output` and potentially `inout` signals (more later).
3. Instantiate the interface (just like a module instantiation) in the module which also contains the module instances to be connected. Then in the module instantiation port maps, the interface instance is mapped to the interface ports to connect the modules.

This example defines the `ifa` interface containing `req`, `gnt`, `start`, `rdy`, `mode`, `addr` and `data` signals. The `memory` module and `cpucore` modules each have an interface port of type `ifa` called `bus`. The parent module (`top`) instantiates the `ifa` interface and the `memory` and `cpucore` modules. In the module instantiations, the `bus` interface ports of `memory` and `cpucore` are mapped to the `busa` instance of the `ifa` interface.

If the signals of the bus change, then only the interface declaration needs to be modified, and all the module connections are then updated.



Quick Reference Guide: More Facts on Interfaces

You declare an interface as a hierarchical object, much like a module:

- You instantiate it in a module, like any other hierarchical block.
- You use the interface like a type in port lists and port maps.

A simple interface is just a named bundle of nets or variables:

- Similar to a `struct` type.
- You can reference the nets or variables where needed.

Interfaces can also contain module-like features for defining signal relationships:

- Continuous assignments, tasks, functions, initial/always blocks, etc.
- Can further instantiate interfaces.
- Can be generated or arrayed.
- Cannot declare or instantiate module-specific items:
 - Modules, primitives, specify blocks, and configurations.



A SystemVerilog interface is declared as a design element like a module. It is usually placed in a separate file and compiled separately by the simulator. It can be instantiated in a module, like a module instantiation, but the interface name is also used as port type in module declarations to create interface ports. Interface ports are connected using interface instantiations.

A SystemVerilog interface is, at its most basic level, simply a bundle of external nets and/or variables normally shared by one or more modules by virtue of port connections. The bundling of these nets and variables into an interface permits the modules each to connect one interface port to the entire bundle, rather than making individual connections for every port.

Interfaces can also contain much more complex features, such as assign statements, subroutines, procedural blocks, assertions etc. Interfaces cannot contain other design element declarations, such as modules, configurations or packages, nor can they contain module-specific constructs such as *specify* blocks.

Interfaces can instantiate other interfaces to create nested interface structures, and interfaces can be created using Verilog generate statements or in arrays.

How to

Bind Interface Instance to Module Ports of Interface Type



SystemVerilog port mapping rules also apply to interfaces.

1. You can map by position or name.
2. You can use dot-name (.name) mapping if instance name matches port name.
3. You can use dot-star (*.*) mapping if all names match.

```
module modone (
    input bit clk,
    ifa busa);
    ...

```

```
module modtwo (
    input bit clk,
    ifa busb);
    ...

```

```
module top;
    logic clk = 0;                                Creating Interface Instances
    ifa oneif(), twoif(), busa(), busb();

    modone mem_pos(clk, oneif);
    modone mem_nam(.clk(clk), .busa(twoif));
    modone mem_dotnam(.clk, .busa);
    modtwo mem_dotstr(.*);                         Binding Interface Instances
    ...

```

Positional
Named
dot-name
dot-star

135 © Cadence Design Systems, Inc. All rights reserved.



The port map syntax for ports of an interface type is identical to that for ports of any other type. You can use ordered connections, named connections, dot-name (.name) connections, and dot-star (*.*) connections.

How to

Access Interface Items Through Module Ports Bound to the Interface Instance



For a module with an interface *port*, you access interface items using the port name as a prefix.

```
interface ifa;
  logic req, start, gnt, rdy;
  logic [1:0] mode;
  logic [7:0] addr;
  wire [7:0] data;
  ...
endinterface : ifa
```

```
module memory ( input bit clk, ifa bus );
  reg [31:0] mem [0:31];
  logic read, write;
  assign read = (bus.gnt && (bus.mode == 0) );
  assign write = (bus.gnt && (bus.mode == 1) );
  always @ (posedge clk)
    if (write)
      mem[bus.addr] = bus.data;
    assign bus.data = read ? mem[bus.addr] : 'z;
endmodule
```

1. Pass the Interface as port bus.

2. Access the Interface item using module port name.

In a module that has a port of an interface type, you can reference the contents of the interface by the port name (`port_name.interface_signal_name`).

How to

Access Interface Items Through the Interface Instance Identifier



For a module with an interface *instance*, you access interface items using the instance name as a prefix.

```
module top;
    logic clk;

    ifa busa();
    memory mem (clk, busa);
    cpucore cpu (clk, busa);
    ...
    always @ (busa.rdy)
        if (busa.mode == 0)
            ...
endmodule : top
```

```
interface ifa;
    logic req, start, gnt, rdy;
    logic [1:0] mode;
    logic [7:0] addr;
    wire [7:0] data;
    ...
endinterface : ifa
```

1. Declare Interface instance busa.

2. Access Interface signal using instance name.

From within a module which has an instance of an interface type, you reference the contents of the interface via the instance name (`instance_name.interface_signal_name`).

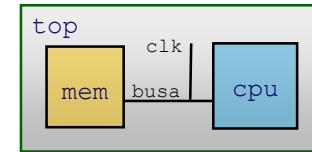
How to

Define Interface Ports



An interface can have its own ports:

- Connected like any module port.
- Used to share an external signal.



```
interface ifa (input clk);
  logic req, start, gnt, rdy;
  logic [1:0] mode;
  logic [7:0] addr;
  wire [7:0] data;
endinterface : ifa
```

```
module memory( ifa bus );
  ...
endmodule
```

```
module cpucore( ifa bus );
  ...
endmodule
```

1. Define 'clk' port in Interface declaration.

2. Connect 'clk' port of 'top' module to Interface port during instantiation.

```
module top;
  logic clk = 0;

  ifa busa(clk);

  memory mem (busa);
  cpucore cpu (busa);
  ...

```

138 © Cadence Design Systems, Inc. All rights reserved.



Interfaces can contain ports similar to modules. Ports of the interface are visible to modules with ports of the interface type as if the interface ports were signals declared within the interface itself.

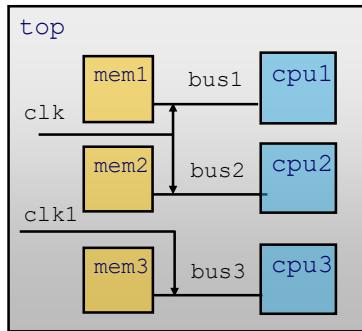
When an interface with ports is instantiated in a module, the ports must be connected to local signals of the interface, just like a module instantiation.

This allows an interface to include a signal declared (and driven) external to the interface or to map different signals into instances of the same interface.

This example adds the `clk` port to the `ifa` interface definition. The `memory` module and `cpucore` module no longer require separate `clk` ports, as they are now part of the interface. The module `top` maps its local `clk` signal to the `clk` port of the interface when it instantiates the interface. The modules can access this interface signal the same way they access any other interface signal.

Example: Interface Port Applications

- Common signals for instances of different interface definitions.
- Different signals for different instances of the same interface definition.



```

interface pds_if (input clk);
    logic [7:0] address;
    wire [7:0] data;
endinterface

interface hbus_if (input clk);
    logic [7:0] address;
    logic [7:0] data;
endinterface

module top;
    logic clk, clk1;

    pds_if bus1(clk);
    mem_pds mem1 (bus1);
    cpu_pds cpu1 (bus1);

    hbus_if bus2(clk);
    mem_hbus mem2 (bus2);
    cpu_hbus cpu2 (bus2);

    hbus_if bus3(clk1);
    mem_hbus mem3 (bus3);
    cpu_hbus cpu3 (bus3);
endmodule : top

```

139 © Cadence Design Systems, Inc. All rights reserved.



Connections from mem1 to cpu1, and from mem2 to cpu2 use different interface types (pds_if and hbus_if), but must be synchronized to the same clock signal. In module top, we pass the same signal clk to the input port of both the bus1 and bus2 interface instances.

The connection from mem3 to cpu3 uses the same interface type as that between mem2 and cpu2, but a completely different clock signal (clk1) is mapped to the interface clock port for the bus3 interface instance.

How to

Define Interface Parameters



You can parameterize interface just like a module.

```
interface fastbus #(DBUS = 32, ABUS = 8) (input clk);
    logic [DBUS-1:0] data;
    logic [ABUS-1:0] address;
    ...
endinterface
```

1. You can define them as part of interface header.

2. You can also define them inside interface with 'parameter' type.

```
interface slowbus;
    parameter WIDTH = 16;
    logic [WIDTH-1:0] a, b;
    ...
endinterface
```

```
module test;
    fastbus #(8, 5) bus8x5(clk); // 8-bit DBUS and 5-bit ABUS
    fastbus #(8) bus8x8(clk); // 8-bit DBUS and 8-bit ABUS
    slowbus #(.WIDTH(8)) bus2(); // 8-bit a, b variables
    ....
```

3. You can override parameters during interface instantiation.

This page does not contain notes.

What Is modport?



To restrict interface access within a module, there are **modport** lists with directions declared within the interface. The keyword **modport** indicates that the directions are declared as if inside the module.

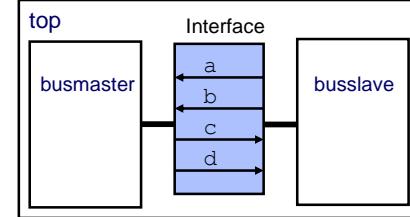
Modports create different views of an interface.

- Specify a subset of interface signals accessible to a module.
- Specify direction information for those signals.

You can specify a modport view for a specific module in two ways:

1. In the module declaration.
2. In the module instantiation.

```
interface mod_if;
  logic a, b, c, d;
  modport master (input a,b, output c,d);
  modport slave (output a,b, input c,d);
  modport subset (output a, input b);
endinterface
```



141 © Cadence Design Systems, Inc. All rights reserved.

cadence®

A simple interface does not contain any direction information for the interface signals. Therefore, a module to which the interface is connected could read or write any interface signal. Typically, a bus connection between two modules will have a direction associated with it. There will be a transmitter or master to send data over the connection and a receiver or slave to read the information. This gives us two views of the bus connection. You can define these views using the **modport** keyword.

An interface can have any number of modports, and each defines a different view of the interface contents. A modport can restrict how connected modules use the interface contents by defining a subset of the interface signals.

The interface `mod_if` declares the following modports:

- Modport `master` which defines signals `a` and `b` as input and signals `c` and `d` as output.
- Modport `slave` which defines signals `a` and `b` as output and signals `c` and `d` as input.
- Modport `subset` which defines signal `a` as output and `b` as input. Any connections to the interface via modport `subset` would not be able to access signals `c` or `d`.

A module can specify which modport to use in its port list declaration or a parent module can specify a modport when making the port connection.

Selecting the Interface Modport by Qualifying the Module Port Interface Type



An interface modport can be selected using the module declaration port of interface type.

1. busmaster declares interface port `mbus`:

- Type is `mod_if`
- Modport is `master`

2. busslave declares interface port `sbus`:

- Type is `mod_if`
- Modport is `slave`

3. testbench instantiates interface and modules as before.

```
interface mod_if;
  logic a, b, c, d;
  modport master (input a,b, output c,d);
  modport slave (output a,b, input c,d);
endinterface
```

```
module busmaster (mod_if.master mbus);
  module busslave (mod_if.slave sbus);
    ...
  endmodule
```

```
module testbench;
  mod_if busa();
  busmaster M1 (.mbus(busa));
  busslave S1 (.sbus(busa));
  ...
endmodule
```

142 © Cadence Design Systems, Inc. All rights reserved.



This example defines an interface `mod_if` with `master` and `slave` modports.

- The `busmaster` module declares an interface port `mbus` of type `mod_if` and linked to the `master` modport of `mod_if`.
- The `busslave` module declares an interface port `sbus` of type `mod_if` and linked to the `slave` modport of `mod_if`.
- The `testbench` module instantiates the interface and maps it to the ports of the `busmaster` and `busslave` instances.

This modport selection method works well when all instances of a module use an interface in the same way.

Selecting the Interface Modport by Qualifying the Module Port Interface Binding



An interface modport can be selected during the port mapping of module Instantiation.

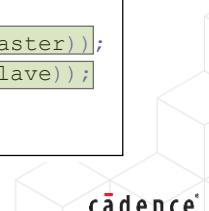
1. busmaster declares interface port `mbus` of type `mod_if`.
2. busslave declares interface port `sbus` of type `mod_if`.
3. testbench instantiates `mod_if` instance `busb`.
4. testbench maps the `mbus` port of `M1` to `modport master` of `busb`.
5. testbench maps the `sbus` port of `S1` to `modport slave` of `busb`.

```
interface mod_if;
  logic a, b, c, d;
  modport master (input a,b, output c,d);
  modport slave (output a,b, input c,d);
endinterface
```

```
module busmaster (mod_if mbus);
  ...
endmodule
```

```
module testbench;
  mod_if busb();
  busmaster M1 (.mbus(busb.master));
  busslave S1 (.sbus(busb.slave));
  ...
endmodule
```

143 © Cadence Design Systems, Inc. All rights reserved.



In this example, the module top specifies which modport the module interface ports will use.

- In the instantiation `M1` of `busmaster`, the interface port `mbus` is mapped to the `master` modport of the interface instance `busb`.
- In the instantiation `S1` of `busslave`, the interface port `sbus` is mapped to the `slave` modport of the interface instance `busb`.

This modport selection method works well when different instances of a module definition use an interface in different ways.

What Are Interface Methods?



A sub-routine defined within an Interface is called an Interface method.

Writing/reading interface signals can be handled by tasks:

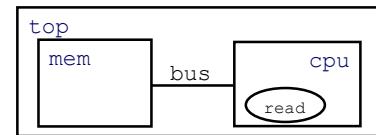
- Abstract interface communication

However, every module must contain copies of the tasks:

- Maintainability issues

Solution is to declare tasks as part of the interface:

- Accessible to any module connected to the interface



```
module cpucore(ifa bus);  
  
task read (input address);  
  @(posedge clk);  
  bus.addr = address;  
  ...  
endtask  
...  
read(8'hFF);  
...  
endmodule
```

```
interface ifa(input clk);  
  logic req, start, gnt, rd़y;  
  logic [1:0] mode;  
  logic [7:0] addr;  
  wire [7:0] data;  
endinterface : ifa
```

144 © Cadence Design Systems, Inc. All rights reserved.



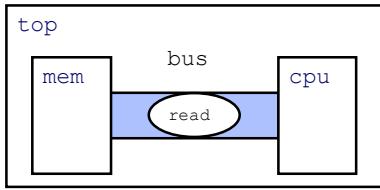
Typically, operations on interface signals, such as read or write operations, are defined as tasks in a testbench. The tasks are then called in required order with different arguments to meet verification goals.

One key aspect of an interface is that it can be re-used everywhere a specific bus or collection of signals is used. When an interface is used with multiple testbenches, each testbench must have a copy of the stimulus tasks used to perform operations on the interface signals. This makes task declarations and calls difficult to maintain.

For SystemVerilog, you can declare the task as part of the interface. Then any module which connects to the interface can access the task.

Accessing Interface Methods in Module

- Same syntax and statements as in a module task definition
- Declare once – visible to any module connected to interface
- Call methods through interface port or instance



```

module cpucore(ifa bus);
    ...
    bus.read(addr,data);
    ...
endmodule

interface ifa (input clk);
    logic req, start, gnt, rdy;
    logic [1:0] mode;
    logic [7:0] addr;
    wire [7:0] data;

    task read (input byte raddr,
               output byte rdata);
        @(posedge clk);
        addr = raddr;
        ...
        rdata = data;
    endtask
endinterface : ifa

```

145 © Cadence Design Systems, Inc. All rights reserved.



You can place subroutine definitions within the interface declaration. Connected modules can call the subroutines using the interface port name as a prefix to the subroutine call, just like accessing an interface signal.

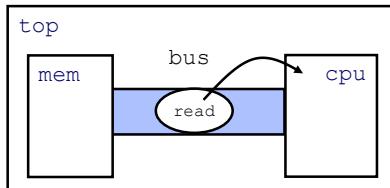
This promotes the interface as a reusable block as it now contains the stimulus tasks which operate on the interface signals.

Controlling Access to Interface Methods Using Modports

If connected to an interface modport, a module sees only:

- Ports defined in that modport
- Methods imported from that modport

Use `import` to make methods visible via the modport



```

interface ifa (input clk);
    logic req, start, gnt, rdy;
    logic [1:0] mode;
    logic [7:0] addr, data;
endinterface

modport cif (
    input clk, gnt, rdy,
    output start, req, mode, addr,
    inout data,
    import read );
task read (input byte raddr,
            output byte rdata);
    ...
endtask

endinterface : ifa

module cpucore(ifa.cif bus);
    ...
    bus.read(addr, data);
    ...
endmodule

```

146 © Cadence Design Systems, Inc. All rights reserved.

 cadence®

If you are accessing an interface using modports, then you must `import` subroutines into the modport to make them visible. Without the `import`, the subroutines are not visible to a module connected via the modport.

Within a modport, apart from importing a subroutine from the interface to make it available to the connected module, you can export a subroutine from a connected module to make it available to the interface.

How to

Declare a Module Port of a Generic Interface Type



You can use the `interface` keyword rather than a specific interface type in your module declaration's list of ports. This serves as a placeholder for an interface type and you can select the interface when instantiating the module.

You can declare a module port of a generic interface type:

- This defers actual interface selection until module instantiation.
- Must use Verilog-2001 ANSI-style port declaration.

```
interface ifa;
    logic req, start, gnt, rdy;
    logic [1:0] mode;
    ...
endinterface : ifa
```

```
module memory(interface bus);
    ...
endmodule : memory
```

```
module cpucore(interface bus);
    ...
endmodule : cpucore
```

1. Use a generic interface port in module definition.

```
module top;
```

```
    ifa busa();
```

```
    memory mem (.bus(busa));
    cpucore cpu (.bus(busa));
```

2. Map Specific interface during instantiations.

```
endmodule : top
```

147 © Cadence Design Systems, Inc. All rights reserved.



You can use the `interface` keyword rather than a specific interface type in your module declaration's list of ports. This serves as a placeholder for an interface type. You can select the interface when instantiating the module. This allows a module to connect to different versions of a specific interface. These versions may implement the interface methods in different ways, or there may be versions which contain extra logic, coverage or assertions checking. The unspecified interface is called a “generic interface reference”. You can use it only with the Verilog named port connections.

Note: You cannot just connect a module interface port to any arbitrary interface. The interface mapped to the port must contain declarations of the signals referenced within the module via the interface port name.

Module Summary

Interfaces simplify design block communication.

- You can represent a number of signals as a single port
- This reduces code verbosity and promotes code reuse
- It “should” be synthesizable if there are no other unsynthesizable constructs

Interfaces raise the level of abstraction

- All declarations are contained in a single location
- They produce a much more readable design

You can define tasks and functions inside an interface

- This encapsulates the inter-module communications protocols
- Example: A “Bus Read” task that you define and execute inside an interface

With modports, you can define different views of the interface

- Example: Define master and slave views of a bus that has shared signals but different port declarations and restricted capabilities

148 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Quiz

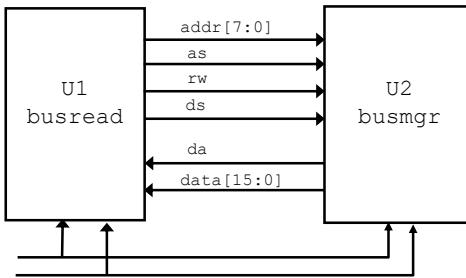


Define an interface to represent the depicted interconnect:

- All integral types in interconnect are of 4-state logic type.
- Use modports to ensure correct signal flow.
- Include `clk` and `rst` interface ports.

Write code to instantiate and connect the modules and interface.

testbench



```
interface quizif (input clk, rst);
  logic as, rw, ds, da;
  logic [7:0] addr;
  logic [7:0] data;

  modport RD (input clk, rst, da, data, output ad
dr, as, rw, ds);

  modport MGR (input clk, rst, addr, as, rw, ds, o
utput da, data);

endinterface : quizif

module testbench;
  logic clk = 0;
  logic rst = 0;
  quizif busa(clk, rst);
  busread U1 (busa.RD);
  busmgr U2 (busa.MGR);
endmodule
```



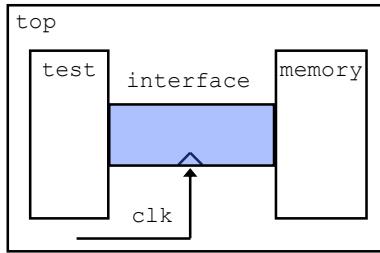
This page does not contain notes.



Labs

Lab 7 Using a Memory Interface

- Modify your memory module test to use an interface.



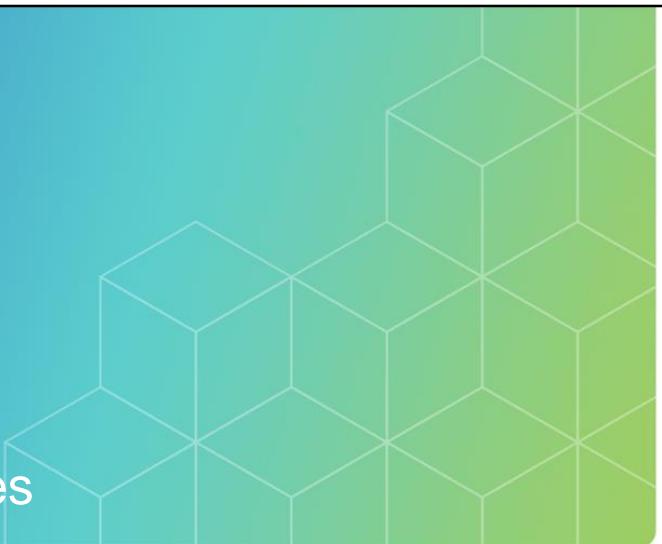
Lab 8 Verifying the VeriRISC CPU (Optional)

- Integrate the modules from previous labs into a complete CPU design and simulate.

150 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



Simple Verification Features

Module **11**

Revision **1.0**

Version **21.10**

Estimated time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Module Objectives

In this module, you explore simple verification features, including the following:

- Declaring strings and string methods
- Analyzing immediate assertions
- Implementing fork-join enhancements



This page does not contain notes.

Strings



IEEE 1800-2012 6.16

string is a new datatype.

- Dynamic array of bytes
- Grows/shrinks automatically to hold contents
- Initial value of "" (empty string)

It is indexed as a normal array.

- With 0 as left-most character

It can be concatenated, replicated and compared.

It can be assigned a string literal:

- Some additional special characters are:
 - \f – form feed
 - \v – vertical tab
 - \a – bell
 - \xdd – hex ASCII

```
string message = "test ";

initial begin
    if (pass)
        message = {message, "passed"};
    else
        message = {message, "failed"};
    $display("%s", message);
    $display("%c", message[0]); // "t"
    ...

```

```
string repstr;

repstr = {2{"go "}}; // "go go "
repstr[2] = "t"; // "gotgo "
```

153 © Cadence Design Systems, Inc. All rights reserved.


In Verilog, you can model strings using arrays of bytes, where each byte contains the ASCII encoding for a single character.

SystemVerilog provides a `string` type, which is essentially a dynamic array of `byte` elements together with defined methods for string manipulation.

The default initial value of a string variable is the special value "", (empty string). Indexing an empty string variable generates an out-of-bounds access error.

The indices of a string are numbered from 0 to N-1, where N is the number of characters in the string, and index 0 accesses the left-most character of the string.

Verilog provides escaped character sequences for including newline (\n), tab (\t), backslash (\) and quote (\") characters, and any octal representation of a character, within a string literal.

To these, SystemVerilog adds the form feed, vertical tab and bell characters, and any hexadecimal representation of a character.



String Operators

Operator	Usage	Function	Description
<code>==</code>	<code>s1 == s1</code>	Equality	Returns 1 if strings are equal
<code>!=</code>	<code>s1 != s2</code>	Inequality	Logical negation of <code>==</code>
<code><</code> <code><=</code> <code>></code> <code>=></code>	<code>s1 < s2</code> <code>s1 <= s2</code> <code>s1 > s2</code> <code>s1 >= s2</code>	Comparison	Returns 1 if condition is true
<code>{str,str}</code>	<code>s = {s1,s2,s3}</code>	Concatenation	Returns concatenated string
<code>{N{str}}</code>	<code>s = {5{s1}}</code>	Replication	Returns string replicated N times
<code>[]</code>	<code>b = s1[index]</code>	Indexing	Returns the byte at index

154 © Cadence Design Systems, Inc. All rights reserved.



All of these string operators are just Verilog operators, so they should look familiar to you.

Equality and comparison operators are evaluated using the lexicographical ordering of the two strings, from left to right.

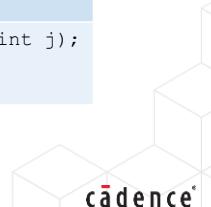
Passing an out-of-bounds index to a string array returns 0 (character NUL).



String Methods

Method	Description	Syntax & Usage
len()	Returns length of the string (length excludes any terminating character).	function int len(); aint = str.len();
putc()	Replaces i^{th} character of the string with the integral value c .	task putc (int i, byte c); str.putc(i, c);
getc()	Returns ASCII code of the i^{th} character in the string.	function byte getc(int i); abyte = str.getc(i);
compare()	Compares strings with regard to lexical ordering.	function int compare(string s); aint = str.compare(s);
icompare()	Compares strings with regard to lexical ordering and is insensitive to character case.	function int icompare(string s); aint = str.icompare(s);
substr()	Returns new string that is a substring formed by characters in position i through j of str.	function string substr(int i, int j); mystr = str.substr(i,j);

155 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



String Methods (continued)

Method	Description	Syntax & Usage
toupper() tolower()	Returns the upper/lower case of the specified string. String is unchanged.	function string toupper(); astr = str.toupper();
atobin() atoct() atoi() atohex()	Returns integer value corresponding to ASCII binary / octal / decimal / hexadecimal representation. Scans all leading digits and underscore (_) characters and stops when any other characters are encountered.	function int atobin(); str = "213A3"; aint = str.atoi(); // = 213 aint = str.atohex(); // = 213A3
atoreal()	Returns real value corresponding to ASCII decimal representation of a number.	function real atoreal(); str = "3.14"; areal = str.atoreal(); //= 3.14
bintoa() octtoa() itoa() hextoa()	Stores the ASCII representation of a binary / octal / decimal / hexadecimal number into a string.	task hextoa(int i); aint = 16'h535; str.hextoa(aint); //str = "535"
realtoa()	Stores the ASCII representation of a real number into the string.	task realtoa(real r); areal = 3.14; str.realtoa(areal); //str ="3.14"

156 © Cadence Design Systems, Inc. All rights reserved.



- toupper(), tolower() – Case-conversion methods return the calling string with all characters converted to upper or lower case, respectively. The original string is left unchanged.
- ato<bin/oct/i/hex>() – ASCII conversion methods return the integral representation of the string. The conversion scans leading digit and underscore characters and stops the scan when an unrecognized character is found. For example, atohex() will recognize characters a to f; atooc() will only recognize characters 0 to 7. The conversions do not accept standard Verilog sized or based literal syntax. They return 0 if no valid integer is found.
- atoreal() – ASCII conversion method returns the real representation of the string. The method accepts all Verilog syntax that constructs a real literal, and stops at any other character. It returns 0 if it finds no valid real literal.
- <bin/oct/i/hex/real>toa() – Conversion methods return the string representation of the integral or real value.

Immediate Assertions



IEEE 1800-2012 16.3

- A procedural `assert` statement is:
 - Similar to an `if` statement.
 - Ignored by synthesis.
- When executed, assertion verifies a Boolean expression:
 - Pass if expression evaluates to 1.
 - Fail if expression evaluates to 0, Z, X.
- Reports `error` message on failure:
 - You can change this behavior.
- You should label your assertions:
 - Label is used in the failure message and helps to manage assertions.

```
module test;
...
always @ (negedge clk)
    A1: assert ( ~ (wr_en && rd_en) );
...
```

Immediate assertion

Verilog Equivalent

```
always @ (negedge clock)
    if (wr_en && rd_en)
        $display("error");
```

Failure message

```
*E,ASRTST (./test.sv,9): (time 10 NS) Assertion test.A1 has failed
```

157 © Cadence Design Systems, Inc. All rights reserved.



Immediate assertions are similar to `if` statements. The assertion passes if the asserted expression evaluates to 1 and fails if the expression evaluates to any other value.

When an assertion fails, the simulator must display the following minimum information:

- The severity level of the assertion.
- The file name and line number of the assertion.
- The simulation time at which the assertion failed.
- The assertion label, or if unlabelled, the scope of the assertion.

The output message can be customized – see the next slide.

The assertion label is optional, but you are strongly encouraged to use meaningful labels for your assertions. You will typically have large numbers of assertions in your design. If you do not label your assertions, the simulator will label them for you, and as the label is used in the failure message, it is much easier to debug, manage and maintain large numbers of assertions with meaningful user-defined labels.

Action Blocks

Assertions can execute statements when they pass or fail:

- Called action blocks.
- Can contain any SystemVerilog constructs.
- Has verification functionality *only*.

`else` block is executed on assertion failure:

- Common to omit pass block.

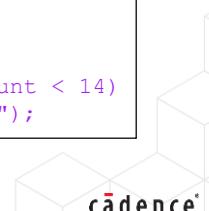
Assertion label can be accessed in the action block via the `%m` format specifier:

- Included in the failure message.

```
always @ (negedge clock)
rwchk: assert (~ (wr_en && rd_en))
$display ("%m: success");
else
begin
$display("read/write fail");
err_count++;
-> rw_err_event;
end

always @ (negedge clock)
valchk: assert (valid)
// pass block omitted
else $display("valid inactive");

always @ (posedge clock)
limit_check: assert (err_count < 14)
else $display("errors >= 15");
```



You can associate an action block with the assertion. In the action block, you place code to execute on either the success or the failure of an assertion. The pass block executes if the assertion succeeds. The fail block executes if the assertion fails. You have the option to include neither, either or both. It is very common to omit the pass block and just include the fail block.

As the assertion is ignored by synthesis tools, only verification code can be placed in action blocks.

The assertion label can be accessed in an action block via the `%m` format specifier. However, as the hierarchical pathname and label for the assertion is always included in the failure message, this is only meaningful for the pass action block.

In the examples above, any `$display` output in the failure action block is printed *in addition*, and after, the standard assertion failure message.

Severity Levels

- Assertion failure can be graded with severity levels:

- \$info
- \$warning
- \$error
 - Default
- \$fatal
 - Terminates simulation

- Severity level is reported.
- You can append output information using syntax identical to that for \$display:
- This is output in addition to the standard failure message.

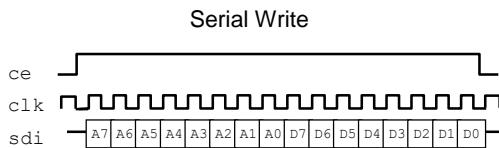
```
always @ (negedge enable)
  valchk: assert (valid)
  else
    begin
      $warning("invalid enable");
      err_count++;
    end

always @ (negedge clock)
  rw_chk: assert (~(wr_en && rd_en))
  else
    $error("wr_en && rd_en");
```



Assertion severity is `error` by default. The simulator reports assertion failure as an error and should return from execution with a nonzero error code. You can override the default severity by using these system tasks in the action block. These system tasks can only be used in an action block. The `fatal` severity has the effect of terminating the simulation. These system tasks accept the same arguments you would use with the `$display` system task. Just remember that prior to your own output, you will also see the standard assertion failure message.

Example: Immediate Assertion for Simple Protocol Check



- Verify that each `ce` pulse contains exactly 16 rising edges on `clk`.
- Immediate assertions provide only an instantaneous Boolean check.
- Extra code is required to execute check at correct point in time:
 - Requires verification and debugging.

```

always begin : CHECK
  // sync to start of frame
  @(posedge ce);
  repeat (16) begin
    // for 16 cycles, find next
    // rising edge on clk
    // or falling edge on ce
    @(posedge clk or negedge ce);
    SP: assert (ce)
    else begin // short pulse
      $error("short ce pulse");
      disable CHECK;
    end
  end
  // find next rising edge on clk
  // or falling edge on ce
  @(posedge clk or negedge ce);
  LP: assert (!ce)
  else // long pulse
    $error("long ce pulse");
end

```



The procedural block follows a standard procedure for checking protocol behavior:

- It synchronizes to the rising edge of `ce` which indicates the start of a frame.
- For 16 iterations, it checks for either a rising edge on `clk` (the expected behavior) or a falling edge on `ce` which indicates a short pulse.
- After every triggering of the event expression, assertion `SP` is used to check `ce` is still high, otherwise `ce` has fallen before 16 rising edges on `clk` were seen, indicating a short pulse. If the assertion fails, we use a Verilog `disable` statement after the error message to terminate the current execution of the procedural block. This is to ensure we only get one error message on a short pulse.
- After the loop, we check for the same event expression, but now a falling edge on `ce` is expected and another rising edge on `clk` indicates a long pulse. Assertion `LP` is used to check that `ce` is low, otherwise we have another rising edge on `clk` indicating a long pulse.

The code performs only a very basic check that `ce` stays high for 16 clocks. In practice, you would have to write much more SystemVerilog code to check other properties of the protocol.

Immediate and Concurrent Assertions

An immediate assertion is an instantaneous Boolean check:

- Single cycle

```
always begin : CHECK
  @(posedge ce);
  repeat (16) begin
    @(posedge clk or negedge ce);
    SP: assert (ce)
      else begin
        $error("short ce pulse");
        disable CHECK;
      end
    end
    @(posedge clk or negedge ce);
    LP: assert (!ce)
      else // long pulse
        $error("long ce pulse");
  end
```

Concurrent assertion describe behavior that spans over time:

- Can span multiple cycles

```
SPI1 : assert property (
  @(posedge clk)
  !ce ##1 ce |> ce[*16] ##1 !ce );
```



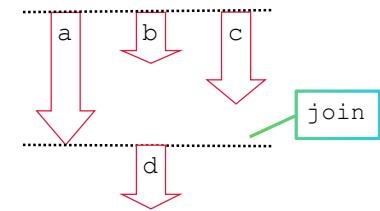
This page does not contain notes.

fork-join Enhancements: join_any, join_none

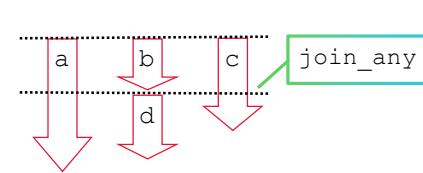


IEEE 1800-2012 16.3

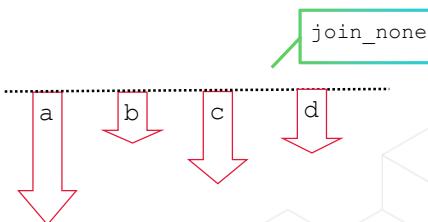
Verilog
`fork
a;b;c;
join
d;`



SystemVerilog
`fork
a;b;c;
join_any
d;`



SystemVerilog
`fork
a;b;c;
join_none
d;`



Verilog fork-join completes when all spawned blocks complete:

- This blocks further execution until the fork-join completes.

SystemVerilog adds two join variants to control when fork-join completes:

- `join_any` completes the fork as soon as any of the blocks completes:
 - Other blocks left running.
- `join_none` completes the fork immediately:
 - All blocks left running.

162 © Cadence Design Systems, Inc. All rights reserved.

cadence®

With the conventional Verilog fork-join, all forked processes must complete before the parent procedure can continue.

SystemVerilog adds the `join_any` and `join_none` variants:

- With `join_any`, the parent procedure continues when any forked process completes.
- With `join_none`, the parent procedure continues without waiting for any forked processes to complete.

As these variants allow the parent procedure to continue while forked processes are still running, SystemVerilog adds statements to control forked processes from outside the fork statement.

Process Control: disable fork, wait fork



IEEE 1800-2012 16.3

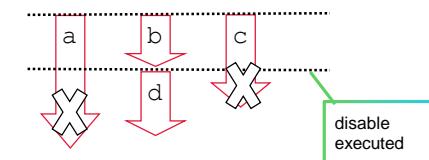
disable fork terminates all active descendants of the current process:

- Use after join_any to ensure only one forked block completes.
- May need to ring-fence to restrict scope.

```

fork
fork
  a;b;c;
join_any
disable fork;
d;
join

```



wait fork stops further execution until all forked blocks complete:

- Affects all forks in current scope.

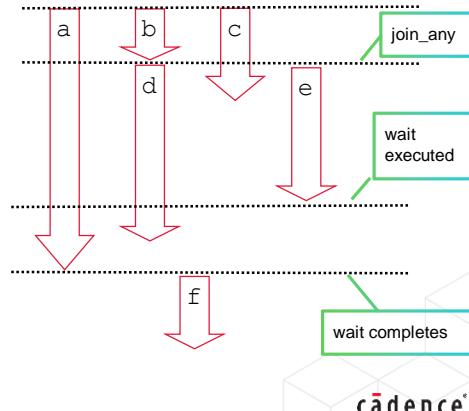
```

fork
  a;b;c;
join_any

fork
  d;
join_none
e;

wait fork;
f;

```



163 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The disable fork statement terminates all sub-processes of the procedure that executes the statement.

Using a fork-join_any with a later disable fork allows a SystemVerilog testbench to spawn several concurrent processes to wait for separate events, for example, separate system interrupts, and as soon as one event is received, terminate the other outstanding processes to handle the received event.

You may need to enclose the disable fork in an additional fork...join layer to restrict its scope of operation. The LRM allows the disable to terminate *any* sub-process, not only forked blocks, and this could also include the procedural block where the disable fork is executed.

The wait fork statement causes the executing procedure to wait until all of its sub-processes have completed.

With the combination of join_any, join_none, and wait fork, you can model complex concurrent and serial behavior. The lower example illustrates this. The parent procedure spawns three subprocesses (a, b and c), and when the first subprocess completes (b), spawns another subprocess (d) with a join_none and continues with its own execution (e). Note that without timing controls, the execution order between the parent (e) and spawned processes (d) is indeterminate. After (e) completes, the procedure executes a wait fork to allow all spawned processes to complete (a and d) complete before continuing its own execution (f).

Using **disable fork**: Ex 1

```

module top;
initial begin : i1
    fork : f1
        #10 $display("pkt_interrupt_handler");
        join_none
        $display("Getting the packet");
        t1();
    end

    task t1();
    fork
        begin : a1
            #1 $display("get_preamble");
        end
        begin: a2
            #2 $display("get_header");
        end
        join_none

    fork
        begin : a3
            #5 $display("get_payload");
        end
        begin : a4
            #6 $display("packet_timeout");
        end
        join_any

    disable fork;
    $display("Processing packet");
endtask
endmodule

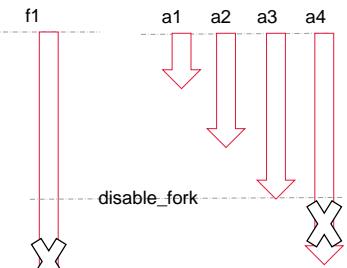
```

output

```

xcelium> run
Getting the packet
get_preamble
get_header
get_payload
Processing packet

```



- Terminates all active descendants of calling process (f1)
- Terminates descendants of process descendants

164 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The disable fork statement terminates all active descendants (subprocesses) of the calling process.

The syntax for disable fork is as follows:

```
disable fork ; // from A.6.5
```

The disable fork statement terminates all descendants of the calling process as well as the descendants of the process's descendants. In other words, if any of the child processes have descendants of their own, the disable fork statement shall terminate them as well.

SystemVerilog provides the disable fork statement, which disables all active threads of a calling process, including any sub-processes spawned by any of those threads. Unlike the disable statement, which terminates the execution of a named block regardless of its relationship to the calling process, the disable fork statement terminates only the processes that were forked by the calling thread.

The disable fork statement kills children of the currently running thread. It means the thread executing the disable fork statement. In this example, the parent thread is enclosed by the i1 scope. It starts a child thread, f1, then calls the t1 task.

Inside t1 task, the first fork starts the a1 and a2 threads. Then, the second fork starts a3 and a4. Since the second fork has join_any, it waits for one of the threads to complete before executing the disable fork statement.

The disable fork will not execute until the a3 thread completes at time 5, and only f1 and a4 threads are still active and will be killed

Using *disable <block_name>* Instead of *disable fork*: Ex 2

```

module top;
initial begin : i1
  fork
    f1;
    #10 $display("pkt_interrupt_handler");
    join_none
    $display("Getting the packet");
    t1();
  end

  task t1();
  fork
    begin : a1
      #1 $display("get_preamble");
    end
    begin: a2
      #2 $display("get_header");
    end
  join_none

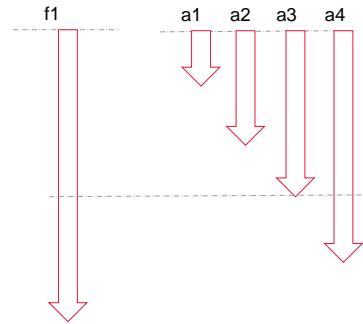
  fork
    begin : a3
      #5 $display("get_payload");
    end
    begin : a4
      #6 $display("packet_timeout");
    end
  join_any
endtask
$display("Processing packet");
endmodule

```

```

xcelium> run
Getting the packet
get_preamble
get_header
get_payload
packet_timeout
pkt_interrupt_handler

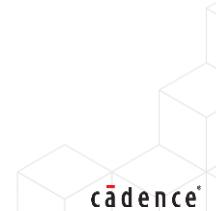
```



- Terminates all active executions of task and tasks called from it
- Does not terminate subprocesses of the task

- disable fork is replaced with disable t1

165 © Cadence Design Systems, Inc. All rights reserved.



The disable statement terminates all active executions of the task and of tasks called from it. It does not terminate subprocesses of the task.

In this same example, we have replaced the 'disable fork' with 'disable t1'. This only kills the active processes of the task, which is the display statement after the 'disable t1' statement.

It cannot kill the subprocesses spawned by same task which is a4. It also cannot kill the processes of the calling process, which is 'f1'.

If you want to do these, you need to use 'disable fork' instead of 'disable t1'.

Usage of **disable fork**: Ex 3

```

module test;
initial
begin
    $display ("%0t: initial block", $time);
    fork:f1
        #300 $display("%0t: this is thread 1", $time);
        #350 $display("%0t: this is thread 2", $time);
    begin // Encapsulating begin-end
        fork: f2
            #350 $display("%0t: this is thread 3", $time);
            #200 $display("%0t: this is thread 4", $time);
        join_none
        #171 begin disable fork; $display("%0t: disable fork", $time);end //
    disable fork
    end // Encapsulating begin-end

    #300 $display("%0t: this is thread 5", $time);
    join_none
end
endmodule

```

```

xcelium> run
0: initialblock
171: disable fork
300: this is thread 1
300: this is thread 5
350: this is thread 2

```

Proper Encapsulation with begin-end is required to disable child processes of the process that spawned it.

166 © Cadence Design Systems, Inc. All rights reserved.



For certain nested fork join_none/join_any scenarios, as shown, “disable fork” might still not work.

If the 'f2' block is not encapsulated, "#171 disable fork" does not disable the thread in the fork:join_none f2 block. You might expect that it should.

To disable the fork:join_none f2 block, an additional begin:end procedural encapsulating block around block f2 and the disable statement is required.

This encapsulating block ensures that disable fork does not kill any other child processes of the process that spawned it.

With the encapsulation in place, thread 4 of block f2 is killed by disable fork.

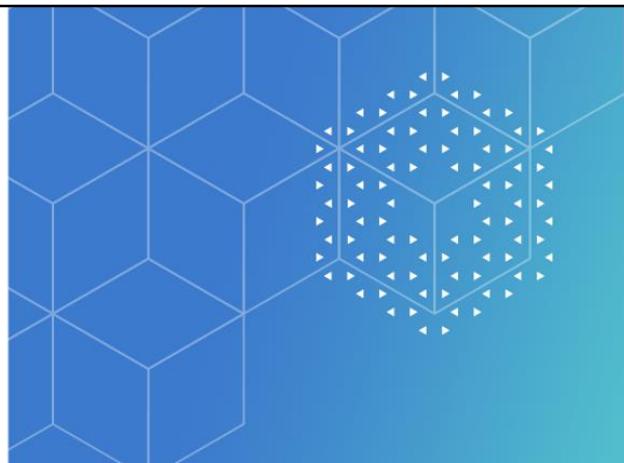
Quiz



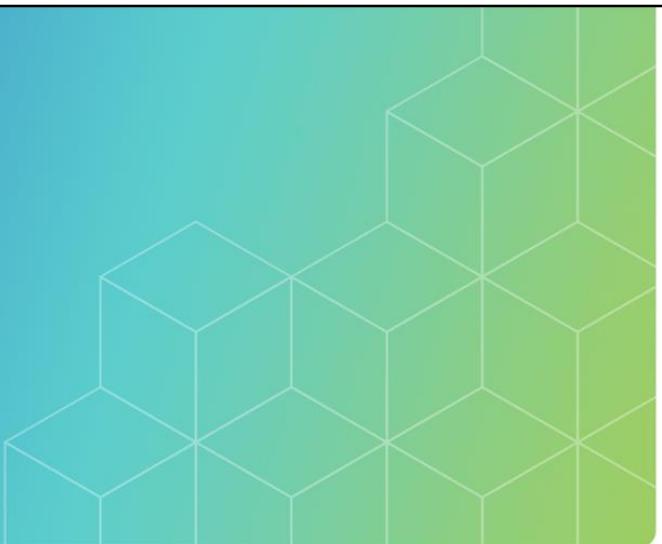
Fill in the following blanks.

- string type variables allow easy creation and manipulation of messages.
 - Arrays are dynamic, so there is no need to worry about fixed array sizes.
 - Supported by large number of conversion and translation methods, for example:
 - itoa() converts from decimal integer to ASCII.
 - atohex() converts from ASCII to Hex.
- Immediate assertions simplify verification and error checking.
 - Action Blocks can execute code on pass or fail.
 - Default severity is \$error, while a \$fatal severity terminates simulation.
- *fork-join* enhancements allow better control of concurrent testbench behavior.
 - join_any completes a fork when the first of the spawned blocks completes.
 - join_none completes a fork immediately.

This page does not contain notes.



Clocking Blocks



Module **12**

Revision

1.0

Version

21.10

Estimated time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Module Objectives

In this module, you

- Analyze the concepts of clocking blocks
- Use clocking blocks to
 - Avoid race conditions when driving clock and design inputs
 - Control sampling of design outputs
- Simplify verification code using clocking block cycle delays

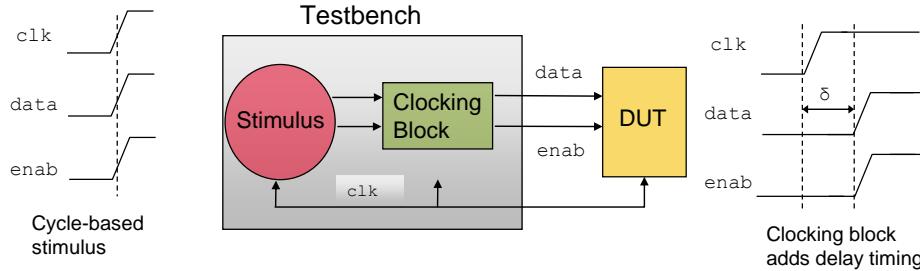


This page does not contain notes.

What Is Clocking Block?



The *clocking block* construct identifies clock signals and captures the timing and synchronization requirements of the blocks being modeled.



- Driving or sampling DUT ports on active clock edges can lead to race conditions.
- Clocking blocks are a verification construct to help avoid this:
 - Driving testbench outputs via a clocking block adds delay (skew) to transitions.
 - A clocking block can also sample testbench inputs with a set delay.
- Clocking blocks separate signal timing from signal function and allow stimulus to be written in terms of cycles and transactions only.



Driving testbench outputs and sampling testbench inputs on the active edge of a clock can potentially lead to race conditions between data and clock. Clocking blocks are a verification construct to make the timing of drives and samples explicit and relative to a specific clocking event.

The aim of a clocking block is to:

1. Separate signal timing from structural, functional and procedural elements of a testbench.
2. Make explicit and unambiguous the timing of input drives and output samples.
3. Simplify stimulus by writing cycle-based transactions and using a clocking block to add fine-grain timing delays. (Stimulus is further simplified by using cycle delays and synchronous events using clocking blocks.)

A clocking block defines a set of timing, relative to a specified clock, for a set of signals. Signals are divided into outputs to be driven into the design and inputs to be sampled from the design.

When you drive an output via a clocking block on the edge of a clock, the clocking block adds delay (called skew) to the output transition, relative to the clock event.

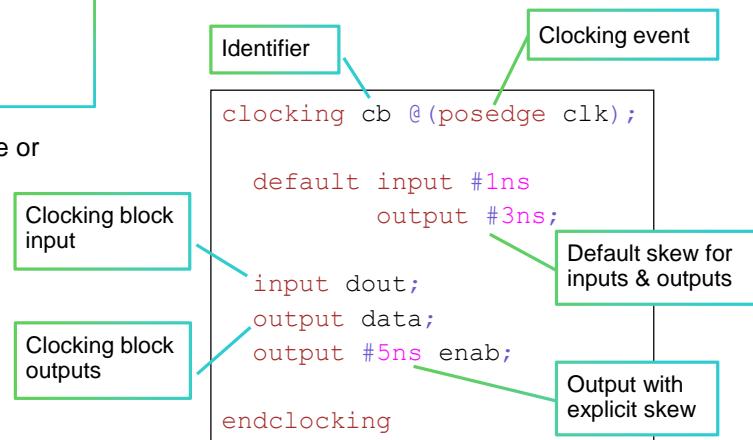
Likewise, a clocking block can sample an input at a specific skew before the edge of a clock. Reading the output via the clocking block retrieves the output value from the last sample time.

Clocking Block Declaration



IEEE 1800-2012 14.3

- A clocking block can be declared in a module or interface.
- Clocking block declaration defines:
 - The clocking event.
 - Signal direction:
 - Does not declare signals.
 - Direction is from testbench perspective.
 - Input and output delay (skew):
 - Explicitly or by default.
- Outputs are driven skew units after the clocking event.
- Inputs are sampled skew units before the clocking event.



171 © Cadence Design Systems, Inc. All rights reserved.



A clocking block is both a declaration and instance of that declaration. You do not separately instantiate a clocking block. A clocking block can be declared in an interface, module or program. It cannot be declared in a package or a Compilation Unit Scope.

A clocking block declaration defines:

- The clocking block event: Usually an edge on a clock signal.
- Signal direction: A clocking block does not declare signals, but only qualifies the direction of pre-declared signals for the clocking block. The direction is always from the perspective of the testbench. Clocking block outputs are design inputs, and clocking block inputs are design outputs. Bidirectional signals can be qualified as `inout`, which is simply a shorthand notation for separate input and output qualifiers on the same signal.
- Input and output delay (skew): Skew can be defined explicitly for each input or output, or by default for all inputs and outputs. Default and explicit skew can be mixed, with explicit skew takes precedence.

Outputs are delayed in the clocking block by skew delay after the clocking block event, before being driven into the design.

Inputs are sampled by the clocking block at skew delay before the clocking block event.

Clocking Block Output Drive

For timing, drive output via the clocking block:

- `cb.enab <= 1;`
- Must use a nonblocking assignment.

This schedules assignment with skew:

- Wait for the clocking block event.
- Use current time if assigned at same time as the event occurs.
- Wait for specified skew delay.
- Drive output.

Can synchronize to clocking block event:

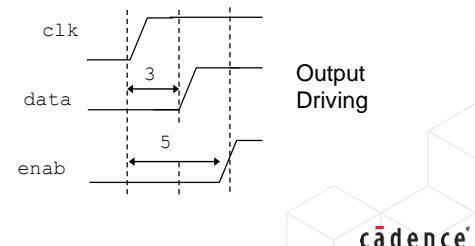
- `@(cb);`
- Similar to `@(posedge clk);`

Timing applies only when output is driven through the clocking block.
Avoid mixing clocking block and regular assignment.

```
bit data, enab;
clocking cb @(posedge clk);
  default input #1ns output #3ns;
  input dout;
  output data;
  output #5ns enab;
endclocking

initial begin
  @(cb);
  cb.data <= 1'b1;
  cb.enab <= 1'b1;
  ...

```



172 © Cadence Design Systems, Inc. All rights reserved.

cadence®

To apply output skew delay, the outputs must be driven via the clocking block using a nonblocking assignment.

The drive value is stored in the clocking block, and the block waits for the clocking event. If a clocking event occurred in the current time step, then the output skew is applied from the current time.

Otherwise, the block synchronizes to the next time step in which the clocking event occurs.

When synchronized to the clocking event, the block waits for the specified skew delay.

After the delay has passed, the signal is assigned in the Re-NBA region. See Appendix B for more details on the SystemVerilog Event Scheduler.

Outputs with explicit zero skew (#0) are driven in the Re-NBA region of the time step in which the clocking event occurs.

You can synchronize to the clocking block event by using the clocking block name as an event expression (for example `@(cb)`). However, to avoid a race condition with sampled inputs, the `@(cb)` event is triggered in the Observed region of the time step in which the clocking event occurs. See Appendix B for more details on the SystemVerilog Event Scheduler.

Clocking block output drive can be mixed with regular procedural assignment, but achieving the required stimulus can be difficult. Driving outputs only via the clocking block is more efficient, readable and maintainable. Stimulus is simplified by describing it purely in terms of clocking block drives and clocking block events.

Clocking Block Input Sample

Clocking block automatically samples inputs:

- Input sampled at specified skew before clocking event.
- Input updated in observed region.

For timing, read input via clocking block:

- `dreg <= cb.dout;`

This reads input from the last sample point:

- Input sample from clocking block may be different from current value.

Can synchronize to a change in input sample:

- `@(cb.dout);`

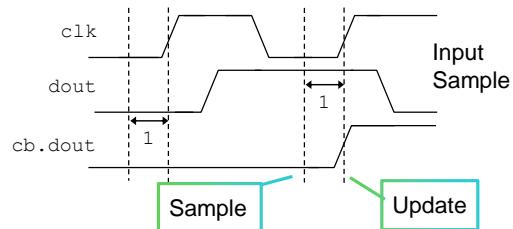
Timing applies only when input is read via a clocking block.



```
bit [7:0] dreg, dout;
clocking cb @ (posedge clk);
  default input #1ns output #3ns;
  input dout;
  output data;
  output #5ns enab;
endclocking

initial begin
  @(cb);
  //read last sample
  dreg <= cb.dout;
  ...

```



cadence®

173 © Cadence Design Systems, Inc. All rights reserved.

To apply input skew delay, the inputs must be read via the clocking block.

A clocking block automatically samples inputs at the specified skew delay before the clocking event. These values are stored in the clocking block, and can be read at any time by reading the input via the clocking block. Obviously, the sampled value read from the clocking block may be different from the current value of the signal.

Inputs with nonzero skew are sampled in the Postponed region of the time step skew time units before the clocking block event.

To avoid a race condition, inputs with explicit zero skew (#0) are sampled in the Observed region of the time step in which the clocking event occurs. See Appendix B for more details on the SystemVerilog Event Scheduler.

You can synchronize to a change in a clocking block input sample, e.g., `@(cb.dout);`, or to a specific edge in an input or slice of input vector, e.g., `@(posedge cb.dout[1]);`

By reading sampled inputs via a clocking block, stimulus can be greatly simplified.

Input and Output Skews



IEEE 1800-2012 14.4

You can specify skew:

- With a default value:
 - Separate default for inputs and for outputs.
- Explicitly in the signal identifier:
 - Overrides the default.

Skew can be:

- A constant expression, parameter or number:
 - Must be positive.
 - Uses timescale of current scope.
- An edge (posedge, negedge, edge) of the clocking event.
- A time literal (including #1step).

```
clocking defio @ (posedge clk);
  default input #1step output #3;
  input sout;
  output ain, bin;
  output negedge cin;
endclocking
```

```
clocking defop @ (posedge clk);
  default output #3;
  input #1step sout;
  output ain, bin;
  output negedge cin;
endclocking
```

```
clocking exp @ (posedge clk);
  input #1step sout;
  output #3ns ain, bin;
  output negedge cin;
endclocking
```

174 © Cadence Design Systems, Inc. All rights reserved.



Skew can be specified with a default value or explicitly in the signal direction identifier. Explicit skew overrides the default skew. There can be separate default values for inputs and/or outputs.

A clocking block can be declared without any default or explicit skew, in which case input skew is #1step and output skew #0.

A skew can be defined a positive number or constant expression, which can include parameters.

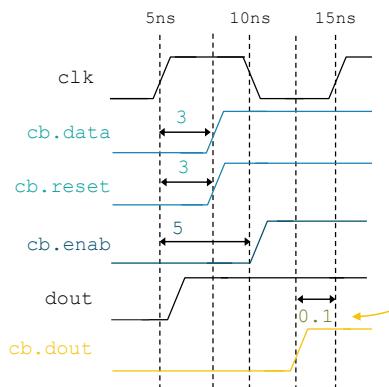
Alternatively, skew can be defined as an edge (which refers to the clocking block event) or a time literal (e.g., #1ns or #1step).

An input with skew of #1step is sampled at the end of the previous time step before the clock event, specifically in the Postponed region. An input with skew of #0 is sampled at the same time as the clock event, but to avoid races is sampled in the Observed region.

An output with skew of #0 is driven at the same time as the clock event in the Re-NBA region.

Clocking block skews are declarations, not procedural statements – in particular a #0 skew does not suspend any process.

Example: Clocking Block with Skews



```
module one;
  timeunit 1ns;
  timeprecision 100ps;
  bit clk, reset, data, enab;
  bit [7:0] dout, din;
  clocking cb @(posedge clk);
    default input #1step
      output #3;
    input dout;
    output reset, data;
    output negedge enab;
  endclocking
  ...

```

Assuming a clock period of 10ns:

reset, data	driven #3 after (posedge clk)	= 8ns
enab	driven on (negedge clk)	= 10ns
dout	sampled #1step before (posedge clk)	= 14.9ns

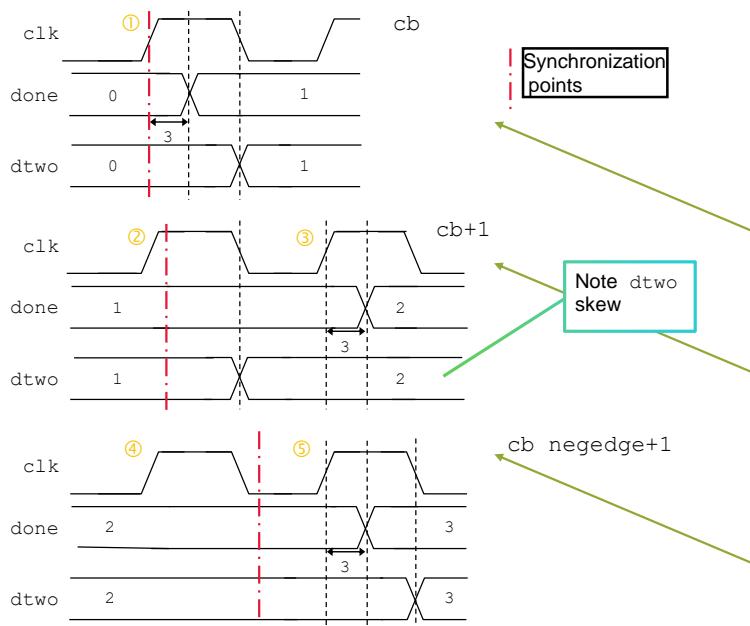
175 © Cadence Design Systems, Inc. All rights reserved.



Assuming a clock period of 10ns, and given a timescale of 1ns and timeprecision of 100ps, the timing for this example is as follows:

- reset and data outputs are driven 3ns after the clock event, at 8ns, 18ns, 28ns, etc.
- enab is driven on the negative edge of the clock, at 10ns, 20ns, 30ns, etc.
- dout input is sampled one simulation time step (100ps), before the clock event, at 4.9ns, 14.9ns, etc.

Example: Output Skew Synchronization



```

bit [3:0] done, dtwo;
clocking cb @ (posedge clk);
  output #3ns done;
  output negedge dtwo;
endclocking

initial begin
  @(cb); // sync to cb event
  cb.done <= 1;
  cb.dtwo <= 1;
  @(cb);
  #1ns; // cb event + 1ns
  cb.done <= 2;
  cb.dtwo <= 2;
  @(cb);
  @(negedge clk);
  #1ns; // cb negedge + 1ns
  cb.done <= 3;
  cb.dtwo <= 3;
  ...

```

176 © Cadence Design Systems, Inc. All rights reserved.



Clocking blocks are intended for use with cycle-based stimulus. If you add delays to cycle-based stimulus, synchronization can be affected and unintuitive results seen.

If clocking block output drives are executed in a time slice at which the clocking block event occurs, then skew is added from the current time. For example, in the first waveform, we synchronize to `clk` cycle 1 with `@(cb)`, assign the drives to `done` and `dtwo`, and add skew from the current time (cycle 1).

Note that the SystemVerilog LRM is not explicit on this behavior, but most vendors conform to this interpretation.

If output drives are executed at a time slice in which the clocking block event has not occurred, then the clocking block waits until the next clocking block event, **unless** a transition occurs on the clocking signal which matches a skew specified with that edge. For example, in the second waveform, we delay until 1ns after cycle 2, then assign the drives. Because `dtwo` has a skew of `negedge` and a falling edge of `clk` (for cycle 2) occurs before cycle 3, `dtwo` is driven at the falling edge of cycle 2, whereas `done` is not driven until 3ns after the rising edge of cycle 3.

In the final waveform, you wait until 1ns after the falling edge of cycle 4 before executing the output drives. In this case, you synchronise to the rising edge of cycle 5, drive `done` 3ns after the rising edge and drive `dtwo` on the falling edge of cycle 5.

Defining an edge skew for an output drive effectively redefines the clocking event for that output – the skew is no longer relative to the clocking block event, but is the absolute next edge event on the clocking signal.

Multiple and Default Clocking Blocks



IEEE 1800-2012 14.8

IEEE 1800-2012 14.12

- You can define multiple clocking blocks in a scope:
 - For multiple clocks, different signals, or different timing
- One block in a scope can be defined as a default clocking block:
 - Add default to the beginning of the clocking block declaration
 - Or use a default statement separate from the declaration
- Default clocking blocks allow cycle delays.

```
clocking cb1 @ (posedge clk1);
  default input #2 output #3;
  input dout;
  output reset, data;
endclocking

clocking si1 @ (posedge clk2);
  input #2 ip1;
  output #2 op1;
endclocking

default clocking si2 @ (posedge clk2);
  input #2 ip2;
  output #5 op2;
endclocking
```

```
clocking si2 @ (posedge clk2);
  input #2 ip2;
  output #5 op2;
endclocking

default clocking si2;
```

177 © Cadence Design Systems, Inc. All rights reserved.



A scope can contain multiple clocking blocks. This may be required for multiple interfaces with different timing or clock events, different timings for the same signals depending on the clock event (e.g., DDR interfaces) or different timings for the same signals on the same clocking event depending on different design modes.

One block only in a scope can be defined as the default clocking block. There are two ways to define the default block:

1. In its declaration, using the keyword `default`.
2. You can define an existing clocking block to be the default.

Only one default clocking block can exist in a module, interface or program.

A default clocking block is valid only in the scope declaring it and any nested declarations.

Default clocking blocks allow the use of cycle delay constructs.

Cycle Delay



IEEE 1800-2012 14.11

- You can insert cycle delays for the default clocking block:
 - `##N`: Positive number or identifier
 - `##(expr)`: Positive integer expression
- Procedural delay always refers to the default clocking block:
 - `##1 cb.sig <= 1;`
- Synchronous drive delay always refers to target variable's clocking:
 - `cb.sig <= ##2 var;`
 - This is not intra-assignment delay:
 - Illegal for clocking block signals

```

default_clocking cb1 @ (posedge clk);
  input #2 dout;
  output #3 reset, data;
endclocking

clocking cb4 @ (negedge clk);
  output #3 enab, rdata;
endclocking

initial begin
  // Wait 2 cb1 cycles
  repeat (2)
    @(cb1);
  // Wait 2 cb1 cycles
  ##2;
  // Drive data after 1 cb cycle
  ##1 cb1.data <= 2'b01;
  ##1; cb1.data <= 2'b10;

  // Drive rdata with current dreg
  // value after 3 cb4 cycles
  cb4.rdata <= ##3 dreg;

```



A default clocking block allows cycle delays. The delays are of the clocking block event.

The number of cycles can be defined using a number or identifier or any expression which evaluates to a positive integer. Expressions must be enclosed in brackets.

`##0` is allowed, and is treated as a special case. `##0` means wait for a clocking event in this time step but do not wait if it has already occurred.

The semantics of the cycle delay (`##N`) operator differ depending on how you use it:

- When used as an intra-assignment delay in an assignment to a clocking block signal, it refers to the clock event of the specified clocking block. These assignments must use the nonblocking operator.
- When not used as intra-assignment delay in an assignment to a clocking block signal, it refers to the default clocking block. For this situation, the compiler shall issue an error if no default clocking block has been declared.

Handling Hierarchical Expressions in Clocking Blocks



Clocking blocks can drive or sample signals via a hierarchical expression:

- Out-Of-Module References (OOMRs)
- Slices or concatenations of signals in current or other scopes

Problem: A clocking block signal *cannot* be a hierarchical expression:

Solution: You must *associate* a clocking block signal with the hierarchical expression

```
clocking cb @ (posedge clk);
    default output #3;
    // output top.dut.data;
    // data associated with hierarchical expression
    output data = top.dut.data;
endclocking
```

Error – direct hierarchical expression is illegal

Correct – hierarchical expression associated with signal



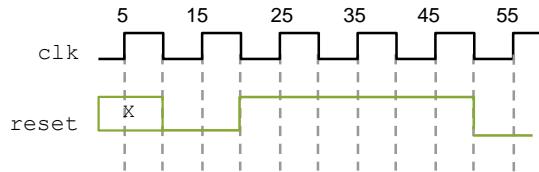
You can use clocking blocks to drive or, more likely, sample signals in other scopes using hierarchical pathnames. For example, you may need to sample internal signals of the design at specific time points. You can also use clocking blocks to drive or sample slices or concatenations of signals from other scopes.

However, you cannot directly qualify a hierarchical signal as an input or output within a clocking block. You must declare a clocking block virtual signal and associate it with an expression that references the hierarchical signal. The semantics of this association are much like those for a port connection – you can use any expression, including slices, concatenations and part selects, that you can legally connect to a port of that direction.

Quiz

Draw the waveform for the following stimulus, assuming the following:

- clk period of 10ns
- An initial value of X for reset



```
clocking cb @ (posedge clk);
    default input #1step    output #5;
    output reset;
endclocking

initial begin
    cb.reset <= 0;           //at first posedge clk + 5ns
    ##2 cb.reset <= 1;      //wait 2x posedge clk then drive @ +5ns
    ##3 cb.reset <= 0;      //wait 3x posedge clk then drive @ +5ns
    ...

```

180 © Cadence Design Systems, Inc. All rights reserved.



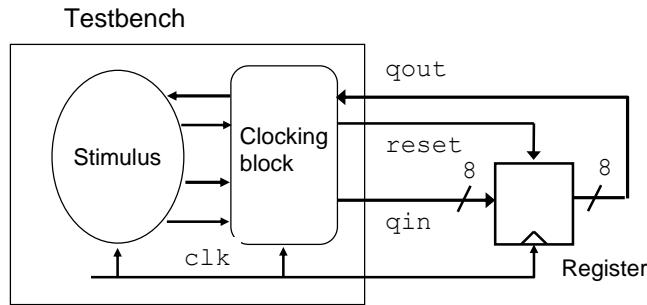
This page does not contain notes.



Lab

Lab 9 Using a Simple Clocking Block

- Use a testbench for a simple register to explore clocking block behavior.



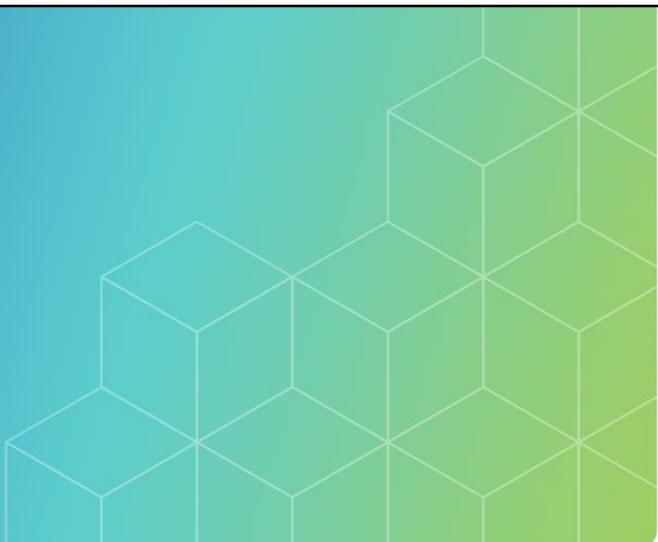
181 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



Random Stimulus



Module **13**

Revision

1.0

Version

21.10

Estimated time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Module Objectives

In this module, you

- Identify the benefits of using random data generation in verification
- Create random data using the randomize method
- Analyze implementation issues in randomization such as seeding and stability
- Use constraints to control randomization
- Randomly select statements using the randcase construct



This page does not contain notes.

CPU Coverage Test Case



Declare opcode and register values as enumerated types.

Assign values for driving into DUT.

Problem: How to test combinations of opcode, register and data?

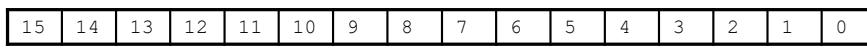
Solution 1: Using Looped CPU Instruction Stimulus

Solution 2: Applying SV Randomization and Constraints

This Simplified 8-bit CPU has:

- 7 immediate instructions:
 - ADDI, SUBI, ANDI, XORI, JMP, JMPC, CALL
- 4 registers:
 - REG0, REG1, REG2, REG3

In the following double byte format...



| X | X | X | - opcode - - | - reg - | - - - - - - - data - - - - - - |

184 © Cadence Design Systems, Inc. All rights reserved.

CPU Instruction Stimulus

```
typedef enum bit[2:0] {ADDI, SUBI, ANDI, XORI, JMP, JMPC, CALL} op_t;
typedef enum bit[1:0] {REG0, REG1, REG2, REG3} regs_t;

op_t          opc;
regs_t        regs;
logic[7:0]    data;

initial begin
    // generate stimulus
    opc = ADDI;
    regs = REG0;
    data = 5;
    // drive into DUT
    ...

```



Let us take a look at why you might want to generate random stimulus and constrain it to meaningful sets of values.

To do this, we will use a simplified example of an 8-bit CPU on which we want to test the defined immediate instruction set with four internal registers.

Stimulus is sent to the CPU as an 8-bit data field, a 2-bit register field to address four registers, and a 3-bit opcode field to specify one of the seven immediate instructions.

Solution 1: Looped CPU Instruction Stimulus

```

typedef enum bit[2:0] {ADDI, SUBI, ANDI, XORI, JMP, JMPC,
CALL} op_t;
typedef enum bit[1:0] {REG0, REG1, REG2, REG3} regs_t;

op_t      opc;
regs_t    regs;
logic[7:0] data;

initial begin
  opc = opc.first();
  regs = regs.first();
  repeat(opc.num()) begin
    repeat (regs.num()) begin
      for (data=0; data<255; ++data)
        @ (posedge clk);
      regs=regs.next();
    end
    opc = opc.next();
  end
  ...

```

ADDI REG0 00 - FF
 ADDI REG1 00 - FF
 ADDI REG2 00 - FF
 ADDI REG3 00 - FF
 SUBI REG0 00 - FF
 SUBI REG1 00 - FF
 SUBI REG2 00 - FF
 SUBI REG3 00 - FF
 AND1 REG0 00 - FF
 ...

Use nested loops to index through 7168 combinations

185 © Cadence Design Systems, Inc. All rights reserved.



A simple approach uses nested loops to generate all possible values over the ranges of the three variables.

The outer loop iterates over the opcode values, the middle loop over the register identifiers, and the inner loop over the range of data values. This generates 7168 separate combinations of values.

Analyzing Looped Stimulus Coverage

- Structured stimulus does not thoroughly test CPU functionality.
- It misses most transitions between value combinations.
 - What if ADDI followed by JMP causes an out-of-range error?

Structural Metrics look good!

	REG0	REG1	REG2	REG3
ADDI	✓	✓	✓	✓
SUBI	✓	✓	✓	✓
ANDI	✓	✓	✓	✓
XORI	✓	✓	✓	✓
JMP	✓	✓	✓	✓
JMPC	✓	✓	✓	✓

Functional Metrics do not!

ADDI:ADDI	✓
ADDI:SUBI	✓
ADDI:ANDI	✗
ADDI:XORI	✗
ADDI:JMP	✗
ADDI:JMPC	✗
...	

REG0:REG1	✓
REG1:REG2	✓
REG2:REG3	✓
REG0:REG2	✗
REG1:REG3	✗
...	

186 © Cadence Design Systems, Inc. All rights reserved.



We then measure the structural and functional coverage of the 7168 patterns.

Structural metrics look good – we have applied every possible value to each variable. Functional metrics do not look good. We have missed most of the transitions between values.

Applying all such transitions requires a much more complex set of stimulus loops!

What we really want to do is to generate random sequences of operations, registers and data, and constrain those random combinations to meaningful combinations.

Solution 2: Employ SV Randomization and Constraints

SystemVerilog provides a new set of constructs for generating and constraining random variables.

- Random generation allows minimal code to generate many data sets.
- Constraints restrict the data set to meaningful data.
- You can apply randomization (with constraints) to:
 - Local variables:
 - "Scope" randomization (this module).
 - Class properties:
 - Class-based randomization (covered later).



Randomization allows you to produce large amounts of stimulus with a minimal amount of code. This lets you more thoroughly and effectively verify the design.

However, full, unrestricted randomization is rarely useful. Control over generated values is required to maintain dependencies between different variables, exclude illegal or undesired values or just restrict data to specific areas of interest.

With randomization constraints, you can restrict the generation and probability of random values to specific values, ranges or areas of interest.

SystemVerilog offers two forms of randomization:

- Randomization of local variables (called scope randomization) covered in this module.
- Randomization of class properties, covered in a later module.

What Are Pseudo-Random Number (PRN) Generators?



PRN Generators generate random sequences algorithmically which are controlled by initial values called as seeds.

Pure random number sequences are useless for verification:

- Sequence must be repeatable.
- Allows debug-fix-debug cycle.

Verification uses *pseudo-random* sequences:

- Controlled by seed(s):
 - Using the *same* seed generates the *same* sequence.
 - Using a *different* seed generates a *different* sequence.

Random

Sim1:	2	5	1	1	6	6	7	0	7	4	3	8
Sim2:	6	2	4	5	4	2	6	0	4	1	5	3
Sim3:	4	4	1	7	3	3	2	8	5	7	3	2

Fixed error cannot be verified, if sequence does not repeat

Pseudo-random

Seed = 4												
Sim1:	4	5	2	1	0	5	1	3	2	1	8	7
Sim2:	4	5	2	1	0	5	1	3	2	1	8	7
Sim3:	4	5	2	1	0	5	1	3	2	1	8	7
Seed = 7												
Sim1:	2	5	6	1	3	2	1	7	4	5	3	5
Sim2:	2	5	6	1	3	2	1	7	4	5	3	5
Sim3:	2	5	6	1	3	2	1	7	4	5	3	5



Pure randomization is useless for verification. Pure randomization creates different data patterns for every simulation run. Hence if you find, debug and define a fix for an error, you cannot check the fix because you cannot repeat the sequence of random values which originally caused the error. The sequence of values created by multiple randomization must be repeatable to successfully debug the design.

For verification and debug, we need a repeatable (pseudo-random) sequence. These sequences are generated algorithmically. The algorithms are controlled by starting values called seeds. The seed is input to the algorithm to generate the first random value. This random value then becomes the input to the algorithm to generate the next random value, and so on. Hence, for a given seed, the sequence of multiple random values is always the same, but the sequence can be changed simply by using a different seed.

randomize(): Randomizing Scope Variables



IEEE 1800-2012 18.7

- The `randomize()` function generates random values on variables:
 - Returns 1 on success:
 - Variables can be randomized
 - Constraints can be met
 - Otherwise returns 0
- You can randomize integral scalar and array variables.
- `randomize()` is defined in a built-in package called `std`:
 - Automatically imported
 - Singular, Integral and Variable restrictions

```
//type declarations as before
opc_t          opc;
regs_t         regs;
logic[7:0]     data;

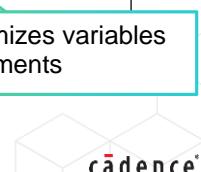
int ok;

initial begin
  repeat(7168) begin
    ok = randomize(opc, regs, data);
    @ (posedge clk);
  end
  ...

```

Function randomizes variables passed as arguments

189 © Cadence Design Systems, Inc. All rights reserved.



It is really easy to generate random values on scope variables. You simply pass the variables as arguments to a built-in function called `randomize()`. The function returns an `int` value, which is 1 for successful randomization and 0 for failure. Typically, randomization will fail if constraints conflict and prevent the random number generator reaching a valid solution. Every time the function is called, random values are applied to the variable arguments.

You can only randomize singular integral variables.

The singular restriction prevents you from randomizing unpacked arrays or unpacked structures. The integral restriction prevents you from randomizing real numbers and the variable restriction prevents you from randomizing any nets.

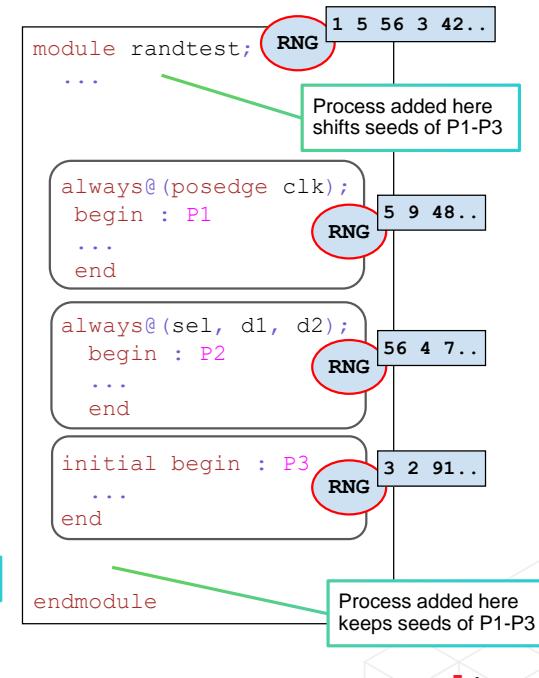
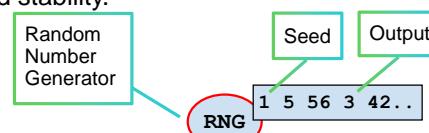
The `randomize()` function is declared in a predefined package called `std`, which is automatically and implicitly imported into every SystemVerilog module. If you wish, you can use the full declarative path for the function `std::randomize()`, but the package reference is optional.

What Is Random Stability?



Random stability is to localize the RNG to threads and objects. Because this makes the sequence of random values returned by a thread or object independent of the RNG in other threads or objects.

- Each design element instance RNG has the same initial seed (default 1).
 - Can be changed on command line `-svseed=x`
- Every instance and process has its own Random Number Generator (RNG).
- Each process RNG is seeded with the next value from parent RNG.
- Adding new processes after existing code maintains seed order and stability.



190 © Cadence Design Systems, Inc. All rights reserved.

cadence

An issue with pseudo-randomization is that the current value is used to generate the next random value. Therefore, to maintain a repeatable sequence, the randomize calls need to be made in the same order every time. Code changes and conditional randomization can change this order, leading to a different order of random values. This is called random instability.

SystemVerilog addresses this by creating a separate, independent Random Number Generator (RNG) for every design element instance, class instance and process thread.

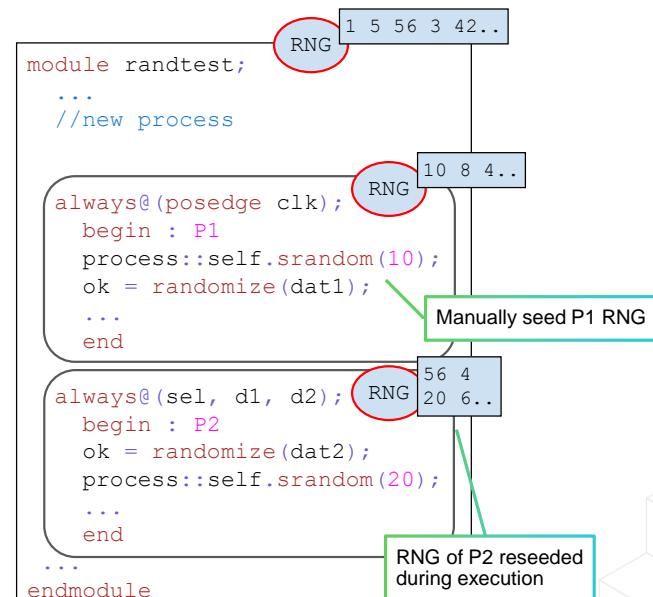
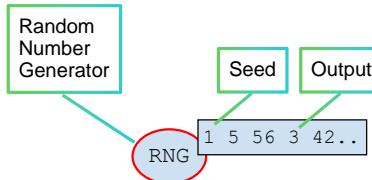
- Each package and design element instance (interface, module) RNG is seeded with an initial seed, usually 1. This value can be changed on the command line (`+svseed=<seed>`).
- When created, the RNG of every static thread or class object is seeded with the next random value of its parent design element's RNG. This is called hierarchical seeding.
- When created, the RNG of each dynamic thread or class object is seeded with the next random value of its parent thread's RNG.
- The RNG of thread and class objects are stable to the extent that their creation order and randomization order are stable.

To maintain random stability, add new threads, class objects and randomization calls after previous threads, class objects and randomization calls.

In the example, the RNG of the module is seeded with 1, and its subsequent values are used to set the RNGs of its procedural blocks (P1 gets 5, P2 56 and P3 3). Inserting another procedural block (P0) before P1 will shift the seeds for all the RNGs (P0 gets 5, P1 56, P2 3, P3 42) breaking the stability. New blocks should be added after P3 to avoid seed stealing.

Setting the Random Seed

- You can manually seed a process Random Number Generator (RNG).
 - Can help with thread stability.
 - But very rare to use this.
 - Use the `srandom()` method of the built-in process class.
- ```
process::self.srandom(seed)
```
- Process object instances are automatically created.



191 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Normally, process Random Number Generators (RNGs) are seeded hierarchically from the parent RNG. However, as we saw on the previous slide, adding new processes in front of existing ones can shift seeding and affect the random stability.

You can remove the dependency of the process RNG on the hierarchical seeding by manually seeding each procedural block using the `srandom()` method via the following syntax:

```
process::self.srandom(seed);
```

where `self()` is a static function of the built-in `process` class which returns a handle to the process making the call.

Objects of type `process` are automatically created when processes are created. A process is either static, created with the `always` or `initial` keyword or dynamic, created with the `fork-join` construct. You cannot create process instances or extend the process class type.

In the example, manually seeding the process `P1` would allow us to add new processes before `P1` without affecting the randomization within `P1`. You can also reseed the RNG at any time. In `P2`, the first randomization uses the hierarchical seed, which could be affected by new processes added before `P1`, but the RNG is then reseeded with 20 for subsequent randomizations, removing the stability dependency on the hierarchical seed.

Manual seeding should only really be used to address issues of random stability when they arise. By fixing the seed manually, changing the default seed with the `+svseed` command option no longer affects the process. A manually seeded process will always generate the same random data.

## What Is a Constraint Block?



The *constraint\_block* is a list of expression statements that restrict the range of a variable or define relations between variables.

- Use the *with* clause to attach a constraint block to the randomize method:

```

typedef enum bit[2:0] {ADDI, SUBI, ANDI, XORI, JMP, JMPC, CALL} op_t;
typedef enum bit[1:0] {REG0, REG1, REG2, REG3} regs_t;

op_t opc;
regs_t regs;
logic[7:0] data;
int ok;
initial begin
 ok = randomize(data) with { data>=32; data<=126; };
 Relational: data between 32 and 126
 ok = randomize(opc) with { opc inside {[ADDI:ANDI], JMP, JMPC}; };
 List: opc in range ADDI to ANDI or JMP or JMPC
 ok = randomize(regs) with { regs dist { [REG0:REG1]:=2, [REG2:REG3]:=1 }; };
 ...
 Distribution: REG0-REG1 twice as likely as REG2-REG3

```

192 © Cadence Design Systems, Inc. All rights reserved.

**cadence**

So far our randomization has been unconstrained, with every possible value of a variable equally probable. Constraints allow us to define a subset of values or different probability distribution. They are rules which the randomizer must follow to generate a value.

A constraint block can be attached to the randomize method using the *with* clause, and enclosing constraint expressions in curly brackets `{ }`. Every constraint expression within the brackets must be terminated in a semicolon.

Constraint expressions can be relational, as in the `data` constraint.

Constraint expressions can define a list of values and value ranges using the *inside* operator, as in the `opc` constraint. The choice of `opc` is limited to the values in the list. Be careful with syntax – the *inside* operator uses curly brackets to define the value list, but the range and value items within the list are comma separated.

Constraints can also change the probability of values using the *dist* operator. *dist* is an enhancement of *inside*, where the values or ranges in the list are given weights – relative probabilities – such as in the `regs` constraint. Here, values `REG0` and `REG1` each has a weight of 2, whereas `REG2` and `REG3` each has a weight of 1. Hence `REG0` has a probability of 2 (its relative weight) divided by 6 (sum of all value weights) i.e., 1/3. The default weight is 1.

A constraint expression can be almost any integral expression. It cannot have side-effects such as using assignment operators. You can use functions in constraints, but there are restrictions. Please refer to the *SystemVerilog Language Reference Manual* (LRM).

## How to Write Conditional Constraints



Conditional constraints allow you to select between different sets of constraints, depending on the value of another variable.

There are two ways of defining conditional constraints:

1. Implication, using the `->` operator
2. if-else, using the `if...else` construct

```
logic[7:0] data;
typedef enum bit[1:0] {SMALL, MEDIUM, LARGE, XL} mode_t;
mode_t mode;

...
ok = randomize(data) with
 {mode == SMALL -> data < 100;
 mode == LARGE -> data > 200;};

ok = randomize(data) with
 { if (mode == SMALL) data < 100; else
 if (mode == LARGE) data > 200; };
...

```

If mode is SMALL, data constraint is <100

If mode is LARGE, data constraint is >200

If mode is neither SMALL nor LARGE, data is unconstrained

Same constraints with if/else

193 © Cadence Design Systems, Inc. All rights reserved.



Conditional constraints allow you to select between different sets of constraints, depending on the value of another variable.

There are two ways of defining conditional constraints:

- Implication, using the `->` implication operator from SystemVerilog concurrent assertion syntax:
  - `expression -> constraint_set`
- If-else (using the `if-else` construct):
  - `if ( expression ) constraint_set [ else constraint_set ]`

Both forms are equivalent, i.e., choice of form is on syntax preference, not functionality.

If no condition in a conditional constraint block is true, then the variables in the block are unconstrained.

Multiple constraints can be defined in each conditional branch by enclosing them in an additional layer of curly brackets.

## Random Weighted Case: `randcase`



IEEE 1800-2012 18.16

You can use `randcase` to randomly select statement(s) for execution.

- You can provide different probabilities (weights) for each branch
- Weights may be constant or variable expressions:
  - Non-negative integral expressions only

Probability(`gen_crc_error()`) = 5/65 ~ 8%

```
repeat (50) begin
 randcase
 20 : gen_atm();
 30 : gen_ethernet();
 10 : gen_ipv4();
 5 : gen_crc_error();
 endcase
 ...

```

```
repeat (50) begin
 randcase
 a : gen_atm();
 a + b : gen_ethernet();
 a - b : gen_ipv4();
 b : gen_crc_error();
 endcase
 ...

```

194 © Cadence Design Systems, Inc. All rights reserved.



You can use the `randcase` keyword to randomly select a statement or group of statements for execution.

For each `randcase` branch, you can define weights (relative probabilities). The default weight is 1. Weights can be defined with expressions which may be either constant or variable, but must evaluate to non-negative integral values. A weight of 0 indicates that the item shall never be selected. Fractional weights are allowed.

## Module Summary

With simple randomization features, you can:

- Generate large amounts of stimulus data from the compact code:
  - Run longer simulations with more stimulus that more thoroughly tests the design
  - Spend more of your own time crafting directed tests of corner cases
- Constrain the values of the random variables:
  - To create legal stimulus
  - To explore areas of interest
  - To conditionally switch between modes of operation
- Use `randcase` to randomly select a block of statements to execute



You can use these SystemVerilog randomization features to easily generate large amounts of stimulus from small amounts of code.

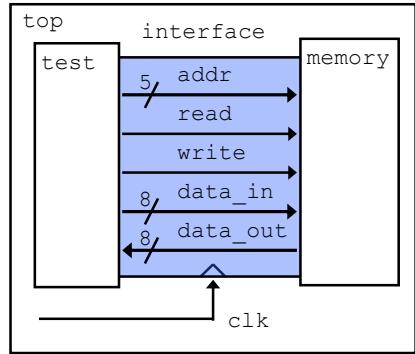
Random stimulus is usually used in conjunction with directed testing. Randomization can test the more easily controlled and observed functionality of the design, while allowing you to spend your time crafting directed tests to verify the less easily controlled and observed functionality.



## Lab

### Lab 10 Using Scope-Based Randomization

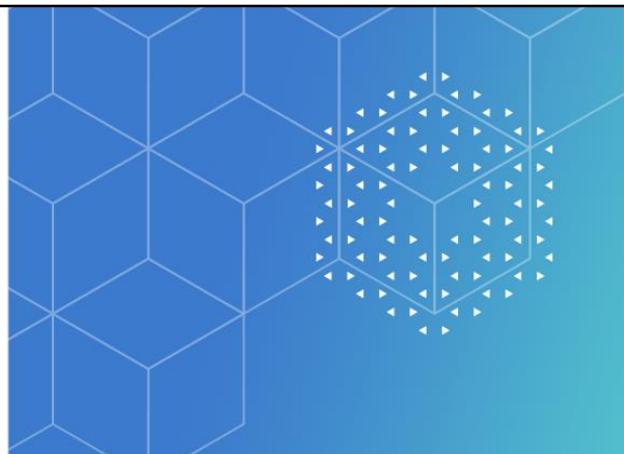
- Modify your memory testbench to use constrained scope-based randomization.



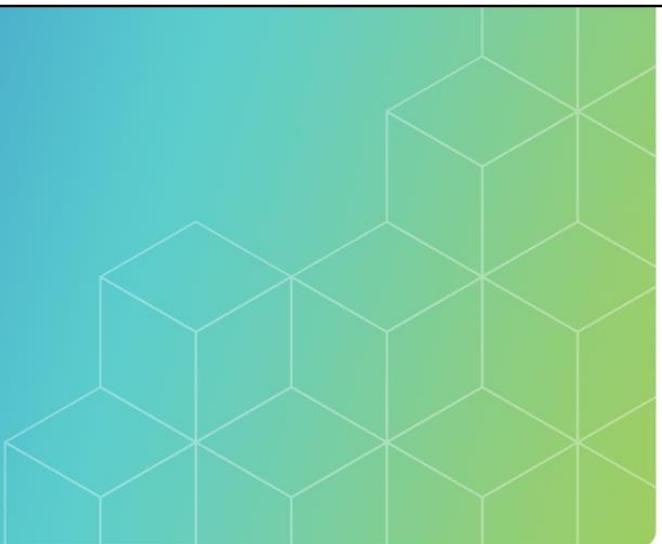
196 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*



## Basic Classes



**Module** **14**

Revision

**1.0**

Version

**21.10**

Estimated time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

- Identify the basic principles and benefits of SystemVerilog classes
- Declare simple classes, properties and methods
- Use explicit constructors to initialize properties
- Define static properties and methods
- Create aggregate classes to group class instances
- Apply inheritance and examine issues with constructors
- Use data encapsulation and information hiding for robust classes



*This page does not contain notes.*

## What Is Object-Oriented Programming?



**Object-Oriented Programming** is a programming paradigm that focuses on objects (as apposed to process).

Data-oriented rather than procedure-oriented

- Focuses on objects instead of actions
- Defines classes to characterize the objects
- Classes declare and own:
  - properties (AKA “attributes” or “data”)
    - Class member data mostly is private – hidden from “outside” code
  - behaviors (AKA “methods” or “functions”)
    - Class member functions often are public – callable by outside code
      - Termed the “public interface methods”

### class Student

#### PROPERTIES

- grade
- name
- number

#### BEHAVIORS

- eat
- play
- sleep
- study



199 © Cadence Design Systems, Inc. All rights reserved.

cadence®

**Object-Oriented Programming** is a programming paradigm that focuses on *objects*, as apposed to *processes*.

It is data-oriented rather than procedure-oriented.

It focuses on objects instead of actions.

It partitions a programming problem into object types, that we refer to as “classes.”

- An object of the class has properties, the collective value of which define the object’s state.
  - The class definition can choose to individually hide or make visible its properties. Classes typically hide most properties.
- An object of the class has behaviors, that collectively define how the object is created, how it interacts with other objects, and how it reacts to applications that use it.
  - The class definition can choose to individually hide or make visible its behaviors. Classes typically expose at least some behaviors.
    - Objects interact with other objects and applications only through the visible properties and behaviors. The visible properties and behaviors are termed the “public interface.”

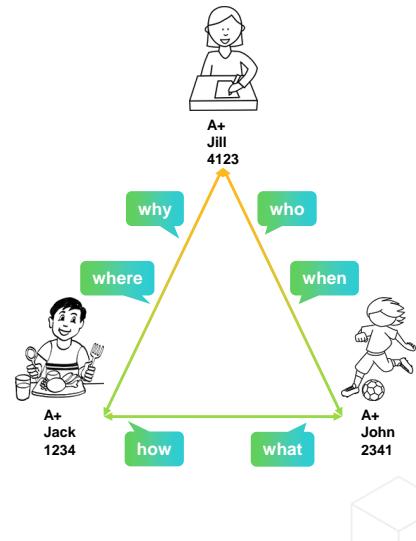
For example, let us imagine an object type that we name “student.” All objects of that type will have the properties of that type, and exhibit the behaviors of that type. Different objects can have different property values, and those values can effect their behaviors, but all objects of that type will have the properties of that type, and exhibit the behaviors of that type.

## What Is Data Encapsulation in Object-Oriented Programming?



**Data Encapsulation** means that each object “hides” its data from external access.

- An object typically prohibits direct external access to its data.
- Objects interact by “sending messages.”
- A message can request the receiving object to change a data value or reply with a data value.
- Programmers call this “calling a function.”



200 © Cadence Design Systems, Inc. All rights reserved.



**Data Encapsulation** means that each object “hides” its data from external access.

An object typically prohibits direct external access to its data.

In the object oriented paradigm, objects interact by “sending messages.”

A message can request the receiving object to change a data value or reply with a data value.

In SV we simply term this “calling a function.”

Here, we illustrate students interacting by sending messages.

## What Is a Class?



A class is a type that includes data and subroutines (functions and tasks) that operate on those data.

```
module cmod;
 class myclass;
 int number;
 endclass
...

```

Class Declaration in a module

```
package myclasses;
 class frame;
 logic [4:0] addr;
 logic [7:0] payload;
 bit parity;
 endclass
...

```

Class Declaration in a Package

- A class is a user-defined data type:
  - Declared in a module, package, interface or other design element
- Classes define data and subroutines (tasks/functions) that operate on the data.
- Class objects can be dynamically created and deleted during simulation.
- Used in Object-Oriented (OO) programming for testbenches and simulation models.
- OO features which are supported include the following:
  - Abstract data modeling
  - Inheritance
  - Data hiding and encapsulation
  - Polymorphism



A class is a user-defined type that includes data items of different types (like a `struct`) but also contains tasks/functions to access and manipulate the data items.

Variables of class types are dynamic, meaning they can be created and deleted during a simulation run, which makes them extremely useful for verification. For example, we can use class variables (handles) to create a random number of data items for stimulus, drive the data into the design, capture the output data and once the data is checked and verified, delete both input and output items to save memory space.

Classes allow the creation of generic, reusable objects that can be later extended, inherited, constrained, overridden, enabled, disabled, and merged with or separated from other objects.

A class declaration is like any other type declaration and must be placed inside a design element (module, program, interface or package).

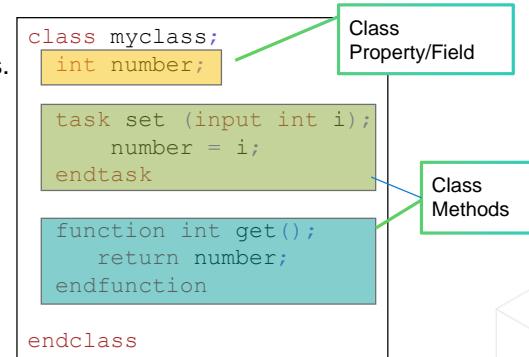
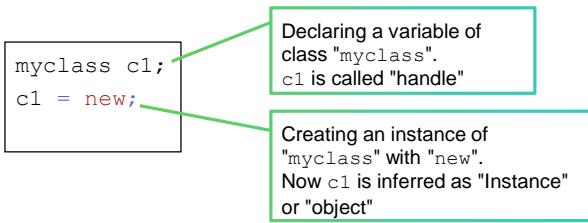
## What Is a Class Object?



A class defines a data type. An object is an instance of that class. An object is used by first declaring a variable of that class type (that holds an object handle) and then creating an object of that class (using the **new** function) and assigning it to the variable.

The class type declaration declares class members:

- Data items, called properties or fields.
- Tasks or functions, which operate on data items, called methods.

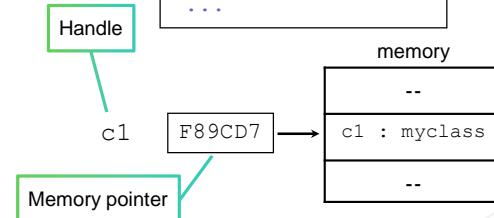


As we know, class is an user-defined data type, and can contain properties and methods as shown. An object is one instance of the class type. An object is an area of memory allocated for your program and associated with a type. In this example, we have a class called, "my class". It has a property called "number". It also has two methods. on the left side, you are declaring a variable C1 of type, "myclass". Now, C1 is called as handle. We cannot access class properties or methods with this handle, since we have not yet created an instance. In the next line, when you say, "new", then, this creates an instance to the class. we will look more into this in upcoming slides.

## Variables of the Class Type

- A variable of a class type is called a handle:
  - The uninitialized value is `null`.
- A class instance must be created for handle:
  - Using function `new`.
  - Assigns a pointer to an area of memory holding the class instance.
  - A handle can be declared and created at the same time:
  - Object persists until it either:
    - Is no longer used.
    - Is assigned `null`.
  - Automatic “garbage collection.”

```
module mone;
 class myclass;
 int number;
 endclass
 myclass c1;
```



203 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

A class is defined as a data type. The declaration contains data items (class properties) and optionally subroutines to manipulate properties (methods).

To use a class, you declare a variable of the class type, called a handle. The handle contains a pointer to a location in memory which holds the class data. An uninitialized handle has the value `null`. A class instance is created by calling a special class method called `new`, which allocates an area of memory for the class instance and places a pointer to that area in the class handle. The `new` method is called the class constructor.

A class instance can be declared and constructed in one line or the handle declared and the instance constructed later in a procedural statement.

In Object-Oriented languages such as C++, there is a corresponding deconstructor method, which must be called when the instance is no longer used. In such languages, memory management (construction of new instances and recycling of used instances) is solely under the user control.

In SystemVerilog, class memory management is handled automatically. There is no deconstructor method and the recycling of used instances is handled by the simulator. Once created, an instance will persist until either:

- The instance is no longer used. For example, if its handle has been used to create a new instance, or the region where the instance was created no longer exists. For example, for a instance declared in a task, the instance can be recycled when the task returns.
- All handles on the instance are explicitly assigned the value `null`. This is not necessary and rarely done.

## Comparing Classes and Objects



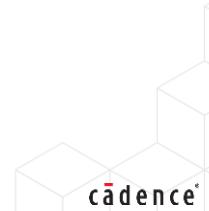
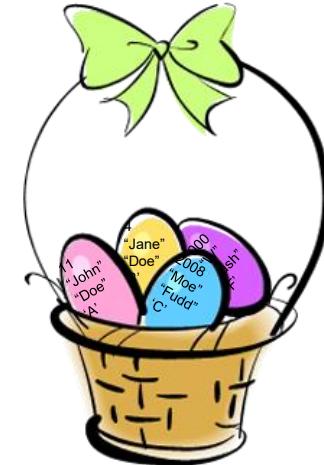
A class defines an entity (person, place or thing) type, having:

- Member data (attributes, properties, variables)
- Member behaviors (methods, functions)



An object is one instance of the class type.

```
class StudentRecord;
 int number;
 string grade;
 string name;
 string surname;
endclass
```



204 © Cadence Design Systems, Inc. All rights reserved.

A class defines an entity type, having:

- Member data, also termed *attributes* or *properties*.
- Member behaviors, termed *functions* or *tasks*, and also termed *methods*.

To declare a class, you can use keyword “**class**.”

- All members of a class are by default **public**, and you can encapsulate them if required.

An object is one instance of the class type.

An object is an area of memory allocated for your program and associated with a type. The example class “Student Record” declares several variables as properties.

Here, we illustrate a collection of objects of the Student Record class, each with the variables *number*, *grade*, and *name*. Where, each object is created by calling a constructor “new.”

## Accessing Object Members by Using "dot" (.) Operator

- Class members are accessed using "."
- Class properties can be accessed directly or via class methods.

Direct access

```
myclass c1 = new;
...
initial begin
 c1.number = 4;
 $display("c1: %0d", c1.number);
end
```

```
class myclass;
 int number;

 task set (input int i);
 number = i;
 endtask

 function int get();
 return number;
 endfunction

endclass
```

Method access

```
myclass c2 = new;
...
initial begin
 c2.set(3);
 $display("c2: %d", c2.get());
end
```



205 © Cadence Design Systems, Inc. All rights reserved.

The class type declaration contains the declaration of class members. Class members can be data items, called properties or fields. Only variable declarations are allowed as class properties; net data types are not allowed. Class members can also be tasks or functions, called methods, which manipulate the class properties. You reference the members of a class instance by using the “dot” (.) notation.

By default, you have full and unrestricted access to all members of a class. We will look at restricting access later in this module.

# Quick Reference Guide: C++ Object Pointers and SystemVerilog Object Handles



| Operation                                 | C++ Pointer | SystemVerilog Object Handle |
|-------------------------------------------|-------------|-----------------------------|
| Arithmetic operations, e.g., incrementing | allowed     | not allowed                 |
| For arbitrary data types                  | allowed     | not allowed                 |
| Assignment to an address of a data type   | allowed     | not allowed                 |
| Access via a null pointer                 | error       | error*                      |
| Casting                                   | unlimited   | limited                     |
| Default value                             | undefined   | null                        |
| Memory management                         | manual      | automatic                   |

SystemVerilog restricts what you can do with class handles.

- This is to reduce the likelihood of user error.

**Note:** SystemVerilog performs automatic memory management.



Compared to other Object-Oriented languages such as C++, SystemVerilog heavily restricts what you can do with a class handle. Things you cannot do include:

- Arithmetic operations, for example incrementing and decrementing
- Using with any arbitrary data type
- Assigning the address of a variable
- Dereferencing
- Casting to any arbitrary pointer type

A simpler class handle implementation leads to fewer issues and errors in using classes.

A key benefit of the SystemVerilog Object-Oriented implementation is that memory management is automatic. The simulator is responsible for recycling used class instances and for preventing memory leaks – where class instances are repeatedly created without being recycled, leading to a gradual reduction in free system memory and performance, and an eventual software crash.

SystemVerilog is closer to modern Object-Oriented languages such as Java in this respect.

## External Method Declaration



Purely for convenience and readability, you can declare the implementation of a class method outside of the class declaration using the `extern` keyword.

- Define the methods outside the class declaration.
- Define the method prototype in the class, prefixed by the keyword `extern`:
  - First line of method, identifying type, name and arguments
- Implement the method outside the class declaration, but in same scope.
  - Implementation must exactly match prototype
  - Link to prototype using the scope resolution operator (`::`):  
`class_name::method_name`

```
class myclass;
 int number;

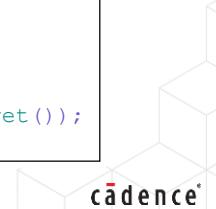
 task set (input int i);
 number = i;
 endtask

 extern function int get();
endclass

function int myclass::get();
 return number;
endfunction
```

```
myclass c2 = new;

initial begin
 c2.set(3);
 $display("c2: %d", c2.get());
end
```



Purely for convenience and readability, you can declare the implementation of a class method outside of the class declaration using the `extern` keyword.

This gives the class user a quick, compact description of the class properties and methods, uncluttered by the full implementation of every method.

To use external method declarations, you declare the method prototype, prefixed with the keyword `extern`, in the class declaration. The method prototype is the first line of the method declaration, without the method code body. This identifies the type of method (function or task), the return type (for a function), the method name and the arguments. The full method description, including the code body, is then declared outside the class declaration. However, as the method is a class member, not a subroutine of the external scope, you must link its declaration to the class by prefixing the method name with the class name, using the scope resolution operator.

External (or out-of-block) method declarations are purely for readability and have no effect on simulation.

## What Is a Constructor Method?



The `new` function which is used to create a class instance and to initialize an instance is called the `class constructor`.

Default constructor

```
class defcon;
 integer number;
endclass

defcon c1 = new;
// c1.number = x
```

Explicit constructor

```
class expcon;
 integer number;

 function new();
 number = 5;
 endfunction

endclass

expcon c2 = new;
// c2.number = 5
```

Method `new` is a special class method called a constructor:

- Defined by default for all classes.
- You can explicitly define it, for example, to initialize class properties.
- Just like any other class function except `new` has no return type.

```
class argcon;
 integer number;

 function new(input int ai);
 number = ai;
 endfunction

endclass

argcon c3 = new(3);
// c3.number = 3
```

Explicit constructor with arguments

208 © Cadence Design Systems, Inc. All rights reserved.



By default, class properties initialize to their default values, e.g., `x` for 4-state logic types, `0` for 2-state logic types, etc.

An explicit constructor can be implemented to initialize properties as required. An explicit constructor is defined as a function named `new` in the class declaration. When you call the `new` constructor to create a class instance, the `new` function in the class declaration is automatically called. The code body of the `new` function can execute whatever code is required, for example, initialize class properties to required values. The function does not have a return type (this is derived from the class declaration), but otherwise follows all the normal rules of a SystemVerilog function. For example, the constructor can have multiple input arguments with default values, if required.

A SystemVerilog class declaration can only contain a single explicit constructor. SystemVerilog does not support subroutine overloading, which would allow multiple declarations of a task or function with the same name (e.g., `new`) but different numbers or types of arguments.

## Example: Class Definition and Instantiation

```

class frame;
 logic [4:0] addr;
 logic [7:0] payload;
 logic parity = 0;

 function new (input int add, dat);
 addr = add;
 payload = dat;
 genpar();
 endfunction

 function void genpar();
 parity = ^{addr, payload};
 endfunction

 function logic [13:0] getframe();
 return ({addr, payload, parity});
 endfunction

endclass

```

Note the following:

- Initial property value
- Constructor arguments are assigned to properties
- Constructor calls method to get updated parity value

```

logic [13:0] framedata;
frame one = new(3, 16);

initial begin
 ...
 @(negedge clk);
 framedata = one.getframe();
 ...
end

```

209 © Cadence Design Systems, Inc. All rights reserved.



Here is a class example of a `frame` object with three properties and three methods (including an explicit constructor).

Class properties can be given initial values, just like module variables. In this example, the initial value on `parity` is redundant because the parity is updated in the constructor.

The `addr` and `payload` properties are initialized in an explicit constructor using the `add` and `dat` arguments. In the constructor, we also call the `genpar()` method to update the `parity` property to be consistent with the `addr` and `payload` values. The parity generation is defined as a separate method as this will be a common operation to be executed on the `frame` instances. In the declaration of `genpar()`, the return type of the function is `void` and no arguments are passed to the method. As the function is declared inside the class, local to the properties it needs to access, then, by convention, the properties are accessed directly and are not passed as method arguments.

The other class method, `getframe()`, is a “packing” method in that it packs all the properties of the class into a single vector for passing into the design.

## Current Object Handle: `this`



IEEE 1800-2012 8.11

- Keyword `this` is a handle to the current class instance.
  - You can use it to reference class identifiers re-declared within a local scope.
  - It is meaningful only for nonstatic members.
- Here, `this.addr` is used to access the class property `addr`.
  - Otherwise, it is hidden by the function argument `addr`.
- Alternatively, do not reuse class property names as class method arguments.

```

class frame;
 logic [4:0] addr;
 logic [7:0] payload;
 logic parity;
 ...
 function new(input int addr, dat);
 this.addr = addr;
 payload = dat;
 genpar();
 ...
 endfunction
endclass

```

210 © Cadence Design Systems, Inc. All rights reserved.



The `this` keyword is a handle to the current class instance.

In the example, in the constructor of `frame`, we have renamed the input argument `addr` to have the same name as the class property `addr`. Therefore in the constructor, any reference to `addr` accesses the function argument only, and the class property `addr` is hidden from view. By prefixing the identifier `addr` with the keyword `this`, we point the reference outside the constructor to the current class instance, and hence we can access the class property `addr`.

Alternatively, we could avoid using the same identifier for both the function argument and class property, and the need for `this` goes away.

In simple SystemVerilog class applications, `this` is rarely required, but in more complex class environments, such as in UVM (Universal Verification Methodology), the `this` reference is extensively used, specifically for class hierarchies using aggregate classes.

## What Are Static Class Properties?



The Static Class Properties are class properties which are created using the keyword "static".

```
class frame;
 static int frmcount;
 int tag;
 logic [4:0] addr;
 logic [7:0] payload;
 logic parity;

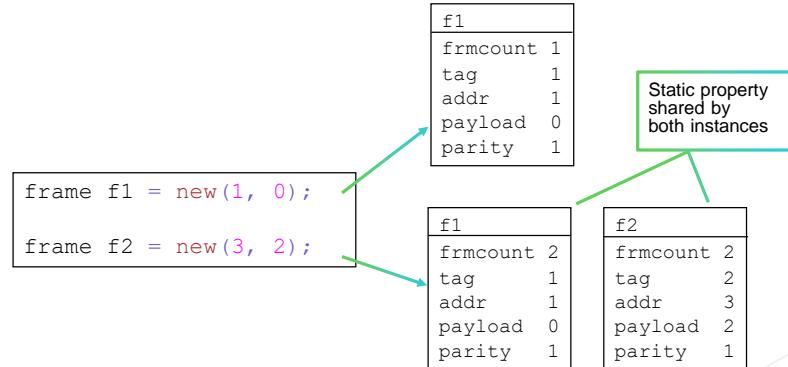
 function new(input int add, dat);
 addr = add;
 payload = dat;
 genpar();
 frmcount++;
 tag = frmcount;
 endfunction

 ...
endclass
```

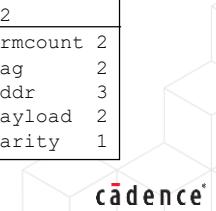
A static property is shared by all instances of a class.

In the example, it is used to:

- Count the number of frames created.
- Create a unique identity for each.



211 © Cadence Design Systems, Inc. All rights reserved.



Normally, each class instance has a separate, unique copy of every class property.

However, a property can be defined as `static`, which means that only one copy of the property exists and this copy is shared among all instances of the class. An update to a static property from one instance updates the property for all instances.

Here a static property is used to keep track of the number of frames created, and also to assign a unique identity to each frame. The static property `frmcount` is incremented in the class constructor and assigned to a nonstatic property `tag`.

When frame instance `f1` is created, `frmcount` is incremented to 1 and assigned to `f1.tag`. When instance `f2` is created, `frmcount` is incremented to 2, which sets both `f1.frmcount` and `f2.frmcount` to 2, and `f2.tag` is assigned to 2.

Static property `frmcount` initializes to 0 (being of type `int`), but could also be explicitly initialized in the declaration:

```
static int frmcount = 0;
```

Note that we do not want to reset `frmcount` in the class constructor. The aim is to keep an independent count of the number of frames created, so every time we create a frame, we want to increment the current frame count, not set it to zero.

## What Are Static Class Methods?



When Methods are declared as `static`, they are subject to all the class scoping and access rules but behave like a regular subroutine that can be called outside the class, even with no class instantiation.

```
class frame;
 static int frmcount;
 int tag;
 logic [4:0] addr;
 logic [7:0] payload;
 logic parity;

 function new(input int add, dat);
 addr = add;
 payload = dat;
 genpar();
 frmcount++;
 tag = frmcount;
 endfunction

 static function int getcount();
 return (frmcount);
 endfunction
 ...
endclass
```

Only static properties or other static methods can be accessed.

Can be called even if no class instantiations exist.

- From class name using resolution operator `::`:
- From any class handle.

```
frame f1, f2;
int frames;
initial begin
 frames = frame::getcount(); // 0
 frames = f2.getcount(); // 0

 f1 = new(3,4);
 f2 = new(5,6);
 frames = f2.getcount(); // 2
end
```

Resolution operator access

Handle access

212 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

A static method knows nothing about the current object and so can access only static properties or other static methods.

The advantage of a static method is that it can be called even if there are no current instances of the class. A static method can be accessed from any handle of the class, regardless of whether the handle contains an instance, using normal method access or from the class name using the resolution operator (`::`):

`<class_name>::<static_method>`

This example uses a static function to return the current frame count, which is a static property.

Some coding guidelines mandate the calling of static methods using resolution operator access from the class type name. This serves to clearly indicate the method is static, and aids readability. For example, at first glance the `getcount()` call from `f2` at the beginning of the initial block could be interpreted as a nonstatic method call from an object instance. The resolution operator syntax is unambiguous as a static method call.

## What Is Class Aggregation?



Aggregation refers to a class which collects (aggregates) a number of other classes into one object.

```
class frame;
 logic [4:0] addr;
 logic [7:0] payload;
 bit parity;

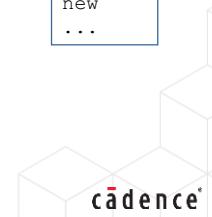
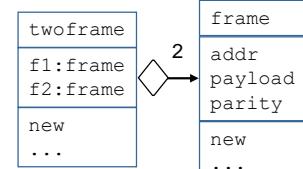
 function new(input int add, dat);
 addr = add;
 payload = dat;
 genpar();
 ...
 endfunction
...
endclass
```

```
class twoframe;
 frame f1;
 frame f2;

 function new(input int basea, d1, d2);
 f1 = new(basea, d1);
 f2 = new(basea+1, d2);
 endfunction
...
```

```
twoframe tf1 = new(2,3,4);
initial begin
 tf1.f2.addr = 4;
 $display("base %h", tf1.f1.addr);
...
```

- A class property can be an instance of another class.
  - Creates an aggregate or composite class.
  - Relationship “has a”.
  - Similar to module instantiation.
- Constructors of class properties must be explicitly called.
- Instance handles must be chained to reach into hierarchy.



213 © Cadence Design Systems, Inc. All rights reserved.

It is common for class objects to be building blocks for other classes. One way to achieve this is to declare class properties which are themselves handles of other classes. This is known as an aggregate or composite class (an aggregation, or collection, of other class instances).

We can use this to create a class which hold multiple frames with consecutive addresses.

Here class `twoframe` contains two object handles (`f1`, `f2`) of the class `frame` as properties. The constructor for `twoframe` should create the instances of `frame` by a call to the `frame` constructor. Without this call, the `frame` object handles in `twoframe` will be `null`. When we create an instance `tf1` of `twoframe`, the constructor creates the instances `f1` and `f2` of `frame` within the `tf1` instance.

To access the properties of `f1` within `tf1`, we must chain handles:

`tf1.f1.addr`

This accesses the property `addr` of the `frame` instance `f1`, which is a property of the `twoframe` instance `tf1`.

This is similar to a module hierarchy in Verilog. We can use a hierarchical pathname to access variables in a module-based hierarchy; likewise, we can use an handle pathname to access members of a class-based hierarchy.

## What Is Class Inheritance?



**Class Inheritance** is extending a class by defining a derived class that inherits members<sup>†</sup> of the base class and may override inherited members and/or add new members.

<sup>†</sup> Does not inherit constructors.

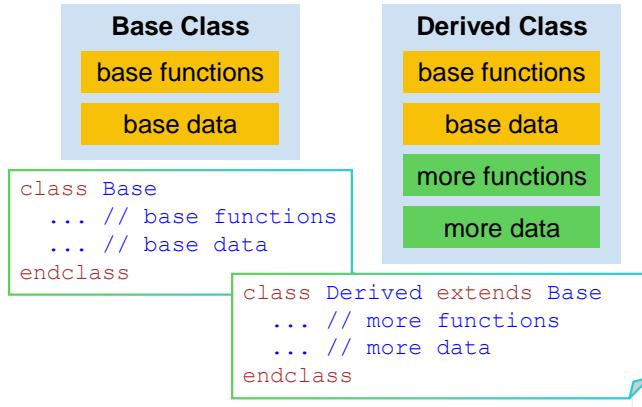
Why inheritance?

- Reusability makes the user more productive.
- Encourages use of proven software (why re-invent the basic wheel?).

Terminology

- Original class is called the superclass (OO) or base class (SV).
- New class is called the subclass (OO/SV) or derived class .
- SV Provides only single inheritance: that is each subclass is derived from a single base class.
- Relationship “is a”.

214 © Cadence Design Systems, Inc. All rights reserved.



**Class Inheritance** is extending a class by defining a derived class that inherits members of the base class and may override inherited members and may add new members.

Inheritance is essential to the object-oriented paradigm. Each initial description or refinement of the class is defined in one place and easily reused by further refinements to the class.

In an inheritance hierarchy, an inherited class is a “superclass” in object-oriented terms and a “base” class in SV terms, and the inheriting class is a “subclass” in object-oriented terms and a “derived” class in other language terms. Base classes and derived classes are relative terms, as a derived class can be a base for a further derivation.

SystemVerilog supports only single inheritance. That is each class can be derived from a single base class.

## Example: Simple Inheritance

A sub-class inherits all the members of its parent class:

- Can add more members.
- Can re-declare (override) parent's methods.
- Constructors are not inherited.

Parent members are accessed as if they were members of the sub-class:

The sub-class constructor automatically calls the parent constructor:

- `frame.new()` called as first line of `badtagframe` constructor.
- Why is this a problem?

```
badtagframe one = new(); //???
one.addr = 0;
```

```
class frame;
 logic [4:0] addr;
 logic [7:0] payload;
 bit parity;

 function new(input int add, dat);
 addr = add;
 payload = dat;
 genpar();
 endfunction

 ...
endclass
```

Base-class

```
class badtagframe extends frame;
 static int frmcount;
 int tag;

 function new();
 frmcount++;
 tag = frmcount;
 endfunction

 ...
endclass
```

Sub-class



Error

215 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

An instance of the sub-class inherits all the members of its parent class and, by default, they can be referenced as if they were members of the sub-class.

A sub-class can also declare additional members and even override members from the parent class. An instance of the sub-class will, by default, always access its local overridden member declarations (if present) in preference to those from the parent class.

Hence using classes as building blocks for more specialized or generic classes is very easy.

The sub-class can include an explicit or default constructor, but the sub-class constructor always calls the parent class constructor as the first line. This is to make sure the properties inherited from the parent class are correctly initialized before the sub-class properties are processed. For example, in the code above, the compiler automatically inserts a call to the `frame` constructor *as the first line* of the `badtagframe` constructor. If you are using default constructors, this is no problem, but, in our example, the `frame` class has an explicit constructor with arguments which do not have default values. The call to the parent constructor inserted by the inheritance mechanism lacks the arguments required and so a compilation error is reported.

## Example: Inheritance and Constructors

```

class frame;
 logic [4:0] addr;
 logic [7:0] payload;
 bit parity;

 function new(input int add, dat);
 addr = add;
 payload = dat;
 genpar();
 endfunction
 ...
endclass

```

Parent

```

class badtagframe extends frame;
 ...
 function new();
 super.new(); // Error
 frmcount++;
 tag = frmcount;
 endfunction
 ...

```

Sub-class

**Automatically inserted**

**Error**

The frame constructor requires arguments:

- Must be explicitly called to pass these arguments.
- Must be the first line of sub-class constructor to overwrite the implicit call.

`super` allows a sub-class to access parent members:

- Otherwise hidden by sub-class declarations.

```

class goodtagframe extends frame;
 ...
 function new(input int add, dat);
 super.new(add, dat); // First line explicit call overwrites implicit call
 frmcount++;
 tag = frmcount;
 endfunction
 ...

```

Sub-class

In this example, the frame constructor requires arguments to set the initial values of `addr` and `payload`. The implicit constructor call automatically inserted by the inheritance mechanism (derived from the `extends`) does not have these arguments, and so a compilation error results.

By explicitly adding the constructor call, we can overwrite the implicit call and pass the required arguments up to the `frame` constructor. However, in the `goodtagframe` class constructor, any reference to `new` would access the local constructor, creating a recursive function call. Nor can we use `this` as it would point to the `goodtagframe` instance. The keyword `super` allows us to access any member of the parent class which is otherwise hidden by being overridden in the sub-class. Therefore `super.new(add, dat)` allows us to call the `frame` constructor and pass the required arguments. Note that the `add` and `dat` arguments must be added to the `goodtagframe` constructor and passed up to the `frame` constructor in the `super` call.

An explicit `super.new` call must be the first line of the sub-class constructor. If we do not make it the first line, an implicit constructor call is added automatically. This means that the sub-class constructor cannot process any arguments before they are passed up to the parent class constructor.



## Quick Reference Guide: Aggregation and Inheritance

|                                                                                                                                                                                                                                                                                                                          |                                                              |                                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Aggregation</b> <ul style="list-style-type: none"> <li>A class has properties which are instances of other classes:           <ul style="list-style-type: none"> <li>Relationship “has a”</li> <li>Similar to module instantiation</li> </ul> </li> <li>Each handle must be explicitly constructed.</li> </ul>        | <pre> twoframe f1:frame f2:frame new ...   </pre>            | <pre> frame addr payload parity new ...   </pre> <p>2</p>                                                                                                                              |
| <b>Inheritance</b> <ul style="list-style-type: none"> <li>Define a new sub-class by extending a parent class:           <ul style="list-style-type: none"> <li>Relationship “is a”</li> <li>Similar to `include</li> </ul> </li> <li>A sub-class inherits all members of the parent class except constructor.</li> </ul> | <pre> Parent frame addr payload parity new getframe   </pre> | <pre> Sub-class tagframe frmcount tag new getcount   </pre> <p>=</p> <pre> Conceptual equivalent of sub-class tagframe addr payload parity frmcount tag new getcount getframe   </pre> |

217 © Cadence Design Systems, Inc. All rights reserved.

Aggregation refers to a class which collects (aggregates) a number of other classes into one object. There is no limit on the number of classes which can be aggregated, but each object handle must be explicitly constructed. This is referred to as a “has a” relationship, in that the class `twoframe` *has a* property `f1`, which is an instance of class `frame` and also *has a* property `f2`, which is also an instance of `frame`. In traditional Verilog, this is similar to module instantiation. Any instance of class `twoframe` instantiates two instances of `frame` `f1` and `f2` and to access the frame instances of `twoframe`, we use a hierarchical pathname.

Inheritance is a different way of using classes as building blocks, and is the basis of object-oriented design. Inheritance allows a class (sub-class) to extend another class (parent). We can use this to separate the tag functionality of the frame class from the packet properties. The `tagframe` class extends `frame`. The sub-class inherits all the base class members (addr, parity etc.) and can add new members (frmcount, tag) or override existing members. This is referred to as a “is a” relationship in that the sub-class `tagframe` *is a* `frame` class, albeit with additional members. In traditional Verilog, this is similar to an `include directive in that the declarations from the parent class appear as if they were declared in the sub-class.

Crucially, the sub-class is still compatible with the base class and handles of both classes can be mixed. SystemVerilog only allows single inheritance in that a sub-class can only extend from one base class.

The base class is also called the super class. The sub-class is also called the derived class or child class.

# Multi-Layer Inheritance



You can layer inheritance to multiple generations. Each new level inherits the members of the previous levels and can override any of these members and can add new members.

`errframe` is a sub-class of `tagframe`.

- Induces parity errors.
  - Has access to `tagframe` and `frame` members.
  - Constructors are explicit and cascaded.

You can only pass arguments one level at a time:

- `super.super.new()` is not allowed.

```
class frame;
 logic [4:0] addr;
 logic [7:0] payload;
 bit parity;

 function new(int add, int dat);
 ...
 endfunction

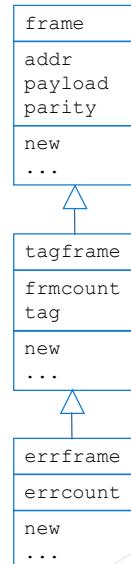
 class tagframe extends frame;
 static int frmcount;
 int tag;

 function new(input int add, dat);
 super.new(add,dat);
 ...
 endfunction

 class errframe extends tagframe;
 static int errcount;

 function new(input int add, dat);
 super.new(add,dat);
 endfunction

 ...
 endclass
 endclass
endclass
```



218 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Inheritance can, and often is, layered to many levels. Each new level inherits the members of the previous levels and can override any of these members and can add new members.

In this example, `errframe` is a specialized version of `tagframe` which inserts errors in the frame data. It has its own methods and a static property `errcount`.

`tagframe` is a parent class of `errframe` and a sub-class of `frame`. `frame` is the parent class of `tagframe`. The highest parent class at the base of the inheritance hierarchy is referred to as the base class.

Because the base class constructor is parameterized, we must explicitly define constructors for each subclass and pass the parameters up the inheritance hierarchy. Note that `super` refers to the immediate parent class only. There is no way to reach higher (`super.super.new` is not allowed).

## What Is Data Hiding and Encapsulation?



To restrict the access to class properties and methods from outside the class by hiding their names, System Verilog (SV) provides local and protected identifiers.

```
class frame;
 local logic [4:0] addr;
 local logic [7:0] payload;
 protected bit parity;

 fun
 class tagframe extends frame;
 local static int frmcount;
 int tag;
 function r
 super.n ...
 ...
 endfunction

 static function int geterr();
 return (errcount);
 endfunction
 ...
 endclass
endcl
```

Parent class

Sub-class

Sub-class

By default, class members are visible externally and in all sub-classes.

Two keywords hide members:

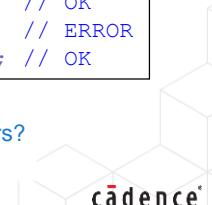
- **local** – Only visible inside the class.
- **protected** – Visible inside class and any sub-classes.
  - That is, can be inherited.

```
errframe one = new(addr1, data1);
initial begin
 one.errcount = 0; // ERROR
 one.parity = 1; // ERROR
 one.add_error(); // OK
 no_errs = one.errcount; // ERROR
 no_errs = errframe::geterr(); // OK
```

Question



Which of the lines above generate errors?



Remember, **protected** members are inherited, whereas **local** members are not.

## What Are Parameterized Classes?



The SystemVerilog parameter mechanism is used to parameterize a class. It allows user to define a generic class whose objects can be instantiated to have different array sizes or data types.

```
class stack #(parameter type sign = int);
 local sign data[100];
 static int depth;

 task push(input sign indat...);
 ...
 endtask
 task pop(output sign outdat ...);
 ...
 endtask
endclass

// int stack (default)
stack intstack = new();

// 8-bit vector stack
stack #(logic[7:0]) bytestack = new();
```

- Classes can have parameters just like modules and interfaces.
- Parameters can be types as well as values.
  - A type parameter creates a "class template."
- Parameter values can be overridden for individual instances.
  - Methods using type parameters must work for all expected type overrides.
- Each new parameter value effectively creates a new class declaration or class specialization.
  - Each class specialization has a separate set of static properties.

220 © Cadence Design Systems, Inc. All rights reserved.



Classes can have parameters, just like modules and interfaces, and these parameters can be mapped to different values for each instance of a class. For example, this would allow the user to change the size of an array property for each instance of a class.

Verilog-2001 allows type parameters, and type parameters can also be used with classes. A class with a type parameter is known as a template class, and typically defines some functionality (here a stack) which can be carried out on many different types. An issue with template classes is defining methods so they work with all combinations of expected types, including SystemVerilog types such as `struct` and `enum`.

Each new value or type for a parameter effectively creates a new class declaration (or class specialization). One consequence is that each class specialization has a separate set of static properties.

Note that there are alternative, and better, constructs in SystemVerilog for implementing stack-like functionality, such as a queue dynamic array, where stack elements are of the same type, or a mailbox where the elements can be of different types.

## Module Summary

With the SystemVerilog Object-Oriented testbench features, you can:

- Declare dynamic data types:
  - Create and destroy objects during simulation
- Declare hierarchical data types:
  - Building up “layers” of types promotes reuse
- Encapsulate data with protected and local class members:
  - Hiding your data from external methods simplifies maintenance of the type
- Raise the level of abstraction for building testbenches:
  - You can develop and debug the test environment more quickly

Other SystemVerilog features are also “class-like”:

- Random constraints
- Covergroups



*This page does not contain notes.*

## Quiz



What is the role of a class constructor?



What is the difference between a static and non-static property?



What is the difference between a static and non-static method?



What is inheritance?



What is the difference between a protected and a local class member?

- The class constructor creates an instance and initializes the class properties.
- One instance of a nonstatic property exists for each instance of the same class type.
- One instance of a static property exists for all instances of the same class type.
- A static method of a class can access only the static members of the class.
- A sub-class inherits the members of its super class, and can add new members and re-declare those of the parent class.
- A sub-class method can access a protected member of its super class.
- Only class member methods can access local members of a class.

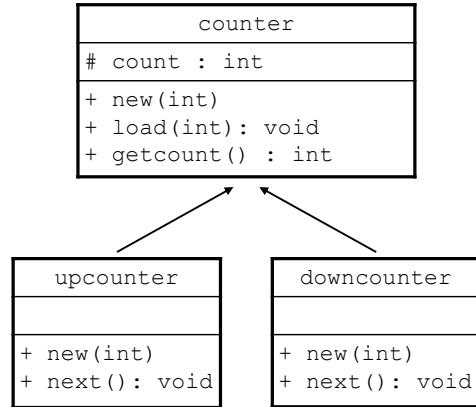
*This page does not contain notes.*



# Lab

## Lab 11 Using Classes

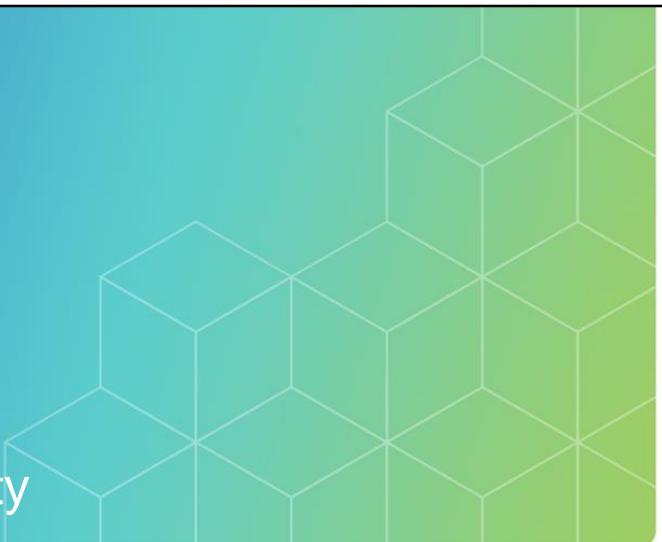
- Create a simple class using user-defined constructors and methods, and explore inheritance and aggregate classes.



223 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*



## Polymorphism and Virtuality

**Module** **15**

Revision

**1.0**

Version

**21.10**

Estimated time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

- Analyze the basic concepts of polymorphism
- Use casting to copy class instances between handles
- Analyze polymorphic class member resolution
- Create virtual methods and virtual method prototypes
- Evaluate the role of virtual classes



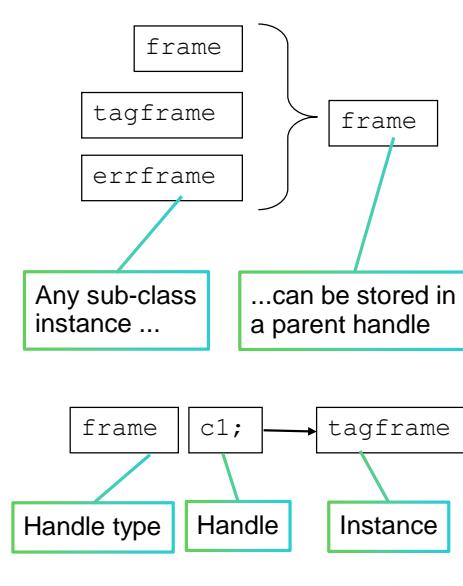
*This page does not contain notes.*

## What Is Polymorphism?



Polymorphism allows the use of a variable of the superclass type to hold subclass objects and to reference the methods of those subclasses directly from the superclass variable.

- An inherited class creates a new type.
  - You need to easily change between sub-class types.
  - **Example:** Switch to `tagframe` without updating the type of all handles.
- A class handle of a given type can be assigned any class extension instance.
  - Polymorphism.
  - This introduces the concept of a handle type.
  - Class type is used in declaration.
- It aims to look at the contents of the handle.
  - Class instance held in the handle.



226 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Every sub-class declaration defines a new type e.g., `frame`, `tagframe` and `errframe` are all different type declarations in SystemVerilog. However, these types are closely related to each other (having a common base type of `frame`) and we need to easily change between different sub-class types without having to change the type of our class handles. A key feature of object-oriented design is the ability to assign to a class handle of a specific type, any instance of a sub-class of the handle type i.e., we can create our design using object handles of type `frame`, and then assign any instance of the sub-classes of `frame` to that handle. This is polymorphism.

This introduces the concept that a handle has a declaration type e.g., `frame`, but its contents – the instance to which the handle points – may be of a sub-class of the handle type e.g., `tagframe`.

## How to

### Copy a Sub-Class Instance to a Parent Handle



A sub-class instance can always be directly copied to a parent class handle.

However, by default, only parent class members are accessible from the handle:

- Even though it contains a sub-class instance

1. Create an instance of sub-class

2. Copy tagframe instance to frame handle

Only parent (frame) method visible

Sub-class (tagframe) property not visible

```
frame f1;
tagframe t1 = new(...);

initial begin
 f1 = t1;
 f1.iam(); // frame
 f1.tag = 5;
 ...

```

```
class frame;
 ...
 function void iam();
 $display ("frame");
 endfunction
 ...
endclass

class tagframe extends frame;
 ...
 int tag;

 function void iam();
 $display ("tagframe");
 endfunction
endclass
```



In polymorphism, we can always directly copy a sub-class instance to a handle of a parent class type.

Here, we create an instance of `tagframe` in the handle `t1`, and copy the instance to the `frame` handle `f1`. As `frame` is a parent class of `tagframe`, the assignment is valid. Remember that both `f1` and `t1` are handle pointers to memory addresses where the instances are actually stored. Therefore, when we copy `t1` to `f1`, we are overwriting the current contents of `f1` (if any) with the address stored in `t1`, which is a pointer to an instance of `tagframe`. This is called a reference copy.

Now `f1` has a handle type of `frame`, but contains (points to) an instance of `tagframe`. However, when we try to access class members from `f1`, by default we can only see members of the handle type class, i.e., `frame`. Calling the overridden `iam()` method from `f1` executes the function defined in the `frame` class, even though `f1` contains an instance of `tagframe`. Likewise, trying to access additional properties of `tagframe`, such as `tag`, reports errors as all member access is resolved by reference to the handle type (`frame`).

## How to

### Copy a Parent Instance to a Sub-Class Handle



It is never legal to directly assign a parent handle to a handle of one of its sub-classes. It is only legal to assign a parent handle to a sub-class handle, if the parent handle contains an instance of the given sub-class.

1. A parent-class instance cannot be copied to a sub-class handle:
  - Unless the parent class handle contains a sub-class instance
2. Requires use of \$cast:
  - Checks that the parent handle contains a sub-class instance

```
frame f1;
tagframe t1 = new(...);
tagframe t2;

initial begin
 f1 = t1;
 f1.iam(); // frame
 $cast(t2, f1);
 t2.iam(); // tagframe
```

```
class frame;
 ...
 function void iam();
 $display ("frame");
 endfunction
 ...
endclass

class tagframe extends frame;
 ...
 function void iam();
 $display ("tagframe");
 endfunction
endclass
```

1. Copy tagframe instance to frame handle

Only parent (frame) method visible

2. Copy from f1 to t2, checking with \$cast

Sub-class (tagframe) method now visible

228 © Cadence Design Systems, Inc. All rights reserved.



One way to access the members of a sub-class instance copied to a parent class handle is to copy the sub-class instance back into a sub-class handle. However, although it is always legal to assign a sub-class instance to a handle of a class higher in the inheritance tree, it is never legal to directly assign a parent handle to a handle of one of its sub-classes. It is only legal to assign a parent handle to a sub-class handle if the parent handle contains an instance of the given sub-class. To check whether the assignment is legal, a dynamic cast must be used.

\$cast is called with two arguments – the first is the target handle for the assignment, and the second is the source handle. \$cast checks whether the contents of the source are compatible with the handle type of the target, and if so, it makes the assignment from source to target.

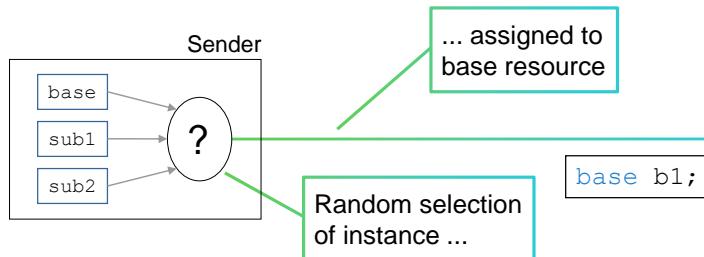
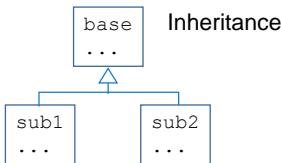
Once the sub-class instance is back in a sub-class handle, the correct members can be accessed.

Here we copy an instance of tagframe into the frame handle f1. When we access the method iam() from f1, the call is directed to the handle class type frame. Using the cast, we can copy the contents of f1 to the handle t2. The cast checks that the contents of f1 (an instance of tagframe) are compatible with the tagframe handle t2. The check is successful and so the assignment completes. We can now call the iam() method from t2 and access the tagframe implementation.

## Using \$cast:



IEEE 1800-2012 8.16



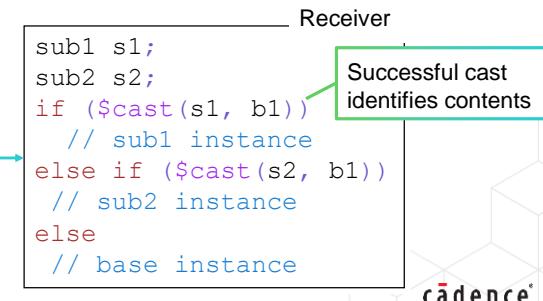
Casting lets sub-class instances use resources defined for parent classes:

\$cast is actually a subroutine:

- Defined as both function and task

Syntax:

- \$cast(destination, source)**
- If the source does not contain a matching instance for the destination:
  - Task gives a runtime error
  - Function returns 0



229 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Casting allows us to use a base or parent class resource, such as a handle, with any instance of a sub-class of the base or parent type. We can assign the sub-class instance directly to the base class resource, and then extract the sub-class instance back into a sub-class handle using the cast.

In the example, we have a resource `b1` (e.g., a handle) of the `base` class type. A Sender randomly creates an instance of a `base`, `sub1` or `sub2` class and assigns it to `b1`. At the receiver, we do not know which instance has been sent, but by casting to a handle of each possible sub-class, we can either identify the sub-class instance or deduce it is a base instance if both casts fail.

The syntax for `$cast()` is as follows:

```

task $cast(destination_handle, source_handle);
function int $cast(destination_handle, source_handle);

```

The function form is preferable because if the cast fails, we can detect the failure and act accordingly. The task form gives a runtime error with no possibility of recovery or debug.

The cast will be successful if the destination is the same type or a superclass (parent) of the source type.

## Advantages of Polymorphism

Polymorphism usage:

- A base class is used for a handle array declaration.
- Different sub-class instances are then assigned to array elements.
- We can dynamically select which sub-class instance to load into an array.

Advantages:

- The type of frame is decided at the start of stimulus code.
- All subsequent references can be made to the base array variable.
- More sub-classes can be easily added to the design.

```
frame framearr[7:0];
tagframe tf;
errframe ef;

initial begin
 foreach (framearr[i])
 randcase
 2 : begin
 tf = new(0,0);
 framearr[i] = tf;
 end
 1 : begin
 ef = new(0,0));
 framearr[i] = ef;
 end
 ...
 endcase
 // Complete frame data
 // Send frame to DUT
```

230 © Cadence Design Systems, Inc. All rights reserved.



Polymorphism allows a variable to be assigned different classes at different times, and for operations on that variable to be dependent on the current class it is assigned.

The variable `framearr` is declared as an array of type `frame`. The testbench is written using the array of `frame` handles. `framearr` can then be assigned from any sub-class of `frame`, dynamically during simulation.

Polymorphism is an essential part of object-oriented design. It allows a basic, generic coding infrastructure to be written using base class handles which can later be easily extended, inherited, overridden, enabled, disabled, and generally modified to suit requirements.

## Class Member Access in Polymorphism



**Problem:** By Default, resolution of class member access is always according to the type of handle, not its contents.

**Solution:** Using Virtual Methods (next slide)

```
class base;
 function void iam();
 $display ("Base");
 endfunction
endclass

class parent extends base;
 function void iam();
 $display ("Parent");
 endfunction
endclass

class child extends parent;
 function void iam();
 $display ("Child");
 endfunction
endclass
```

- Class members are resolved by searching from the class handle type:
  - Through the inheritance hierarchy.
  - Even if the handle contains a sub-class instance.
- This blocks access to sub-class members for a sub-class instance in a parent class handle.

```
base b1;
parent p1 = new();
child c1 = new();

initial begin
 b1 = p1;
 b1.iam(); //Base
 p1 = c1;
 p1.iam(); //Parent
 ...

```

Handle type = base

Handle type = parent

231 © Cadence Design Systems, Inc. All rights reserved.



When you copy a sub-class instance into a parent class handle, then any references to class members are resolved by checking the declaration of the handle class type.

Here we create an instance of the parent class and copy it into the base class handle b1. When we call b1.iam(), the method call is resolved by looking in the class type of b1 i.e., the base class. So b1.iam() returns "Base" even though b1 contains a parent instance.

Likewise, when we copy an instance of child into the parent handle p1, the call p1.iam() is resolved by looking in the parent class. If iam() was not declared in class parent, then the base class would be checked (as parent extends base). If iam() was not declared in base, the call would fail to compile, even if iam() was declared in child.

By default, resolution of class member access is always according to the type of a handle, not its contents.

This can be a serious limitation, so there must be a solution. One solution is to use casting to copy sub-class instances back into sub-class handles, but this requires much modification to code when using sub-classes. A better solution is to use virtual methods.

## Solution: Using Virtual Methods

```

class base;
 virtual function void iam();
 $display ("Base");
 endfunction
endclass

class parent extends base;
 virtual function void iam();
 $display ("Parent");
 endfunction
endclass

class child extends parent;
 virtual function void iam();
 $display ("Child");
 endfunction
endclass

```

virtual keyword optional

- Virtual methods direct method resolution to the *contents* of a handle.
- They allow access to methods of a sub-class instance when held in a parent class handle.
- When a method is declared as virtual, it is automatically virtual in every sub-class.

```

base b1;
parent p1 = new();
child c1 = new();

initial begin
 b1 = p1;
 b1.iam(); //Parent

 p1 = c1;
 p1.iam(); //Child
 ...

```

Handle type = parent

Handle type = child



Simply declaring a method as virtual means a call is resolved according to the contents of a handle, not the type of the handle.

Here we create an instance of the parent class and copy it into the base class handle b1. When we call b1.iam(), the method call is resolved by first looking in the class type of b1, i.e., the base class. In class base, iam() is declared as virtual, and this forces the resolution to go back to the call and examine the contents of b1. Here we find b1 contains an instance of parent, so the resolution is directed to the parent class. So b1.iam() returns Parent when iam() is declared as virtual.

Likewise, when we copy an instance of child into the parent handle p1, the call p1.iam() is resolved by looking in the parent class first. Again we find iam() is declared as virtual, so the resolution is redirected to the type of the contents of p1, i.e., Child.

Only methods can be virtual, not properties. If you need to access a sub-class property from a base class handle, you need to access the property via a virtual method.

Once a method is declared as virtual, it is automatically virtual in every sub-class. Therefore declaring iam() as virtual in base means it is virtual in parent and child, and the virtual keyword can be omitted for the parent and child declaration (although this can be considered bad practice).

## How to Resolve Class Method Access



With Polymorphism, when a method is accessed off a class handle, how do you identify which class method is used?

1. Examine the class declaration of the handle type.
2. If the method is not virtual, then the call is directed to the handle class.
3. If the method is virtual, then examine the contents of the handle.
4. If the handle contains a sub-class instance, the call is directed to the sub-class.
5. If the handle does not contain a sub-class instance, the call is directed back to the handle class.

```
class base;
 function void nvirt();
 ...
 endfunction

 virtual function void virt();
 ...
 endfunction
endclass

class parent extends base;
 virtual function void virt();
 ...
 endfunction
endclass

base b1;
parent p1 = new();

initial begin
 b1 = p1;
 b1.nvirt(); //base
 b1.virt(); //parent
```

Calls to a normal method resolve to the handle class, if implemented there; else the nearest higher class where implemented i.e., compile-time resolution.

Calls to a virtual method resolve to the object class in the handle, if implemented there; else the nearest higher class where implemented, i.e., runtime resolution.

## Abstract Classes and Pure Virtual Methods



IEEE 1800-2012 18.16

```

virtual class base; Virtual class
...
pure virtual function void iam(); Pure virtual method

endclass

class parent extends base;
 virtual function void iam();
 $display ("Parent");
 endfunction
endclass
endclass

class child extends parent;
 virtual function void iam();
 $display ("Child");
 endfunction
endclass

```

**Sub-class implementations**

A virtual class exists *only* to be inherited – it cannot be instantiated.

- Also known as an “abstract” class.

A virtual class (only) may have *pure virtual* methods:

- Prototype only – no implementation.
- Sub-classes must provide implementation.

```

base b1 = new(); //instance illegal
parent p1 = new();

initial begin
 b1 = p1;
 b1.iam(); //Parent

```



234 © Cadence Design Systems, Inc. All rights reserved.

You can use the `virtual` keyword to declare a virtual class. A virtual (or abstract) class type cannot be instantiated; it only exists as a base class for sub-classes to use for inheritance. You cannot create an instance of a virtual class, although you can declare handles (for use in polymorphism).

The implication is that virtual class declares a set of class members which can be shared by many sub-classes, but that the set is not sufficient to be used by itself – the sub-classes must add extra members to make the class usable.

An abstract class (and only an abstract class) can declare *pure virtual* methods. A pure virtual method is a method *prototype* declared with the `pure` and `virtual` keywords i.e., only the first line is defined. This identifies the method as a task, or function, and defines its name and arguments.

A sub-class which extends the abstract class must provide a full implementation of the pure virtual method. Specifically, a sub-class instance cannot be created unless all pure virtual methods have an implementation.

So a virtual (abstract) class can define:

- Methods
- Virtual methods
- Pure virtual methods

A nonvirtual class can define:

- Methods
- Virtual methods

## Module Summary

Polymorphism, virtual methods and virtual classes are essential object-oriented features, which allow:

- A generic testbench to be written using base classes
- A testbench to be easily extended, inherited, overridden, and modified to suit requirements
  - While changing very little of the existing code
- Improved reusability and maintainability



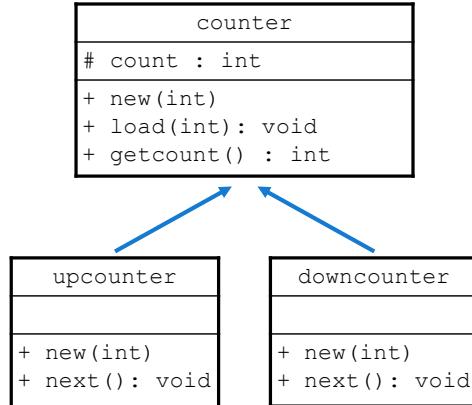
*This page does not contain notes.*



## Lab

### Lab 12 Using Class Polymorphism and Virtual Methods

- Analyze polymorphism and virtual methods using simple classes.



236 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*



## Class-Based Random Stimulus

**Module** **16**

Revision **1.0**  
Version **21.10**

Estimated time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

- Define class properties for randomization
- Create random data using the `randomize()`, `pre_randomize()` and `post_randomize()` methods
- Select random properties with in-line and `rand_mode()` options
- Control random data with constraint blocks and `constraint_mode()`
- Analyze the order in which properties are randomized and explore how to control the order



*This page does not contain notes.*

## Random Class Properties



Random Class Properties are integral class properties that are defined/declared as random using `rand` or `randc`.

```
class randclass;
 rand bit[1:0] p1;
 randc bit[1:0] p2;
endclass
```

- `rand` – Random with uniform distribution.
- `randc` – Random-cyclic randomly iterates through all values without repetition:
  - When an iteration is complete, a new random iteration automatically starts.
  - Implementation may limit size.

Each value has an equal probability (1/4)

`p1` rand example output: 01 11 00 10 01 11 01 10 11 00 01 11

Close repetition is common

`p2` randc example output: 01 11 00 10 00 11 10 01 10 00 11 10

Each iteration cycles through all values without repetition

Close repetition across iterations is possible

Class properties are defined as random using either the `rand` or the `randc` modifier. Random properties must be integral types.

`rand` properties have a uniform distribution, i.e., every possible value has an equal probability every time the property is randomized. Repetition of specific values over a small number of randomizations is possible and likely.

`randc` properties are random-cyclic, where every possible value must appear once, in random order, before a specific value can be repeated. `randc` randomization is broken down into a series of iterations. Within each iteration, randomization cycles through all possible values, in random order, without repetition. Once all values have been generated, a new iteration automatically starts. The new iteration randomly cycles through all possible values, but in a different order than the previous iteration. Repetition of values over a small number of randomizations is possible, but these values will be from different iterations.

At the start of an iteration, all values are equally probable. Once a value is generated, it is excluded until the start of the next iteration.

`randc` generation is computationally intensive, therefore implementations can impose a limit on the maximum size of a `randc` variable, but SystemVerilog dictates that a minimum of 8 bits should be supported.

## Randomizing Class Objects: `randomize()`



IEEE 1800-2012 18.3

Randomize the properties by calling the `randomize()` function.

- Every class has a built-in randomize virtual method.
- You cannot re-declare this method.

```

class randclass;
 rand bit[1:0] p1;
 randc bit[1:0] p2;
endclass
randclass myrand = new();
int ok;
initial begin
 ok = myrand.randomize();
 if (!myrand.randomize())
 $display ("myrand randomize failure");
end
...

```

240 © Cadence Design Systems, Inc. All rights reserved.



A clocking block is both a declaration and instance of that declaration. You do not separately instantiate a clocking block. A clocking block can be declared in an interface, module or program. It cannot be declared in a package or a Compilation Unit Scope.

A clocking block declaration defines:

- The clocking block event: Usually an edge on a clock signal.
- Signal direction: A clocking block does not declare signals, but only qualifies the direction of pre-declared signals for the clocking block. The direction is always from the perspective of the testbench. Clocking block outputs are design inputs, and clocking block inputs are design outputs. Bidirectional signals can be qualified as `inout`, which is simply a shorthand notation for separate input and output qualifiers on the same signal.
- Input and output delay (skew): Skew can be defined explicitly for each input or output, or by default for all inputs and outputs. Default and explicit skew can be mixed, with explicit skew takes precedence.

Outputs are delayed in the clocking block by skew delay after the clocking block event, before being driven into the design.

Inputs are sampled by the clocking block at skew delay before the clocking block event.

## **pre\_randomize() and post\_randomize()**

randomize() automatically calls two “callback” functions:

- pre\_randomize() before randomization.
- post\_randomize() after successful randomization.

If defined, these methods are automatically called on randomization.

```
function void pre_randomize();
 ...
endfunction

function void post_randomize();
 ...
endfunction
```

```
class randclass;
 rand bit[1:0] p1;
 randc bit[1:0] p2;
 bit[1:0] parity;
 function void post_randomize();
 parity = p1 ^ p2;
 endfunction
endclass

randclass myrand = new();
int ok;

initial begin
 ok = myrand.randomize();
 ...

```

Define post\_randomize

randomize automatically calls post\_randomize

241 © Cadence Design Systems, Inc. All rights reserved.



Although you cannot re-declare the randomize function, you can customize randomization by declaring either of two “callback” methods in the class. When randomize is called, the simulator automatically searches the class for a declaration of pre\_randomize and, if it finds one, executes this before randomization. Then after successful randomization, the simulator automatically searches the class for a declaration of post\_randomize and, if it finds one, executes it.

The pre/post\_randomize declarations must match the prototypes shown, i.e., they must be void functions with no arguments.

A good use for post\_randomize is to calculate parity so it is consistent with the new randomized properties, as shown in this example.

It is considered bad practice to call pre/post\_randomize directly. They should only be executed as part of the randomize call. If pre/post-randomize needs to contain functionality which can be called outside of randomization, then a separate method should be declared for this and called from within the randomization callback methods.

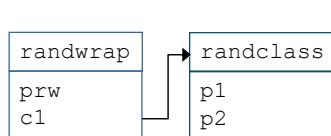
Be careful with naming. An error in the name of the callback method will mean it is not executed when randomize is called.

pre/post randomize are effectively virtual despite not being explicitly defined as such.

## Example: Applying Randomization in Aggregate Classes

Randomize can operate hierarchically on aggregate classes:

- The class instance property must be declared as `rand`.
- Otherwise, that instance is skipped for randomization.



c1 must be `rand` for its p1 and p2 properties to be randomized

```

class randclass;
 rand bit[1:0] p1;
 randc bit[1:0] p2;
endclass

class randwrap;
 rand int prw;
 randclass c1;
endclass

randwrap mywrap = new();
int ok;

initial begin
 ok = mywrap.randomize();
 ...
end

```

242 © Cadence Design Systems, Inc. All rights reserved.



An aggregate class may contain class instance properties which themselves contain random properties. Here the `randwrap` aggregate class contains an instance of `randclass` with two random variables, `p1` and `p2`.

The `randomize` method can push down into the `randclass` instance within `randwrap`, and randomize `P1` and `P2`, but only if the `randclass` handle (`c1`) is declared as `rand`. If `c1` is not declared as `rand`, properties `mywrap.c1.p1` and `mywrap.c1.p2` will not be randomized, even though they are random properties.

Effectively, the `rand` modifier acts as a gate for randomization. If a class instance property is `rand`, randomization pushes into the property to randomize its contents, otherwise the instance is skipped.

## Example: In-Line Random Variable Control with `randomize()`

Specific class variables can be randomized by passing them as arguments.

- Also, allows nonrandom (state) properties to be randomized.

```
class randclass;
 rand bit [0:1] p1;
 randc bit [1:0] p2;
 bit [1:0] s1, s2;
endclass

randclass myrand = new;
int ok;

initial begin
 ok = myrand.randomize();
 ok = myrand.randomize(p1);
 ok = myrand.randomize(p1,s1);
 ok = myrand.randomize(s1,s2);
end
```



Normally, the `randomize` method randomizes all random properties of the class. However, you can select specific properties to be randomized, even non-random state properties, by passing property names as arguments to the `randomize` method. Once any argument is passed as an argument, then only the arguments are randomized, and all other properties are left unchanged.

Randomization of state variables passed as `randomize` arguments is always as if they were declared as `rand`, i.e., randomization is not cyclic. If you need cyclic randomization, then the properties must be explicitly declared as `randc`.

## Controlling Randomization: `rand_mode()`



IEEE 1800-2012 18.8

Every random property has an enable switch called `rand_mode`.

- Enabled by default (1).
- If disabled (0), the property will not be randomized.

Mode can be written with `task rand_mode`:

```
task rand_mode(bit on_off);
```

- Called off a random property, the task changes the mode of that property.
- Called off an instance, the task changes the mode for all random properties of the instance.

Mode can be read with `function rand_mode`:

```
function int rand_mode();
```

Only random properties have `rand_mode`:

- Calling method off a non-random property is a compile error.

244 © Cadence Design Systems, Inc. All rights reserved.



Every random property in a class has an enable bit named `rand_mode`. If the mode is disabled (set to 0), then the property will not be affected by randomization, however the property value will still be used in constraints. The default mode setting is enabled.

The mode can be changed using the built-in task method `rand_mode()`. The task can be applied to an individual random property, in which case the mode is changed just for that property. Alternatively, the task can be applied to a class instance, in which case the mode of all of the random properties in that instance are changed. For aggregate classes, setting the `rand_mode` of a random property, which is a class instance, does not change the mode of the random properties within the instance, but can affect whether or not they are randomized.

The mode can be read using the built-in function method `rand_mode()`. The function can only be applied to an individual random property and returns the current value of the mode.

You can also apply these methods to individual elements of an unpacked array and individual elements of an unpacked structure providing that the array or structure are declared as random properties.

## Example: `rand_mode()`

```

class randclass;
 rand bit[1:0] p1;
 randc bit[1:0] p2;
 bit[1:0] s1, s2;
endclass

randclass myrand = new;
int ok, state;
initial begin
 myrand.rand_mode(0);
 myrand.p2.rand_mode(1);
 state = myrand.p2.rand_mode();
 ok = myrand.randomize();
 state = myrand.s1.rand_mode();
end

```

p1, p2 are random properties

s1, s2 are state properties

Disable randomization of all random variables of myrand

Re-enable p2 randomization

Read p2 mode (1)

Only p2 randomized

Error – s1 is not a random variable

245 © Cadence Design Systems, Inc. All rights reserved.



The example uses the `rand_mode()` task call to set the mode of instance `myrand` of class `randclass` to 0. This sets the `rand_mode` of `myrand.p1` and `myrand.p2` to 0.

The `rand_mode()` task is then used to set the mode of `myrand.p2` to 1.

The `rand_mode()` function reads the mode of `myrand.p2` and stores the value in `state`, which will contain the value 1.

When the `myrand` instance is randomized, only `myrand.p2` is affected.

Applying the `rand_mode` task or function to a non-random property (such as `s1`) is a compilation error. Only random properties have a `rand_mode`.

## Constraint Blocks



IEEE 1800-2012 18.5

- Constraints can be embedded in classes using constraint blocks.
- A block can contain any number of any form of constraints:
  - Relational
  - List or distribution
  - Conditional

```

class randclass;
 rand bit [1:0] p1;
 rand bit [7:0] p3;

 constraint c1 { p1 != 2'b00; }
 constraint c3 { p3 >= 64; p3 <= 192; }
endclass

randclass myrand = new;

int ok;
initial begin
 ok = myrand.randomize();
 ...
end

```

246 © Cadence Design Systems, Inc. All rights reserved.



Constraints apply restrictions to randomization to exclude values or change the probability distribution.

You declare a constraint block as a class member with the `constraint` keyword, followed by an identifier, followed by a list of constraint items enclosed within curly braces `{ }`. Each constraint item in the list is terminated by a semicolon, but the constraint block itself is terminated by the closing curly bracket `}` and does not end with a semicolon.

Constraint block items can be constraint expressions and statements ordering the constraint solution. Constraint expressions can be almost any integral expression. They cannot have side effects (for example through assignment operators). The use of functions is also restricted (see LRM for more details).

## Constraint Inheritance



IEEE 1800-2012 18.5.2

Constraints are class members and are inherited just like other members.

```

class randclass;
 rand bit [1:0] p1;
 constraint not0 {p1 != 2'b00;}
endclass

class rcx1 extends randclass;
 constraint not3 {p1 != 2'b11;}
endclass

class rcx2 extends randclass;
 constraint not0 {p1 != 2'b01;}
endclass

class rcx3 extends randclass;
 constraint not0 {}
 constraint not1 {p1 != 2'b01;}
endclass

...
rcx1 myrand = new;
initial begin
 ok = myrand.randomize();
 ...

```

not0 constraint in base class

rcx1 adds constraint not3 to constraint not0 from randclass

rcx2 overrides constraint not0

rcx3 removes constraint not0 and defines new constraint not1

p1 is 01 or 10

247 © Cadence Design Systems, Inc. All rights reserved.



Constraints are class members and are inherited just like all other inherited class members.

Here we declare a base class `randclass` with a single random property `p1` and single constraint `not0` which restricts `p1` to 01, 10 or 11.

Sub-class `rcx1` defines a new constraint `not3`. When instances of `rcx1` are randomized, local constraint `not3` and inherited constraint `not0` will both apply and `p1` is restricted to 01 or 10.

Sub-class `rcx2` redefines the constraint `not0` and so overrides the inherited constraint. When instances of `rcx2` are randomized, the single local constraint `not0` applies and `p1` is restricted to 00, 01 or 10.

Sub-class `rcx3` redefines the constraint `not0`, but does not define an expression. This has the effect of removing the constraint. We then define a new constraint `not1`. If you want to replace a constraint, it may be more readable to remove the existing constraint by defining it as empty, and define a new constraint with a better name. If we had not declared `not1`, then removal of `not0` would mean there are no restrictions on `p1`.

In cases such as `rcx2` when you want to redefine an existing constraint, it may be confusing to use the same name. It may be better to remove the existing constraint (as in `rcx3`) and then define a new constraint with a more meaningful name.

## Constraint Expressions: Set Membership



IEEE 1800-2012 18.5.3

The `inside` operator is particularly useful in constraint expressions.

- The operator can also be negated to generate a value outside of a set.

```
class randclass;
 rand bit[7:0] p3;
 constraint c1 {p3 inside {3, 7, [11:20]};}
endclass

randclass myrand = new;

int ok;
initial begin
 ok = myrand.randomize();
 ...
end

class not_inside;
 rand bit[7:0] p3;
 constraint c2 { !(p3 inside {1, 7, [10:255]}) ; }
endclass
```

c1 constrains p3 to  
the set 3,7,11-20

c2 constrains p3 to  
outside the set 1,7,10-255

248 © Cadence Design Systems, Inc. All rights reserved.



The `inside` operator allows you to define a constraint which constrains a random property to a value in a list. The list can contain ranges as well as individual values. The `inside` operator uses curly brackets {} to define the list of allowed values. Values or ranges in the list are comma separated.

An inside constraint expression can also be negated to constrain a random property to a value outside of a list of values. Indeed, any constraint expression can be negated, by enclosing the expression in parentheses and applying an inversion operator.

For example, constraint c2 constrains p3 to be 0, 2, 3, 4, 5, 6, 8 or 9.

## Constraint Expressions: Weighted Distributions



IEEE 1800-2012 18.5.4

- You can change distribution by defining weights for values using `dist`.

- Default weight is 1

- There are two ways to assign weight:**

- `:=` assigns weight to the item or *every* value in a range.

```
constraint c1 { p4 dist { [101:200]:=200 }; }
```

101 to 200 each gets a weight of 200

- `:`/`/` assigns weight to the item or to a range as a *whole*.

```
constraint c2 { p4 dist { [101:200]:/200 }; }
```

101 to 200 each gets a weight of 2  
(200/100=2)

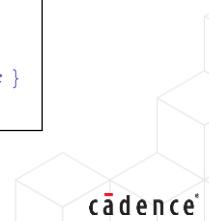
```
class randclass;
 rand int p4;
 constraint c3 {p4 dist {7:=5, [11:20]:=3, [26:30]:/1};}
endclass
```

7 has a weight of 5

11-20 each has a weight of 3

26-30 each has a weight of 1/5

249 © Cadence Design Systems, Inc. All rights reserved.



Normal distribution (probability) for randomization is uniform, where every possible value is equally likely.

You can change the distribution of values using the `dist` operator. This is very similar to the `inside` operator in that you constrain randomization to a list of values. However, the `dist` operator allows you to define weights, or relative probabilities, to each value in the list.

The default weight is 1.

There are two forms of weight assignment, and these only differ when assigning a weight to a range.

The `:=` operator assigns the weight to every value in the range.

The `:`/`/` operator divides the weight by the number of values in the range, i.e., for a range of n values and a weight of w, the weight of each individual value is w/n.

## Constraint Expressions: Iterative Constraints



IEEE 1800-2012 18.5.8

- You can use a loop to apply separate constraints to each array element.

```
class randclass2;
 rand logic [3:0] arr[7:0];
 constraint c1 { foreach (arr[i])
 (i <= 4) -> arr[i] <= i; }
 constraint c2 { foreach (arr[i])
 (i > 4) -> arr[i] >= i; }
endclass
```

Iterative  
constraints

### Constraints

```
arr[0] <= 0;
arr[1] <= 1;
arr[2] <= 2;
arr[3] <= 3;
arr[4] <= 4;
arr[5] >= 5;
arr[6] >= 6;
arr[7] >= 7;
```

Can affect performance for large arrays or complex constraints.



250 © Cadence Design Systems, Inc. All rights reserved.



You can apply individual constraints to each element of an array dimension using the `foreach` loop.

Iterative constraints are very powerful. Consider the following constraint for the example above:

```
constraint c3 {
 foreach (arr[k])
 (k < 7) -> arr[k + 1] > arr[k];
}
```

This constrains each array value to be greater than the preceding element, giving you a random array with element values sorted in descending order.

## Controlling Constraints with `constraint_mode()`



IEEE 1800-2012 18.9

Every constraint block has an enable switch called `constraint_mode`.

- Enabled by default (1).
- If disabled (0), the constraint block will not be used.

Mode can be written with *task* `constraint_mode`.

```
task constraint_mode(bit on_off);
```

Mode can be read with *function* `constraint_mode`.

```
function int constraint_mode();
```

Only constraint blocks have `constraint_mode`.

251 © Cadence Design Systems, Inc. All rights reserved.



Every constraint block in a class has an enable bit named `constraint_mode`. If the mode is enabled (set to 1), then the constraint block will be used by the randomize method. If the mode is disabled (0), then the constraint block will be ignored by randomize. The default mode setting is enabled.

The mode can be changed using the built-in task method `constraint_mode()`. The task can be applied to an individual constraint block, in which case the mode is changed just for that block.

Alternatively, the task can be applied to a class instance, in which case the mode of all of the constraint blocks in that instance are changed.

The mode can be read using the built-in function method `constraint_mode()`. The function can only be applied to an individual constraint block and returns the current value of the mode.

## Example: Application of constraint\_mode()

```

class randclass;
 rand bit [1:0] p1;
 constraint blue {p1 != 2'b00;};
 constraint green {p1 != 2'b11;};
endclass
randclass myrand = new;

int state, ok;
initial begin

 myrand.constraint_mode(0);
 myrand.blue.constraint_mode(1);
 state = myrand.green.constraint_mode();
 ok = myrand.randomize();

end

```

blue constrains p1 to not 00

green constrains p1 to not 11

Disable all constraints for myrand

Re-enable blue constraint

Read green mode (0)

green constraint disabled:  
p1 will be 01, 10, 11

The example uses the `constraint_mode()` task call to set the mode of instance `myrand` of class `randclass` to 0. This sets the `constraint_mode` of `myrand.blue` and `myrand.green` to 0.

The `constraint_mode()` task is then used to set the mode of `myrand.blue` to 1.

The `constraint_mode()` function reads the mode of `myrand.green` and stores the value in `state`, which will contain the value 1.

When the `myrand` instance is randomized, only the `myrand.blue` constraint is used.

Applying the constraint mode task or function to a class property is a compilation error. Only constraint blocks have a constraint mode.

## Randomization Procedure and Its Effects



**Problem:** Randomization Procedure can lead to unexpected results:

- For example, here, you expect mode to be one half the time, and this is not the case.

**Solution:** Order of randomization can be defined:

- Use `solve...before` (Applies to rand variables only).
- Or use `randc` properties.

```
class randclass;
 rand bit[2:0] vect;
 rand bit mode;
 constraint c1
 { mode -> vect == 0; }
endclass
```

Expectation: mode to be '0' one half the time

Result: mode is '0' one ninth of the time

Applying `solve...before`

```
class randclass_ordered;
 rand bit[2:0] vect;
 rand bit mode;
 constraint c2
 { mode -> vect == 0;
 solve mode before vect; }
endclass
```

Ordering constraint

When a class instance is randomized, `randc` properties are always randomized first, simultaneously. Then all the `rand` properties are randomized together. Then the constraints are checked to see whether the solution is correct, and if not, the process is repeated until either a solution is found or the randomization options are exhausted. If a solution cannot be found, then randomize fails, the randomize method returns 0 and the simulator reports a constraint violation.

Obviously, this is the conceptual view and the process is heavily optimized in the simulator.

Randomization of properties in parallel can lead to unexpected results.

For example, the constraint `c1` defines that if `mode` is 1, then `vect` is zero. This may lead you to expect `mode` will be one (1) 50% of the time. However, this is not seen in simulation, because both `mode` and `vect` are generated simultaneously. The next slide describes this in more detail.

You can control the order in which `rand` properties are generated by using the `solve... before` constraint. If you generate `mode` first, then it will be 1 half the time and so `vect` will be zero half the time. The other half of the time, `mode` is zero and `vect` is unconstrained.

Keep in mind the following restrictions:

- You cannot order `randc` variables. The solver solves `randc` variables first, then ordered `rand` variables, then unordered `rand` variables.
- You can only order `rand` variables of integral types.
- You must avoid circular dependencies e.g.,
 

```
* {solve a before b; solve b before c; solve c before a;}
```

## Solution: Ordering Constraint

Conceptual un-ordered solution

Unordered solution:

- vect and mode randomized simultaneously
- Constraints applied
- All solutions equally likely
- Probability mode = 1 is 1/9

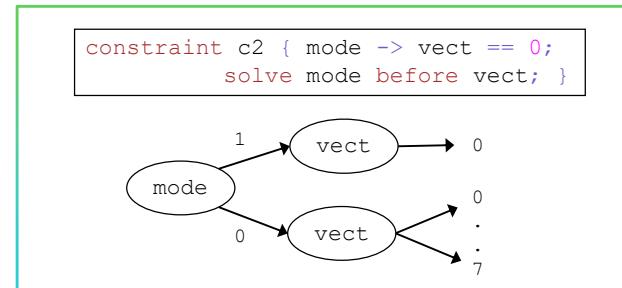
|      |   | vect |     |     |     |     |     |     |     |
|------|---|------|-----|-----|-----|-----|-----|-----|-----|
|      |   | 0    | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
| mode | 0 | 0,0  | 0,1 | 0,2 | 0,3 | 0,4 | 0,5 | 0,6 | 0,7 |
|      | 1 | 1,0  | 1,1 | 1,2 | 1,3 | 1,4 | 1,5 | 1,6 | 1,7 |

Excluded by constraint

Conceptual ordered solution

Ordered solution:

- mode randomized first
- vect randomized based on constraints
- Probability mode = 1 is 1/2



cadence®

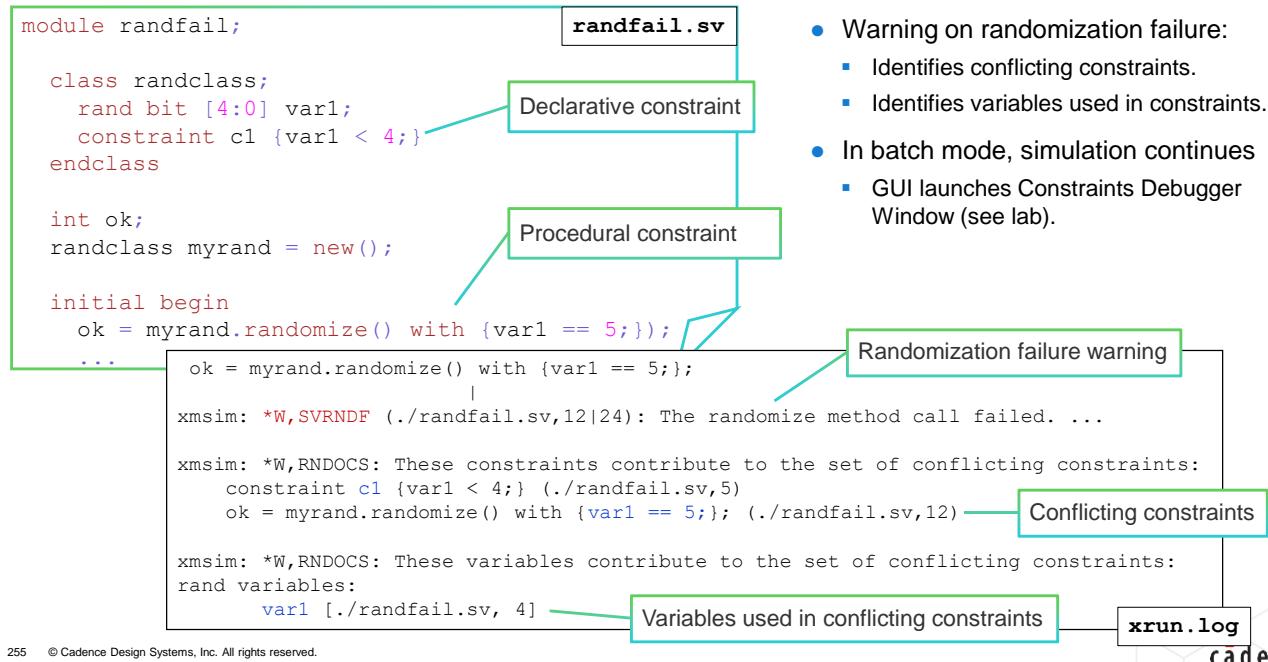
254 © Cadence Design Systems, Inc. All rights reserved.

In the un-ordered solution, both mode and vect are randomized simultaneously. This generates 16 possible combinations for mode and vect. Then the constraints are applied and the invalid combinations are removed – when mode is 1, any value of vect other than 0 is invalid. The remaining solutions are all equally likely, meaning the combination of mode = 1 and vect = 0 has a probability of 1/9.

In the ordered solution, the solve constraint requires the simulator to generate mode before vect. This gives a 50% chance of mode = 1, from which the only possible value of vect is 0. If mode is 0, then vect is unconstrained and there are 8 possible values for vect. As mode is generated first, the combination of mode = 1 and vect = 0 has a probability of 1/2 .

These are conceptual explanations to aid understanding. Optimizations in the Random Number Generation may use different processes, but will generate the same results.

## Randomization Failure



255 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

Randomization will fail if a solution cannot be found which meets all applicable constraints. By default, the failure will generate a warning message, not an error or fatal message. The message identifies the conflicting constraints and the variables used in the constraints. If the list of variables includes static (non-random) variables, then their value at the time of randomization will be shown.

The actual format of the message may vary between tool versions but the essential information is always included.

If randomization fails for one variable, then it fails for all variables used in the randomize call, and no values are changed.

In batch-mode, by default, the simulation generates the warning and continues. Therefore you need to check the simulation logs carefully for failures.

In the GUI, by default, the simulation generates the warning; stops the simulation and opens the Constraints Debugger Window showing the information on the conflicting constraints and variables affected.

## Module Summary

With these class randomization features, you can:

- Generate large amounts of stimulus data from compact code:
  - Run longer simulations with more stimulus:
    - That more thoroughly tests the design
  - Spend more of your own time crafting directed tests of corner cases
- Constrain the values of the random variables:
  - To create legal stimulus
  - To explore areas of interest
  - To conditionally switch between modes of operation



*This page does not contain notes.*

## Quiz



Which variables are randomized in below code?

Both these are enabled with  
obj3.obj2.rand\_mode(1)

This is not declared  
as random, hence  
obj1 properties not  
randomized

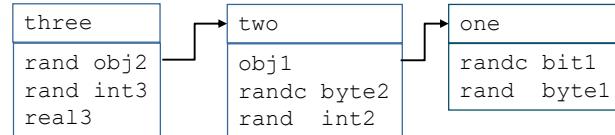
obj3.int3 – Disabled  
hierarchically by  
obj3.rand\_mode(0)

obj3.real3 – Not a  
random property

```
class one;
 randc bit bit1;
 rand byte byte1;
endclass

class two;
 randc byte byte2;
 rand int int2;
 one obj1;
 function new();
 obj1 = new;
 endfunction
endclass

class three;
 rand int int3;
 real real3;
 rand two obj2;
 function new();
 obj2 = new;
 endfunction
endclass
```



```
three obj3 = new;
int ok;

initial begin
 obj3.rand_mode(0);
 obj3.obj2.int2.rand_mode(0);
 obj3.obj2.rand_mode(1);
 ok = obj3.randomize();
end
```

Class 'two' properties  
byte2 and int2 will be  
randomized

257 © Cadence Design Systems, Inc. All rights reserved.



There are 6 properties in instance obj3 of the class three:

- obj3.int3  
Disabled hierarchically by obj3.rand\_mode(0).
- obj3.real3  
Not a random property
- obj3.obj2.byte2  
Disabled hierarchically by obj3.rand\_mode(0), but re-enabled hierarchically by  
obj3.obj2.rand\_mode(1)
- obj3.obj2.int2  
Disabled explicitly by obj3.obj2.int2.rand\_mode(0)
- obj3.obj2.obj1.bit1  
obj3.obj2.obj1 is not a rand property, so not randomized
- obj3.obj2.obj1.byte1  
obj3.obj2.obj1 is not a rand property, so not randomized

So the only property randomized is obj3.obj2.byte2.

## Quiz

1. What value(s) will be printed for r1 after the 1<sup>st</sup> randomize()?

'0' because s1 and s2 are initialized to 0.

2. What value(s) could be generated for r2 in the 1<sup>st</sup> randomize()?

r2 cannot be 0 so would be 1, 2 or 3.

3. How could we get r2 == 0?

Initialize s1 or s2 to a non-zero value, or disable one of the constraints.

```
module m;

class myrand;
 bit [1:0] s1, s2;
 rand bit [1:0] r1, r2;
 constraint c1 { r1 != r2; }
 constraint c2 { r1 inside {s1,s2}; }

 function void post_randomize;
 $display("s1=%id, s2=%id, r1=%id, r2=%id", s1,s2,r1,r2);
 endfunction

endclass

myrand ci = new;

initial begin
 void'(ci.randomize());
 void'(ci.randomize(s1,s2));
end

endmodule
```

4. What values would be printed for s1 and s2 after the 2<sup>nd</sup> randomize() call?

s1 and s2 are subject to constraints c1 and c2, we know r1 == 0 after the 1<sup>st</sup> randomize(), so one of s1 or s2 must be 0, and the other can take any value 0,1,2,3.

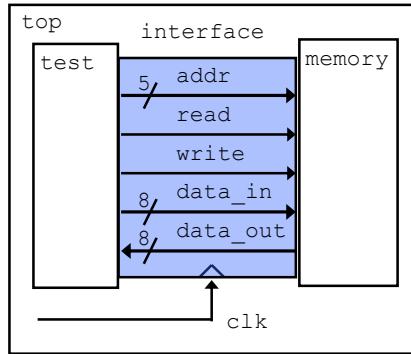
*This page does not contain notes.*



## Lab

### Lab 13 Using Class-Based Randomization

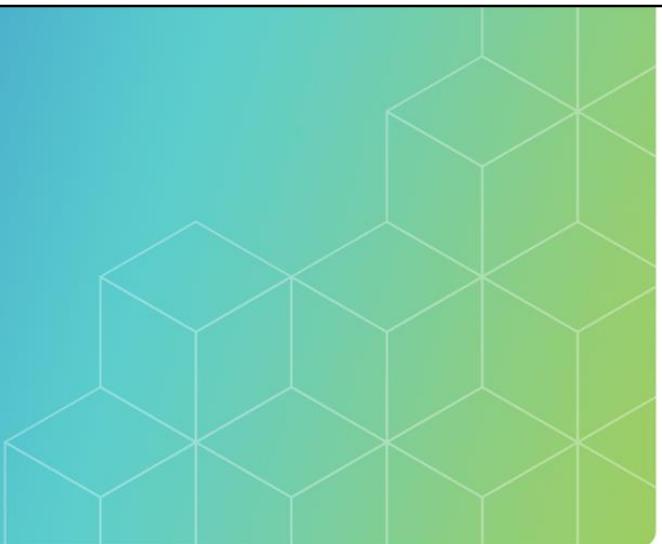
- Modify your memory module testbench to use constrained class-based randomization.



259 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*



## Interfaces in Verification

**Module** **17**

Revision **1.0**

Version **21.10**

Estimated time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

- Analyze the concept of Transaction-Based Verification
- Identify the architecture of a typical Verification Component (VC)
- Write virtual interfaces to create a reusable module and class-based VCs using generic stimulus tasks
- Apply clocking blocks to interface declarations



*This page does not contain notes.*

## Class-Based Testbench



It is a way of creating testbenches that are more efficient, easy to write, and enables the user to verify design with high-abstraction level modeling.

Why use class-based testbenches?

- Easier to maintain, reuse, etc.
- Inheritance allows instance-specific customization of data and functionality.
  - Without affecting original code.
- Separation of stimulus and environment.
  - Allows test writers to drive environment with minimal knowledge of protocols, hierarchy, etc.

User testbench



UVM class library



SystemVerilog

**Note:** SystemVerilog provides very little class infrastructure and hence requires the UVM verification class library which provides everything from `print()` methods to simulation control.

262 © Cadence Design Systems, Inc. All rights reserved.



There are many reasons we use SystemVerilog classes for testbenches.

Classes force higher-abstraction level modelling, which, in addition to being more efficient to write, is easier to maintain and reuse.

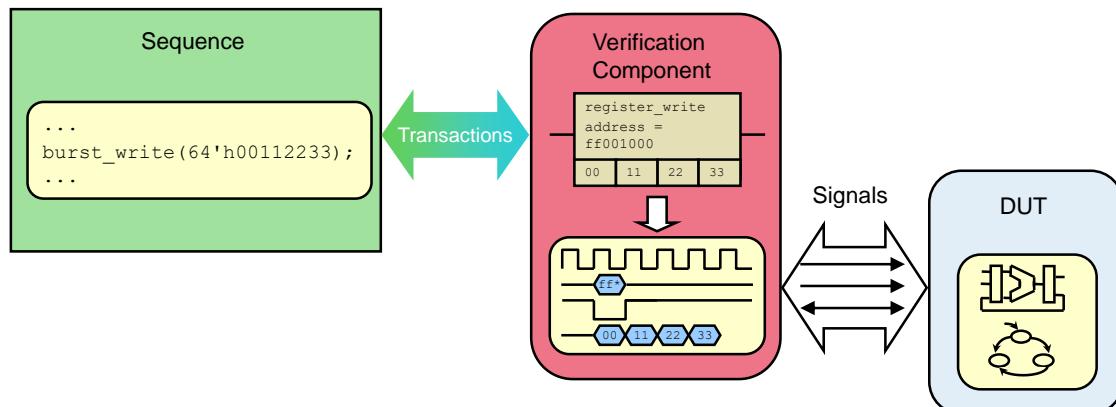
Through inheritance, we can create a sub-class of any existing class and extend or modify its data or functionality by overwriting or adding properties or methods. Crucially, the original class is unaffected. Sub-classes can then be swapped in and out for specific class instances in a verification environment. This allows multiple versions of a verification component to co-exist, greatly extending reuse.

By separating stimulus and the environment which drives the stimulus into the Design Under Test(DUT), we create a test-writer interface to the verification environment. This allows a verification engineer to drive an environment while having minimal knowledge of the environment hierarchy or protocols.

The downside to a class-based testbench is that SystemVerilog provides very little class infrastructure. You can randomize a class, but to print, copy, compare or create building blocks for verification requires additional code. The Universal Verification Methodology (UVM) does exactly this. It is a library of pre-written verification building-block classes as well as a methodology defining how to construct a testbench using these classes.

UVM is covered in the *SystemVerilog Advanced Verification with UVM* training class.

## Separation of Stimulus and Environment



Testbench interaction with a bus is in two layers:

- Sequence:
  - Defines progression of high-level stimulus (transactions).
- Verification Component (VC):
  - Translates between transactions and signal transitions.

263 © Cadence Design Systems, Inc. All rights reserved.



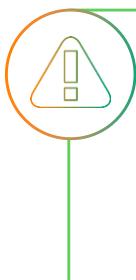
Each separate bus protocol or group of related signals will have a separate Verification Component (VC) which reads and writes the signals of the bus.

A sequence defines the stimulus that the VC will drive onto the bus. This is a sequence of high-level transactions abstracted from the protocol of the bus.

The VC takes the transactions of the sequence, one at a time, and drives them onto the signals of the DUT with the correct protocol.

Within the Verification Component, there will be a driver, or Bus Functional Model (BFM), which converts between the high-level transactions and the low-level signal transitions at the Design Under Test (DUT). This conversion can go both ways. For example, the next data operation from the sequence may depend on the response of the DUT from the last operation. Therefore, the VC needs to convert the response from low-level signal transitions to a high-level transaction so the sequence can understand the DUT response.

## Class Connection to DUT Ports



**Problem:** Class-based VCs drive DUT signals via interfaces. However, VCs should not be directly connected to an interface instance.

- Breaks reusability
  - If `myvc` is hardwired to `bus1`, cannot drive `bus2`.
- Interface instances are static
  - `myvc` has to be declared local to the `bus1` instance.

**Solution:** What we need is an interface variable:

- Used in class to access interface signals (via **Virtual Interface**).
- Assigned to a specific interface instance when class is instantiated.

```
interface myif (input clk);
 logic [7:0] data;
 ...
endinterface
```

```
module test;
 logic clk;
 myif bus1(clk), bus2(clk);
 comp DUT1 (bus1);
 comp DUT2 (bus2);

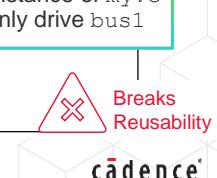
 class myvc;
 task write_data(input int datin);
 @ (negedge bus1.clk);
 bus1.data = datin;
 ...
 endtask
 endclass : myvc

 myvc c1;

 initial begin
 c1 = new();
 c1.write_data(5);
 ...

```

264 © Cadence Design Systems, Inc. All rights reserved.



A class-based verification component needs to be able to drive stimulus on the DUT ports. The most convenient approach is to instantiate interface instances connected to the DUT at one end, and read/written from the verification component at the other. However we don't want to reference a specific interface instance in the verification component class.

Firstly, this breaks reusability. If we hardware the class to drive interface instance `bus1`, then the user must always use that instance name for their interface. This prevents the class from being used with a different instance name, and would require separate classes to drive multiple instances of the interface and DUT.

Secondly, interface instances are static. Therefore, the class is to be declared in the same scope as the interface instance, preventing its declaration in a more convenient package.

## Solution: Virtual Interfaces

The solution is to use a virtual interface.

- An interface variable that can be connected to an interface instance.
- Can be referenced inside classes.
- Allows access to interface signals using variable name as a prefix.
- Needs to be assigned to an actual interface.

A virtual interface:

- Can be declared as a class property.
- Default value is `null`.
  - Must be assigned to an interface instance.
- `interface` keyword in the declaration is optional.
  - Include for readability.

```
interface myif (input clk);
 logic [7:0] data;
 ...
endinterface
```

```
class myvc;
 virtual interface myif vif;
 ...
 task write_data(input int datin);
 @ (posedge vif.clk);
 vif.data = datin;
 endtask
endclass
```

265 © Cadence Design Systems, Inc. All rights reserved.



A virtual interface is a variable that can be mapped to different interface instances (of the same type) during simulation.

Declaration syntax is:

```
virtual interface <interface type> <variable name>;
```

The `interface` keyword is optional, but as the keyword `virtual` is used for many different purposes, `interface` is recommended for clarity.

The signals of the interface can be referenced by using the virtual interface name as a prefix, instead of a specific interface instance. The virtual interface must then be individually assigned to an actual interface instance for every VC. A virtual interface can be defined as a class property, allowing the dynamic class to reference a static interface instance.

By writing the VC to access RTL signals via a virtual interface, the VC can then be connected to any instance of that interface as required, thus ensuring reusability.

Also the VC does not have to be declared local to a specific interface instance. This means the VC can be declared somewhere more convenient like a package.

## Example: Virtual Interface As Class Property

```

class myvc;
 virtual interface myif vif; // Property
 function new (virtual interface myif aif); // Argument
 vif = aif;
 endfunction // Assigned in constructor

 task write_data (input int datin);
 @(posedge vif.clk);
 vif.data = datin;
 endtask // Accessed in method

endclass

myif bus1(), bus2();
myvc c1, c2;

initial begin
 c1 = new(bus1); // c1 connected to bus1
 c1.write_data(5);
 c2 = new(bus2); // c2 connected to bus2
 c2.write_data(8);

```

```

interface myif (input clk);
 logic [7:0] data;
 ...
endinterface

```

- Virtual interface is declared as a class property.
- Used to access interface declarations.
- Must be assigned a value by one or more of the following:
  - Constructor
  - Method
  - Direct assignment
- The `interface` keyword in declaration is optional.

266 © Cadence Design Systems, Inc. All rights reserved.



A virtual interface can be a class property.

The default value of a virtual interface is `null`, and so it must be assigned before it can be used, either in a class constructor, a separate method or by directly assignment (`c1.pif = bus1`).

Again, the `interface` keyword is optional, but as class may be virtual and may contain virtual methods, the keyword is good for readability.

A virtual interface is the only reusable way a class instance can access the RTL signals of a design.

## Example: Virtual Interface Access in a Module

```

module test;
 interface myif (input clk);
 logic [7:0] data;
 ...
 endinterface

 myif bus1();
 myif bus2();
 comp DUT1 (bus1);
 comp DUT2 (bus2);

 virtual interface myif vif;
 task wrt(input int datin,
 virtual interface myif aif);
 @(posedge aif.clk);
 aif.data = datin;
 endtask
 endinterface

 initial begin
 vif = bus1;
 wrt(5, vif);
 wrt(8, bus2);
 end
...

```

Virtual interface:

- Default value is null.
- Can be declared as:
  - Module variable
  - Subroutine argument
  - Class property
- Can only be assigned or compared to:
  - An interface instance of the correct type.
  - Another virtual interface of the same type.
  - null

The `interface` keyword in the declaration is optional.

- Include for readability.



267 © Cadence Design Systems, Inc. All rights reserved.

The default value of a virtual interface is null – it must be assigned before it is used.

A virtual interface can only be assigned or compared to an interface instance or virtual interface of the same type, or to null.

A virtual interface can be declared as a local variable of a module, as an argument to a task or function or as a class property (see next slide). Virtual interfaces cannot be declared as `ref` arguments, although they behave as such.

Use of virtual interfaces promotes code reuse (write once, use often).

You can also iterate through an array of interface variables and pass each element in turn to a testbench task, so that a single task can access multiple instances of similar design blocks, each through its own interface instance.

## Virtual Interface Limitation



**Problem:** Nets and variables can be read via a virtual interface. But, Virtual interfaces cannot write to nets:

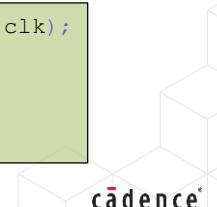
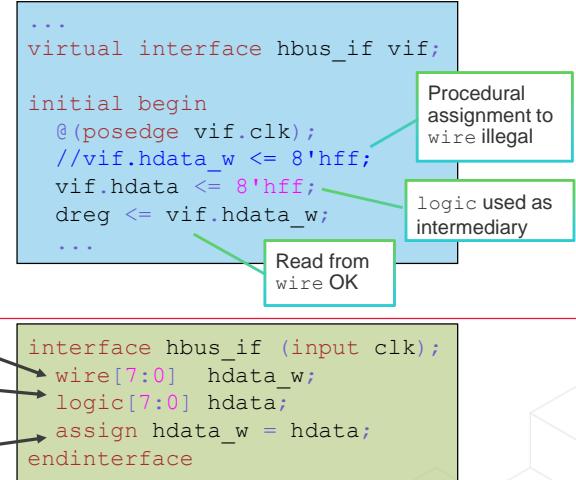
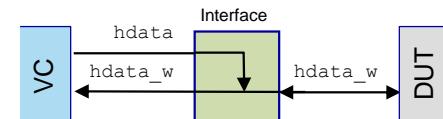
- As procedural assignment to a net (`wire`) is illegal.
- Bidirectional connections must be declared as `wire`.

**Solution 1:** Dual Representation

**Solution 1:** Utilize a clocking block

**Solution1** is to define two representations for an interface bidirectional connection.

- `wire (hdata_w)` for mapping to DUT `inout port` and for reading via virtual interface.
- `logic (hdata)` for writing from virtual interface.
- An `assign` statement is needed to drive logic writes onto wire.



Interface signals can only be driven via a virtual interface using procedural assignment. This is a problem for bidirectional connections. They must be declared as nets (`wire`) which precludes driving them via procedural assignment. One solution is for the interface to provide a way of driving the net procedurally. To do this, we need two representations of the net connection in the interface:

1. A `wire` (`hdata_w` in the HBUS interface): This is for *mapping* to the `inout port` of the DUT in the module instantiation and for *reading* the bidirectional connection, via the virtual interface, in the VC.
2. A `logic` (`hdata` in the HBUS interface): This is for *driving* the bidirectional connection, via the virtual interface, from the VC.

The interface must assign the `logic` variable to the `wire` signal in order to allow driven values from the VC to pass into the DUT.

There is an alternative to the above “dual representation” option, and that is to use a clocking block. See the next slide.

## Solution 2: Utilizing a Clocking Block

- Adding a clocking block to an interface allows better timing control.
- You can access signals via the clocking block with a virtual interface.
  - Procedural assignment to a net via a clocking block is allowed.
- Clocking block defines signal direction from the testbench perspective.
- You can reuse this direction information in an interface modport.
  - Use the `clocking` keyword in modport.
  - It can be mixed with the other direction or `import` information.

```
interface hbus_if (input clk);
 wire [7:0] hdata_w;

 clocking cb1 @(posedge clk);
 default input #1 output #3;
 input hdata_w;
 output hdata_w;
 endclocking

 modport tb (clocking cb1,
 ...);

endinterface : hbus_if
```

```
virtual interface hbus_if vif;

initial begin
 @ (posedge vif.clk);
 vif.cb1.hdata_w <= 8'hff;
 dreg <= vif.cb1.hdata_w;
 ...
...
```

Procedural  
assignment to  
clocking block  
wire

269 © Cadence Design Systems, Inc. All rights reserved.



You can define clocking blocks within an interface, and you can then access design signals using the timing of the clocking block. You can also access the interface clocking block via an interface instance or virtual interface using the following notations:

```
<interface instance>.<clocking block>.<signal name>
<virtual interface>.<clocking block>.<signal name>
```

Procedural assignment to a net is allowed via a clocking block; therefore, this example is an alternative to the “dual representation” option for driving bidirectional signals as described on the previous slide.

The specified direction of the clocking block signals is from the perspective of the testbench. Instead of re-entering this information for the testbench modport, you can reuse the clocking block direction information. In the modport, the direction information is replaced by a reference to the clocking block using the keyword `clocking`:

```
modport <modport name> (clocking <clocking block name>);
```

## Module Summary

You can verify designs at the transaction level instead of at the signal level

- This allows you to develop and debug at a higher level of abstraction:
  - Rather than wading through a “sea of waveforms” at the signal level

Transaction patterns are defined in sequences which execute on Verification Components (VCs):

- You typically have one VC for every protocol interface on your design

You can partition test development into stimulus (sequences) and structure (Verification Component):

- This partition separates stimulus development, done by test writers, from environment development

For reusability, VCs connect to the DUT ports through virtual interfaces

This methodology is most effective for designs with well-defined interfaces



Transaction-based verification is all about performing system-level development and verification with abstract transactions, without getting involved with an ocean of individual signal transitions.

Stimulus is defined as a pattern of transactions in a sequence, which are then executed on a Verification Component (VC). The VC processes the individual transactions of the sequence and converts them to and from DUT signal transitions. There can be many sequences defined for a specific VC, each focussed on achieving a particular verification goal.

Each VC is associated with a particular signal protocol, bus or interface, and you typically have one VC for every protocol or bus in your design. VCs for common bus protocols are usually available as Verification IP (VIP) from Cadence® or third-party vendors.

This structure allows you to partition test development into stimulus, which deals solely with transactions, and structure – the development of the VC.

This methodology works only as well as your design team adheres to a top-down design flow, where the design specification is defined before it is implemented.

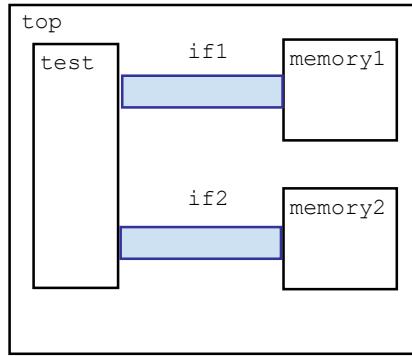
Transaction-Based Verification (TBV) has been used for many years. What is relatively new is the increasing tool support, in simulators and waveform viewers, for the creation, recording and analysis of transactions, and recently developed high-level verification languages such as *e*, SystemC and SystemVerilog, which support more sophisticated transaction-based verification by borrowing from the software world, powerful proven concepts such as object-oriented design and dynamic resource generation.



# Lab

## Lab 14 Using Virtual Interfaces

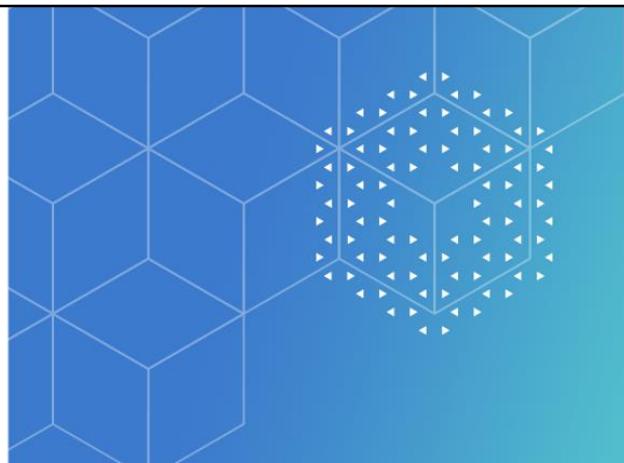
- Modify your memory module classes to use virtual interfaces and connect these to multiple instances of memories.



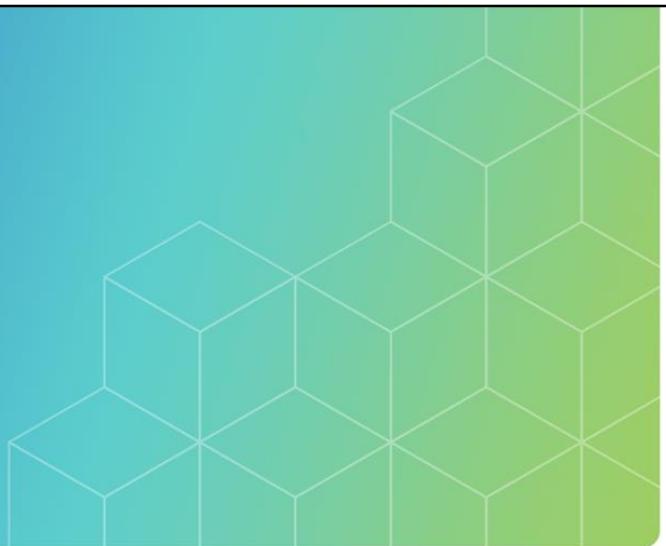
271 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*



## Covergroup Coverage



**Module** **18**

Revision **1.0**

Version **21.10**

Estimated time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

- Identify the role of functional coverage in verification
- Create simple coverage models using SystemVerilog covergroups
- Refine simple coverage using user-defined bins
- Implement cross-coverage
- Apply common options for modifying coverage behavior



*This page does not contain notes.*

## Structural and Functional Coverage



Structural metrics (code coverage) are tool-instrumented monitors to check:

- Execution of design blocks, lines or statements
- Assignment of values to variables



Functional metrics are user-defined scenarios to check:

- Assignment of *sequences* of values to variables
- Stepping of the design through control states (transactions)

Code coverage is:

- Instrumented by tool
  - Tool blindly focuses on individual items
- Less difficult to set up
- More difficult to analyze

|        |                                     |
|--------|-------------------------------------|
| add    | <input checked="" type="checkbox"/> |
| sub    | <input checked="" type="checkbox"/> |
| divide | <input checked="" type="checkbox"/> |

Code coverage

```
typedef enum bit[1:0]
 {add, sub, divide} op_t;
```

|                    |                                     |
|--------------------|-------------------------------------|
| add;sub;divide     | <input checked="" type="checkbox"/> |
| add;divide;sub     | <input checked="" type="checkbox"/> |
| add;add;add;add    | <input checked="" type="checkbox"/> |
| sub;sub;sub;divide | <input checked="" type="checkbox"/> |

Functional coverage



Functional coverage complements but does not replace code coverage.

274 © Cadence Design Systems, Inc. All rights reserved.



Code coverage can measure the number of times a line, statement, block or branch has executed by a given simulation run. More advanced code coverage can measure how many times a given subterm determines whether an expression is true.

For example, assume the following code is given:

```
if (a || b || c) ...
```

Code coverage could measure the number of times the expression was true due to a, b, c or any combination of these terms.

Code coverage is automatic and can be simply enabled with a simulation option.

Code coverage does not automatically check transitions between design states as the transition space is extremely large.

Transition or temporal checks are important. If you know that two consecutive sub-operations followed by a divide has caused an overflow error on previous designs, you need to check that stimulus has been applied to the current design. This is where user-directed functional coverage becomes imperative.

Functional coverage, as the name suggests, checks whether the test exercises the functionality of the design. You identify the critical combinations and sequences that should be exercised to verify the design functionality.

## Data-Oriented and Control-Oriented Functional Coverage



Data-oriented functional coverage defines a coverage model to capture value changes/signal transitions.



Control-oriented functional coverage is a sequence of control states.

- Covergroups for data-oriented functional coverage: covered in this module.

### Covergroup functional coverage

```
covergroup cg @(posedge clk);
 cpa: coverpoint addr
 {
 bins low = { [0:'h0F], 19 };
 bins mid[] = { 16, 17, 18 };
 bins high = { ['h14:'hFF] };
 }
 addrxvalid : cross cpa, valid;
endgroup
cg cg1 = new;
```

- SystemVerilog Assertions (SVA) for control-oriented functional coverage: covered in a later module.

### Property functional coverage

```
property req_gnt (cyc);
 @(posedge clk)
 req ##[cyc] gnt;
endproperty

Covreqgnt_3: cover property (req_gnt(3));
Covreqgnt_4: cover property (req_gnt(4));
Covreqgnt_5: cover property (req_gnt(5));
```



SystemVerilog offers both data-oriented functional coverage and control-oriented functional coverage. This module focuses on data-oriented functional coverage. Later modules introduce SystemVerilog Assertions (SVA), which provide control-oriented functional coverage.

SystemVerilog Assertions (SVA) are covered in more detail in a separate training class.

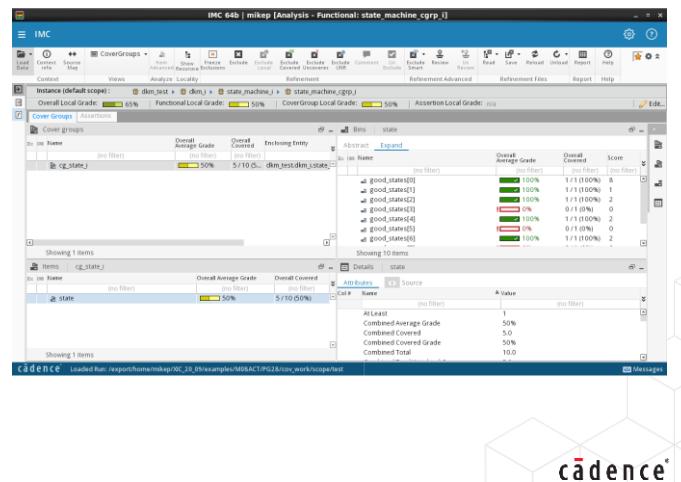
## What Is Data-Oriented Functional Coverage?



Data-oriented functional coverage is user-instrumented coverage using SystemVerilog cover groups to focus on design data.

The following are some key aspects of data-oriented functional coverage:

- It is user specified – not automatically inferred from the design.
- You write a test plan capturing the functionality to be tested:
  - Based on design specification and independent of actual implementation.
    - Less likely to verify “what I built” rather than “what I should have built.”
  - Needs to capture how to:
    - Test a given feature.
    - Check the test passes.
    - Measure the test is successfully applied (coverage).
- Coverage model is defined based on the test plan.
- The simulator counts how many times a variable hits the value or transition defined in the coverage model.



276 © Cadence Design Systems, Inc. All rights reserved.

Unlike structural coverage, functional coverage is not automatic. You must define a coverage model using SystemVerilog constructs to capture the value changes and signal transitions to confirm how thoroughly the design is being tested. A coverage model is usually developed as part of the design test plan.

Working with the design specification, you develop a test plan to define the following:

- How to test the design features (stimulus to be applied).
- How to confirm whether a given test passes (checkers or assertions).
- How to confirm whether the test is successfully applied (coverage).

Based on the test plan, you can define a SystemVerilog data-oriented functional coverage model, capturing the data values and value transitions to confirm how thoroughly the design is tested. During the simulation run, the simulator counts the value and transition occurrences and saves them in a database. Separate analysis tools allow you to interpret the coverage post-simulation.

## What Is a Cover Group?



A **Cover Group** is a user-defined type specifying a coverage model.

The user can construct multiple instances of that type in different contexts.

- Define a coverage model using the `covergroup` keyword and you can declare it in a package/interface/module/program/class.
- A cover group contains:
  - A **sampling event** whose coverage can be manually sampled – see later.
  - A **coverpoint** for each variable to track which can also track integral expressions.
- Define a variable of the `covergroup` name.
- Instantiate the coverage model using `new`:
  - One covergroup can have multiple instantiations in different contexts.

```
module example;
 logic clk;
 logic [2:0] address;
 logic [7:0] data;

 covergroup cg1 @ (posedge clk);
 c1: coverpoint address;
 c2: coverpoint data;
 endgroup : cg1

 cg1 cover_inst = new();
 ...
endmodule
```



You can declare a covergroup in a package, interface, module, program or class.

A covergroup contains the following:

- Sampling event: This is a SystemVerilog event expression which defines when variables are sampled for coverage. Sampling can be manual (see later) and so defining an event in the covergroup is optional. The expression could also be a block event expression indicating the start or the end of a given named block, task, function, or class method.
- Coverpoints: These define variables whose values must be sampled for coverage. Coverpoints can also be defined for specific variable values, value transitions and ranges of values.

A covergroup is a declaration only. An instantiation of the covergroup must be created to collect coverage. A covergroup is a class-like construct. Therefore in a module, you declare a variable of the covergroup type and then instantiate the model by calling `new`.

This allows multiple instantiations of a covergroup in different contexts to, for example, track data passing through a design. Every context must, however, have visibility of the covergroup declaration and the variables tracked in the coverpoints.

## What Is a Cover Point and Automatically Created Cover Point Bins?



A **Cover Point** is a user-defined item in a coverage model specifying an *expression* to cover and optionally a condition guarding its sampling. A **Coverage Bin** is a tool-defined or user-defined counter associated with a cover point value set, a cover point value transition set, or a cover cross tuple set.

**Automatic Cover Point Bin:** By default, the tool automatically creates a unique bin for each value.

- Each coverpoint generates a series of counters (bins):
  - Stored in a database.
- By default:
  - There is one for each coverpoint value.
    - Up to a preset limit.
  - Named `auto[<value>]`.
- At every sample point, the corresponding bin for the coverpoint value is incremented.

```
module example;
 logic clk;
 logic [2:0] address;
 logic [7:0] data;

 covergroup cg1 @ (posedge clk);
 c1: coverpoint address;
 c2: coverpoint data;
 endgroup : cg1

 cg1 cover_inst = new();
 ...
endmodule
```

address bins

278 © Cadence Design Systems, Inc. All rights reserved.



Each coverpoint in a covergroup creates a series of counters, called bins, which are stored in a coverage database. When coverage is sampled, the bin corresponding to the variable value is incremented.

By default, a coverpoint creates a single bin for every value in the variable range. These are called automatic, or implicit, bins and are named using the identifier `auto` and the value for the bin. There is a configurable limit on the number of automatic bins created. The default limit is 64.

For an enumerated coverpoint, there is one bin for each valid value. For an integral coverpoint variable, the number of automatic bins is at most  $2^M$  where M is the number of bits required to represent the variable. When  $2^M$  is greater than the default limit, the values are distributed as evenly as possible, with the last bin getting any extra values. Later slides show how to define explicit bins and how to change the default limit for automatic bin creation.

## Defining a Cover Point Bin for Values (Explicit Bins)



IEEE 1800-2012 19.5.1

- You can define the bins yourself:
  - To track only a subset of values.
  - To control what values increment which bin.
- Use the `bins` keyword, and provide a:
  - Bin name.
  - List of values or value ranges.
- With explicit bins, unlisted values are not tracked.
- Each coverpoint can have multiple `bins` clauses.

```
c1: coverpoint var1 {
 bins V = {1, 2, 5};
}
```

No semicolon at end of  
coverpoint with explicit bins

Bin V increments  
for var1 = 1, 2 or 5

Value list examples

|                   |                         |
|-------------------|-------------------------|
| { [0:5], 10 }     | - values 0-5 and 10     |
| { [0:5], [9:14] } | - values 0-5 and 9-14   |
| { 'h1, 'h2, 'hF } | - values 1, 2, 15       |
| { [1:9], [7:12] } | - range overlap allowed |
| { [16:\$] }       | - range 16 to max value |



One way of optimizing and managing coverage information is to define explicit bins to track only a subset of variable values or to group related values into the same bin. For example, instead of tracking every address value individually, it may be sufficient to track address ranges, and have a bin for all the small, medium and large address values sampled.

To define explicit bins, the semicolon at the end of the coverpoint is removed, and a series of `bins` clauses are added, enclosed in curly braces `{ }`. Note that each `bins` clause is terminated with a semicolon inside the braces, but there is not a semicolon after the closing brace `}` at the end of the coverpoint.

Each `bins` clause contains the `bins` keyword, a bin name and a list of values or value ranges corresponding to the bin. The list is a comma-separated series of individual values or ranges. For ranges, the maximum value only can be defined using `$`.

With explicit bins, only the values explicitly defined in the `bins` clause list are tracked. Coverage for unlisted values is not collected.

## Explicit Scalar and Vector Bins

You can define scalar or vector bins:

- Scalar bin
  - A single bin.
  - Incremented for all values in value range list.
- Vector bin (array of bins)
  - A unique bin for each value in the range list.
  - Incremented when variable takes the corresponding value.
- You can mix scalar and vector bins for the same coverpoint.

Scalar example

```
cs: coverpoint var1 {
 // bin V increments for 1, 2 or 5
 bins V = {1, 2, 5} ;
}
```

Creates a single bin cs.V

Vector example

```
cv: coverpoint var1 {
 // bins V[1], V[2] and V[5]
 bins V[] = {1, 2, 5} ;
}
```

[] for 3 values creates 3 bins  
cv.V[1], cv.V[2], cv.V[5]



A scalar bin is one bin that counts occurrences of any of the values in its list.

A vector bin is an array of bins. Using the unconstrained array declaration [] a separate bin is created for every unique value in the list.

## Example: Explicit Scalar and Vector Bins

Each coverpoint can define:

- Bin(s) for values that are illegal even if they appear in other bins.
- Bin(s) for values to be ignored even if they appear in other bins.
- A single bin for all the values in a range list.
- An unconstrained array giving a separate bin for each *unique* value in a range list.
- A fixed number of bins for *all* values in a range list:
  - Duplicates retained.
  - Ignored and illegal values removed *after* distribution.
- A default bin for all remaining values.

```
logic [3:0] var1;

ce: coverpoint var1 {
 // 1 bin for illegal values {0, 15}
 illegal_bins a = { 0, 15 };
 // 1 bin for ignored values {13, 14}
 // (value 15 is illegal)
 ignore_bins b = { [13:15] };
 // 1 bin for {2, 3}
 bins c = { 2, 3 };
 // 3 bins e[1], e[2], e[6]
 bins e[] = { [0:2], 2, 6 };
 // 2 bins - d[0] = {9,10,11,9}
 // - d[1] = {12}
 bins d[2] = { [9:11], 9, [12:15] };
 // 1 bin for {4,5,7,8},
 bins f = default;
}
```

Coverpoints may have multiple bins clauses defining both scalar and vectored bins. The options are as follows:

- `illegal_bins` specifies a bin for illegal values of the coverpoint variable. Once a value is defined in the list for illegal bins, it is automatically excluded from all other bins clauses in the same coverpoint, even if explicitly listed. The simulator issues an error when a sampled variable has an illegal value.
- `ignore_bins` specifies bins for ignored values. Once a value is defined in the list for ignored bins, it is automatically excluded from all other bins clauses in the same coverpoint, even if explicitly listed. Illegal bins takes precedence over ignore bins.
- A scalar bin is one bin for all the specified values in the list.
- For a vector bin of unconstrained size, each unique value in the list has its own bin. The standard implies this to mean duplicate values are not retained. Illegal and ignored values are removed after the values are distributed. Here, that results in an empty bin (`e[0]`), which is removed.
- For a vector bin of a specified size, the values are distributed as evenly as possible by order of appearance in the list, with the later bins getting any extra values. Duplicate values are retained, so can show up in multiple bins. Illegal and ignored values are removed *after* the values are distributed.
- The `default` keyword specifies bins for values that do not appear in any other bins. You cannot use the `illegal_bins` or `ignore_bins` keywords with `default`.

## Covergroup Coverage – How Many Bins?



How many bins are there in this covergroup?

What are their names?

```

typedef enum bit[2:0] {ADDI, SUBI, ANDI, XORI, JMP, JMPC, CALL} op_t;
typedef enum bit[1:0] {REG0, REG1, REG2, REG3} regs_t;

op_t opc;
regs_t regs;
logic[7:0] data;

covergroup cg @ (posedge clk);
 c1: coverpoint opc { bins op[] = { [ADDI:$] } ; }
 c2: coverpoint regs;
 c3: coverpoint data { bins low[] = { [0:'h0F] } ;
 bins high = { ['h10:'hFF] } ;
 }
endgroup

cg cg1 = new();

```

c1: 7 explicit vectored bins named  
c1.op[ADDI] to c1.op[CALL]

c2: 4 automatic vectored bins named  
c2.auto[REG0] to c2.auto[REG3]

c3: 16 explicit vectored bins named  
c3.low[0] to c3.low[15]  
and 1 explicit scalar bin named c3.high

Coverpoint c1 has 7 explicit vectored bins named c1.op[ADDI] to c1.op[CALL].

Coverpoint c2 has 4 automatic bins named c2.auto[REG0] to c2.auto[REG3].

Coverpoint c3 has 16 explicit vectored bins named c3.low[0] to c3.low[15] and 1 explicit bin named c3.high, a total of 17.

Total number of bins is 28.

## What Is a Cover Cross?



A **Cover Cross** is a user-defined item in a coverage model specifying a cross-product of cover points to cover and optionally a condition guarding its sampling.

You can track cross-products of:

- Coverpoints within the covergroup.
- Other scope variables.
  - Creates implicit coverpoint.
  - Participates in cover cross.
  - No separate coverage for variable.

Use the `cross` keyword and provide a:

- Cross name (optional).
- List of coverpoints and/or variables.

Bins are created for every cross-combination.

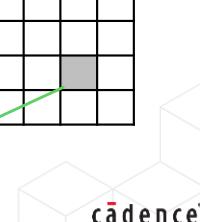
```
typedef enum bit[2:0] {ADDI, SUBI, ANDI, XORI, JMP, JMPC, CALL} op_t;
typedef enum bit[1:0] {REG0, REG1, REG2, REG3} regs_t;
op_t opc;
regs_t regs;

covergroup cg @(posedge clk);
 c1: coverpoint opc;
 c2: coverpoint regs;
 opcxreg: cross c1, c2;
endgroup
```

|               |               |               |               |              |               |               |
|---------------|---------------|---------------|---------------|--------------|---------------|---------------|
| c1.auto[ADDI] | c1.auto[SUBI] | c1.auto[ANDI] | c1.auto[XORI] | c1.auto[JMP] | c1.auto[JMPC] | c1.auto[CALL] |
|---------------|---------------|---------------|---------------|--------------|---------------|---------------|

|               |  |  |  |  |  |  |
|---------------|--|--|--|--|--|--|
| c2.auto[REG0] |  |  |  |  |  |  |
| c2.auto[REG1] |  |  |  |  |  |  |
| c2.auto[REG2] |  |  |  |  |  |  |
| c2.auto[REG3] |  |  |  |  |  |  |

Bin incremented for `opc=JMPC and regs=REG2`



cadence®

You can declare cover crosses of two or more already declared coverpoints within the covergroup and/or other integral variables. SystemVerilog automatically creates implicit coverpoints for cover cross variables which do not have a coverpoint. It does not report coverage data separately for implicit coverpoints.

Cross coverage effectively creates a multidimensional matrix of bins. By default, one automatic bin is created for every cross-combination with no limit on the number of bins created. In the example here, with two coverpoints, cross coverage is a 7x4 2-dimensional bin matrix containing 28 separate bins. Now you can see coverage showing the number of times each specific opcode was used with each specific register.

## What Are the Automatically Created Cover Cross Bins?



A **Coverage Bin** is a tool-defined or user-defined counter associated with a cover point value set, a cover point value transition set, or a cover cross tuple set.

For *cross-products* the tool automatically creates a unique bin for each tuple.

```

logic [1:0] a;
logic [3:0] b;
logic c;
covergroup cg @(posedge clk);
 bcp: coverpoint b {
 bins b1 = { [9:12] }; //one bin b1
 bins b2[] = { [13:15] }; //3 bins: b2[13], b2[14], b2[15]
 bins restofb[] = default; //9 bins: [0] ... [8] not in cross
 }
 ccp: coverpoint c; // two automatic bins
 axbxc: cross a, bcp, ccp; // 32 bins = a(4) x bcp(4) x ccp(2)
endgroup : cg

```

```

axbxc.<a.auto[0], b.b1, c.auto[0]>
axbxc.<a.auto[0], b.b1, c.auto[1]>
axbxc.<a.auto[0], b.b2[13], c.auto[0]>
axbxc.<a.auto[0], b.b2[13], c.auto[1]>
axbxc.<a.auto[0], b.b2[14], c.auto[0]>
...

```

Crosses created

Implicit  
coverpoint for a

284 © Cadence Design Systems, Inc. All rights reserved.



This example declares a cover cross of one variable (a) and two previously declared coverpoints (bcp, ccp).

- Variable *a* has four possible values for which four implicit bins are automatically created (*a.auto[0]* to *a.auto[3]*).
- Coverpoint *bcp* has one explicit scalar bin (*bcp.b1*) and one explicit vector bin containing three elements (*bcp.b2[13]* to *bcp.b2[15]*), for a total of four bins. Note that the vectored bin *bcp.restof[]* is not included in cross coverage as it uses the *default* keyword.
- Coverpoint *ccp* has two possible values for which two bins (*ccp.auto[0]* and *ccp.auto[1]*) are automatically created.

The cover cross thus has 32 automatic bins, one for each combination of coverpoint bins that make up the cross.

## Defining a Cover Cross Bin (Explicit Cross Bin and Select Expressions)



IEEE 1800-2012 19/6/1

You can create explicit cross coverage bins:

- `binsof` selects specific bins from a coverpoint.
- `intersect` filters bin selection to specified value ranges.
- You can use `!, &&, ||` on resulting selections.

|      | c1   |      |      |      |     |      |
|------|------|------|------|------|-----|------|
| c2   | ADDI | SUBI | ANDI | XORI | JMP | JMPC |
| REG0 | █    | █    | █    | █    |     |      |
| REG1 |      |      |      |      |     |      |
| REG2 |      |      |      |      |     |      |
| REG3 | █    | █    | █    |      |     |      |

```

typedef enum bit[2:0] {ADDI, SUBI, ANDI,
 XORI, JMP, JMPC,
 CALL} op_t;
typedef enum bit[1:0] {REG0, REG1,
 REG2, REG3} regs_t;

op_t opc;
regs_t regs;

covergroup cg @ (posedge clk);
 c1: coverpoint opc ;
 c2: coverpoint regs ;
 opxr : cross c1, c2 {
 bins x1 =
 binsof(c1) intersect {[ADDI:XORI]} &&
 binsof(c2) intersect {REG0, REG3};
 }
endgroup

```

x1 – track logical opcodes  
(ADDI–XORI) on REG0 and REG3

285 © Cadence Design Systems, Inc. All rights reserved.



SystemVerilog by default automatically creates a single bin for every product of the cover cross. It does not include any coverpoint bins declared with the `default` range or any declared with `illegal_bins` or `ignore_bins`.

You can explicitly declare cover cross bins enclosed in curly braces `{ }` immediately after the list of coverpoints to cross. As with coverpoint bins, each cross bins clause in the list is separately terminated with a semicolon, but the cross declaration line itself is terminated with the close curly brace `}`, not a semicolon.

In explicit cross coverage bins, you use the `binsof` keyword to select which coverpoint bins should participate in the cross. You can filter the bins selected to intersect with a list of values or value ranges using the `intersect` keyword. You can then perform conjunction, disjunction and negation operations on the resulting bin selections.

The example declares the cover cross bin `x1` to include the bins of coverpoint `c1` that intersect with range ADDI to XORI, and to include the bins of `c2` which intersect with the values REG0 and REG3. Therefore `x1` is a single cross coverage bin which is incremented for any combination of logical opcode (ADDI to XORI) on registers REG0 or REG3.

## Defining Cover Cross with Illegal and Ignored Cross Bins

Use `ignore_bins` to exclude bins from a cross:

- Even if selected elsewhere in the same cross.

Use `illegal_bins` to specify illegal bins:

- Even if selected or excluded elsewhere in same cross.

|    | c1   | ADD I  | SUB I  | AND I  | XOR I  | JMP    | JMPC   | CALL   |
|----|------|--------|--------|--------|--------|--------|--------|--------|
| c2 | REG0 | ██████ | ██████ | ██████ | ██████ | ██████ | ██████ | ██████ |
|    | REG1 | ██████ | ██████ | ██████ | ██████ | ██████ | ██████ | ██████ |
|    | REG2 | ███    | ███    | ███    | ███    | ███    | ███    | ███    |
|    | REG3 | ███    | ███    | ███    | ███    | ███    | ███    | ███    |

```

typedef enum bit[2:0] {ADDI, SUBI, ANDI,
 XORI, JMP, JMPC,
 CALL} op_t;
typedef enum bit[1:0] {REG0, REG1,
 REG2, REG3} regs_t;
op_t opc;
regs_t regs;

covergroup cg @(posedge clk);
 c1: coverpoint opc ;
 c2: coverpoint regs ;
 opxr : cross c1, c2 {
 bins x3 = ! binsof(c2) intersect {[REG2:REG3]};
 ignore_bins x4 = binsof(c1) intersect {[JMP:JMPC]};
 }
endgroup

```

x3 – select all bins not of REG2 or REG3, but ignore all JMP or JMPC bins (x4)

As with coverpoint bins, you can declare cover cross bins to be illegal or to be ignored. Ignored bins are removed from any other cover cross bin in the same cross. Illegal bins are also removed from any other cover cross bin in the same cross. In addition, the simulator must issue an error upon the occurrence of any illegal cross-product.

This example declares the cross coverage bin `x3` to exclude the bins of coverpoint `c2` that contain the values `REG2` and `REG3`. The cross coverage ignore bin `x4` removes all bins of `c1` which contain the values `JMP` or `JMPC` from any bins in the cross `opxr`.

Therefore, `x3` is a single bin which increments for any opcode other than `JMP` or `JMPC` on `REG0` or `REG1`.

## Defining Easier Cover Cross Bins

Complex cross expressions can be avoided using dedicated coverpoints:

- Define multiple coverpoints for required values or ranges.
- Cross coverpoints directly.

|      |      |      |      |      |      |
|------|------|------|------|------|------|
| c1nj | ADDI | SUBI | ANDI | XORI | CALL |
| c201 |      |      |      |      |      |
| REG0 |      |      |      |      |      |
| REG1 |      |      |      |      |      |

Select all opcodes except JMP & JMPC

Select REG0 & REG1

Select all bins not of REG2 or REG3, but ignore all JMP or JMPC bins

```
// declarations as before
op_t opc;
regs_t regs;

covergroup cg @(posedge clk);
 c1 : coverpoint opc;
 c1nj: coverpoint opc {
 bins opnj={[ADDI:XORI],CALL};
 }

 c2 : coverpoint regs;
 c201: coverpoint regs {
 bins rg01={REG0,REG1};
 }

 opxr : cross c1nj, c201;
endgroup
```

One bin – opxr.<opnj, rg01>



Cross coverage expressions can become complex and hard to interpret. An alternative to such complex expressions may be to define multiple coverpoints which select the ranges or values required, and simply cross these coverpoints directly.

Remember you can have as many coverpoints as you wish on the same variable, as long as each is uniquely named.

In the example above, we define a coverpoint `c1nj` which creates a single bin, `opnj`, for all the opcodes except `JMP` and `JMPC`. We also define a coverpoint `c201`, creating a single bin, `rg01`, for registers `REG0` and `REG1`. Then we can simply cross `c1nj` and `c201` to create a single cross coverage bin `opxr.<opnj, rg01>`. Note that this is a single bin – using vectored bins in the coverpoints would generate multiple cross coverage bins if this was required.

The disadvantage of this approach is that we have individual coverage on all the extra coverpoints. If these coverpoints are purely to help create cross coverage and this individual coverage is meaningless, we may need to filter away individual coverage by, for example, setting weight option to zero (see covergroup options).

## How to Use Cover Groups in Classes



To define the cover group in the class definition is an intuitive way to define the coverage model for the class.

1. Define the class member cover group instance.
  - The declaration creates an *instance* variable of an *anonymous* cover group type.
    - You cannot create another instance of that type.
  - The cover group definition can access any class properties – even if **protected** or **local**.
2. Construct the cover group instance.
3. Define cover group sampling.
  - For a design or test component class object, define a sampling event.
  - For a data class object, upon communicating the data object, call the cover group method `sample()`.

```
class covclass;
 logic [2:0] address;
 logic [7:0] data;

 covergroup cg1;
 c1: coverpoint address;
 c2: coverpoint data;
 endgroup : cg1

 function new();
 cg1 = new();
 endfunction
endclass

covclass one = new();
initial begin
 ...
 one.cg1.sample();

```

A class covergroup instance is not separately named

288 © Cadence Design Systems, Inc. All rights reserved.



When a covergroup is declared inside a class, the syntax to create a covergroup instance is different than that of a module covergroup.

In a class, the covergroup is declared as *normally*, but you do not create a property of the covergroup type to create an instance. In a class, the instance is created by calling `new` directly off the covergroup type name.

Each instance of the class will track coverage separately.

With class-based coverage, sampling is more likely to be manual, either by calling `sample()` directly from the covergroup of the class instance, or by defining a method to call `sample` (see later for list of coverage methods).

## Defining a Cover Point Bin for Value Transitions



IEEE 1800-2012 19.5.2

```

typedef enum bit[2:0] {ADDI, SUBI, ANDI, XORI, JMP, JMPC, CALL} op_t;
op_t opc;

covergroup cg;
 c1: coverpoint opc {bins adsu=(ADDI => SUBI);
 bins suan=(ADDI => SUBI => ANDI);
 bins su3= (ADDI,SUBI => ANDI); }
endgroup

```

Coverpoints can also track transitions:

- Single value change
- Sequence of values
- Multiple changes between arrays of values

It also allows sequence syntax similar to SystemVerilog Assertions (SVA):

- Repetition

|                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                                                                                                |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>(ADDI =&gt; SUBI)      : 1 transition (JMP =&gt; JMPC =&gt; CALL): sequence (SUBI =&gt; ANDI,XORI) : 2 transitions : (SUBI =&gt; ANDI) : (SUBI =&gt; XORI) (ADDI[*3]) : consecutive repetition (ADDI =&gt; ADDI =&gt; ADDI)</pre> | <div style="border: 1px solid black; padding: 5px; display: inline-block;">1 transition</div> <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-top: 10px;">1 sequence transition</div> <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-top: 10px;">(ADDI =&gt; ANDI), (SUBI =&gt; ANDI)</div> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

289 © Cadence Design Systems, Inc. All rights reserved.



Transitions are enclosed in parentheses and use `=>` to specify transition. Transitions can be defined as being between specific values (`ADDI => SUBI`). The bin will be incremented if the variable had the value ADDI on the last sample event, and SUBI on the current. Transitions can be longer than 2 values (`ADDI => SUBI => ANDI`). Arrays or ranges of values can also be used to indicate multiple changes. For example, (`ADDI, ANDI => JMP, JMPC`) defines 4 transitions: (`ADDI => JMP`), (`ADDI => JMPC`), (`ANDI => JMP`) and (`ANDI => JMPC`).

Transition bins can also use sequence syntax similar to that used in SystemVerilog Assertions. Specifically, repetition operators can be used to indicate repetition of a value over several sample events.

Syntax and rules for transition coverage can differ slightly than for value-based functional coverage. Please see the SystemVerilog Language Reference Manual (LRM) for more details.

## How to Specify Coverage Options



SystemVerilog provides options to control the behavior of:

- Covergroups
- Coverpoints
- Crosses

There are two types of options:

### 1. Type options

- Affect every instance (static)
- Set with `type_option`

### 2. Instance-specific options

- Can be set on individual instances
- Set with `option`

Type option

```
int a, b;
covergroup cg1 @ (posedge clk);
 c1: coverpoint a {
 type_option.comment = "a";
 }
 c2: coverpoint b;
endgroup : cg1

cg1::type_option.comment = "ab";
```

Instance option

```
int a, b;
covergroup cg1 @ (posedge clk);
 c1: coverpoint a {
 option.auto_bin_max = 10;
 }
 c2: coverpoint b;
endgroup : cg1

cg1 one = new;
one.c2.option.auto_bin_max = 256;
```

**cadence**

- The covergroup construct has a number of built-in options to control the behavior of covergroups, coverpoints and crosses. This information is usually passed into the coverage database for the analysis tool to act upon.
- There are two types of option:
  - **Type options:** These options affect every instance of a covergroup, similar to static properties of a class. They are set with `type_option` and can be embedded in coverpoint or cross definitions, or set externally using a scope resolution operator (`::`) just like a static method call.
  - **Instance options:** These options can affect the individual instance of a covergroup. They are set with the `instance` option and are usually applied procedurally to specific instances of covergroups. They can also be embedded in coverpoint or cross definitions to affect every instance or to redefine default values.



## Reference: Type-Specific type\_option Fields

| Type   | Identifier | Default Value | cg | cp | cc | Description                                                              |
|--------|------------|---------------|----|----|----|--------------------------------------------------------------------------|
| int    | weight     | 1             | ✓  | ✓  | ✓  | Weight of this element for calculation of covergroup or overall coverage |
| int    | goal       | 100           | ✓  | ✓  | ✓  | Target goal                                                              |
| string | comment    | ""            | ✓  | ✓  | ✓  | Comment to include in coverage report                                    |
| bit    | strobe     | 0             | ✓  |    |    | Whether to defer samples to end of time slot                             |

cg = allowed for covergroup  
 cp = allowed for coverpoint  
 cc = allowed for cover cross



weight, goal and comment options, which are set at the covergroup level, do not affect the values set at the coverpoint or cover cross level. You can also set weight, goal and comment options as instance-specific options. The next slide describes the instance-specific options.

- Set weight to modify the relative weight of the coverpoint or cover cross when calculating the enclosing covergroup coverage metric, and the relative weight of the covergroup when calculating the overall coverage metric, for a type-based coverage report.
- Set goal to modify the target goal for the element. The standard does not state how the vendor should use this field.
- Set comment to provide a string information for the coverage report.
- Set strobe to defer sampling to the postponed region at the end of the time slot, where \$monitor and \$strobe system tasks execute (see Appendix B).

## Reference: Instance-Specific option Fields

| Type   | Identifier              | Default Value | cg | cp | cc | Description                                                              |
|--------|-------------------------|---------------|----|----|----|--------------------------------------------------------------------------|
| string | name                    | unique        | ✓  |    |    | Instance name                                                            |
| int    | weight                  | 1             | ✓  | ✓  | ✓  | Weight of this element for calculation of covergroup or overall coverage |
| int    | goal                    | 90            | ✓  | ✓  | ✓  | Target goal                                                              |
| string | comment                 | ""            | ✓  | ✓  | ✓  | Comment to include in coverage report                                    |
| int    | at_least                | 1             | ✓  | ✓  | ✓  | Target count to consider bin as "covered"                                |
| int    | auto_bin_max            | 64            | ✓  | ✓  |    | Max number of auto bins                                                  |
| int    | cross_num_print_missing | 0             | ✓  |    | ✓  | Max number of uncovered cross bins to report                             |
| bit    | detect_overlap          | 0             | ✓  | ✓  |    | If true issue warning for values overlapping bins                        |
| bit    | per_instance            | 0             | ✓  |    |    | If true track coverage for each instance                                 |

You can set instance-specific options within a covergroup definition, and except for the `per_instance` option, you can also set them for each individual instance of the covergroup. Instance-specific options other than weight, goal and comment that you set at the covergroup level provide new default values for coverpoint and cover cross options.

- Set `name` to provide a name for the covergroup instance. The simulator generates a unique name for each covergroup instance for which you do not supply a name.
- Set `weight` to modify the relative weight of the coverpoint or cover cross when calculating the enclosing covergroup coverage metric, and the relative weight of the covergroup when calculating the overall coverage metric (instance-based coverage report).
- Set `goal` to modify the target goal for the element. The standard does not state how the vendor should use this field.
- Set `comment` to provide a string for the coverage report.
- Set `at_least` to modify the hit count for considering a bin covered.
- Set `auto_bin_max` to modify the maximum number of automatic bins.
- Set `cross_num_print_missing` to modify the number of uncovered cover cross bins that must be saved to the coverage database and printed in the coverage report.
- Set `detect_overlap` to have the simulator issue a warning when the values of two coverpoint bins overlap.
- Set `per_instance` to track coverage data for each covergroup instance as well as each covergroup type. You may want to track data on a per-instance basis if you differently parameterize each covergroup instance.

## Reference: Covergroup Methods

| Method                                                          | cg | cp | cc | Description                                        |
|-----------------------------------------------------------------|----|----|----|----------------------------------------------------|
| function void sample()                                          | ✓  |    |    | Force an immediate sample                          |
| function real get_coverage<br>([ref int cov, ref int tot])      | ✓  | ✓  | ✓  | Get coverage for type<br>– this is a static method |
| function real get_inst_coverage<br>([ref int cov, ref int tot]) | ✓  | ✓  | ✓  | Get coverage for instance                          |
| function void<br>set_inst_name(string)                          | ✓  |    |    | Set the instance name                              |
| function void start()                                           | ✓  | ✓  | ✓  | Start collecting coverage                          |
| function void stop()                                            | ✓  | ✓  | ✓  | Stop collecting coverage                           |

cg = allowed for covergroup  
cp = allowed for coverpoint  
cc = allowed for cover cross

These are methods of the covergroup base class that you call for specific covergroup instances. The `get_coverage()` method is a static method that you can also call for the covergroup type using the scope resolution operator.

- Use `sample()` to force an immediate sample.
- Use `get_coverage()` to obtain the current coverage percentage for all instances of the covergroup type. If you provide the optional reference arguments, the simulator places the covered bin count and total bin count in the referenced variables.
- Use `get_inst_coverage()` to obtain the current coverage percentage for only the one specific instance of the covergroup type.
- Use `set_inst_name()` to provide a new instance name for the covergroup instance.
- Use `start()` and `stop()` to start and stop collecting coverage data for the specific covergroup instance.

## Module Summary

SystemVerilog data-oriented functional coverage with covergroups offers:

- Specification of the sampling event
- Specification of variables to sample:
  - Automatic value bins
  - Explicit scalar, vector, ignore, illegal and default value bins
  - Explicit bins for transitions between values
- Specification of cross-products:
  - Automatic and explicit binning of cross counts
  - Filtering and combining cross-product bins
- Options to control covergroup, coverpoint and cover cross behavior
- Methods to start, stop and calculate coverage, and force sampling
- System functions and tasks to manage a coverage database



*This page does not contain notes.*

## Quiz



How many coverpoint bins are there in each of the cross c bins? What are their names?

Solution

```
bit [7:0] avec, bvec;
covergroup cg @(posedge clk);
cpa: coverpoint avec
{ bins a1 = { [0:63] };
 bins a2 = { [64:127] };
 bins a3 = { [128:191] };
 bins a4 = { [192:255] }; }
cpb: coverpoint bvec
{ bins b1 = {0};
 bins b2 = { [1:84] };
 bins b3 = { [85:169] };
 bins b4 = { [170:255] }; }
c : cross cpa, cpb
{ bins c1 = ! binsof(cpa) intersect {[120:200]};
 bins c2 = binsof(cpa.a2) || binsof(cpb.b2);
 bins c3 = binsof(cpa.a3) && binsof(cpb.b4); }
endgroup
```

c1: 4  
<a1,b1>, <a1,b2>,  
<a1,b3>, <a1,b4>

c2: 7  
<a2,b1>, <a2,b2>,  
<a2,b3>, <a2,b4>,  
<a1,b2>, <a3,b2>,  
<a4,b2>

c3: 1  
<a3, b4>



295 © Cadence Design Systems, Inc. All rights reserved.

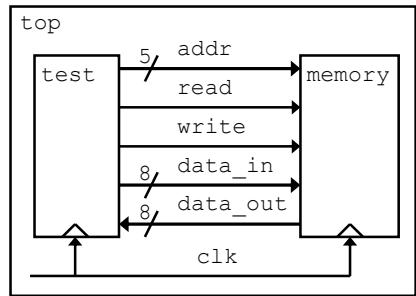
*This page does not contain notes.*

## Labs



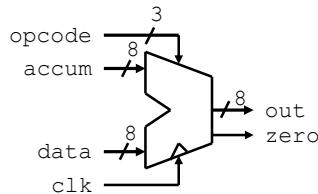
### Lab 15 Simple Covergroup Coverage

- Apply covergroup coverage to the memory testbench.



### Lab 16 Analyzing Cross Coverage (Optional)

- Apply cross coverage on an ALU design.



296 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*



## Queues and Dynamic and Associative Arrays (QDA)

**Module** **19**

Revision **1.0**

Version **21.10**

Estimated time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

- Use the following array constructs:
  - Dynamic arrays
  - Associative arrays
  - Queues
- Analyze the following array methods:
  - Locator
  - Ordering
  - Reduction



*This page does not contain notes.*

# Dynamic Arrays



IEEE 1800-2012 7.5

- Use a dynamic array when an array size must change during simulation.
  - Declared by leaving an unpacked dimension unsized [].
- A dynamic array doesn't exist until it is explicitly created during runtime.
- Use `new` to create array or change size.
  - `new[size]` – Creates array initialized to default values.
  - `new[size](array)` – Creates array initialized from existing array.
- Dynamic Array built-in methods:
  - `size()` returns array size.
  - `delete()` clears elements and sets size to 0.

Dynamic array of 8-bit logic

```

logic [7:0] dynarr[];
int index;
...
initial begin
 dynarr = new[8];
 for (int i=0; i<8; i++)
 dynarr[i] = i+1;
 index = dynarr.size(); //8

 //resize array keeping values
 dynarr = new[16](dynarr);

 index = dynarr.size(); //16
 // delete array
 dynarr.delete();

```

Create array

299 © Cadence Design Systems, Inc. All rights reserved.



Use a dynamic array when the array size changes during the simulation. Declare the dynamic array by leaving an unpacked dimension unsized. Create, and re-create, the array during run time using the `new` operator. As an argument to the `new` operator, you can provide an existing array of the same type to initialize the new array. Dynamic arrays have the `size()` and `delete()` methods, as well as the standard array manipulation methods and standard array query system functions.

## Dynamic Array Example

```

parameter num_vectors = 32; Dynamic array declaration
logic [7:0] dyn[]; Array allocation size:32
bit [4:0] addr;
logic [7:0] wdata, rdata;
initial begin
 dyn = new[num_vectors];
 $display("dyn size:%d", dyn.size());
 for (int i=0; i<num_vectors; i++) begin
 write_mem(i, i+5);
 dyn[i] = i+5; Dynamic array write
 end
 for (int i=0; i<num_vectors; i++) begin
 read_mem(i, rdata);
 if (rdata != dyn[i])
 $display("Error addr:%h, Wrote:%h, Read:%h", i, dyn[i], rdata);
 end
 dyn.delete(); Delete array
 $display("dyn size:%d", dyn.size()); Should be 0
end

```

300 © Cadence Design Systems, Inc. All rights reserved.



This example declares a dynamic array to store 8-bit logic vectors. During run time, it allocates space for 32 elements. In a loop, it writes data to a memory component and stores the data in the dynamic memory. In a second loop, it reads data from the memory component and compares it to the data in the dynamic memory. It deletes the dynamic memory when finished with it.

Effectively, we use the dynamic array as a scoreboard, to store the data we write to the memory, and then check the data read from the memory against the stored write data in the array.

## Associative Arrays



IEEE 1800-2012 7.8

Use an associative array when the data space is unbounded or sparsely populated.

- Declare the array by specifying a type instead of a size for its one dimension.
  - The key can be any nonreal type for which the equality operator is defined.
  - Array elements do not exist until you assign to them.
  - Array elements are “pairs” of associated key (address) and data values.
  - Elements are stored by key, ordered by key – see slide notes\*.

```

bit [3:0] aa1 [int]; // associative array of 4-bit 2-state
 // with index type of int
logic [7:0] aa2 [string]; // associative array of 8-bit logic
 // with index type of string
int aa3 [myclass]; // associative array of int
 // with index type of myclass
bit aa4 [byte]; // associative array of bit
 // with index type of byte

```

301 © Cadence Design Systems, Inc. All rights reserved.



Use an associative array when the address space is unbounded or sparsely populated. Declare the associative array by specifying a type instead of a size for its one dimension. The key can be any type for which a relative order can be determined, that is, any type to which SystemVerilog can apply a relational operator. An associative array is dynamic and elements do not exist until you assign them, therefore the associative array is, by default, empty at the start of simulation.

The address or index of an associative array is known as a key. When you write to an associative array, the key and its data are stored as a pair in the array. Subsequent writes to the array create new pairs which are ordered in memory according to the index type.

For an integral type, ordering is numeric. For a string type, ordering is lexicographical. For a class type, ordering is deterministic but arbitrary.

Reading an index value that has not yet been written returns the default value for the array data type.

The key can be wildcard (using \*) to permit indexing by any integral type. It is recommended to avoid using a wildcard key as it affects optimization (and hence performance) and can lead to confusing results if different data types are assigned.

\*- 1800-2012-7.8.5 - If the relational operator(<,>,==) is defined for the index type, the ordering is as defined in the preceding clauses. If not, the relative ordering of any two entries in such an associative array can vary, even between successive runs of the same tool. Index<5, index==10...etc.

## Associative Array Methods

| Method          | Description                                                                                                    | Syntax                                      |
|-----------------|----------------------------------------------------------------------------------------------------------------|---------------------------------------------|
| num() or size() | Returns the number of entries in an associative array                                                          | function int num();<br>function int size(); |
| delete()        | Removes a single entry if an index is specified<br>Removes all entries if no index specified                   | function void delete(input index);          |
| exists()        | Returns 1 if an element exists at the indexed location, otherwise returns 0                                    | function int exists(input index);           |
| first()         | Assigns the value of the first (smallest) index in the array to index<br>Returns 0 if array empty, otherwise 1 | function int first(ref index);              |
| last()          | Assigns the value of the last (largest) index of the array to index<br>Returns 0 if array empty, otherwise 1   | function int last(ref index);               |
| next()          | Assigns next value to index<br>Returns 1 if value exists, otherwise 0                                          | function int next(ref index);               |
| prev()          | Assigns previous value to index<br>Returns 1 if value exists, otherwise 0                                      | function int prev(ref index);               |

302 © Cadence Design Systems, Inc. All rights reserved.



Associative array access is usually by methods.

Associative arrays have the methods defined above as well as the standard array manipulation methods and standard array query system functions.

The `first()`, `last()`, `next()` and `prev()` methods return 2 values, an `int` as a return value of the function, and, if the method is successful, a key value by the argument `index`.

Remember that for some index types (e.g., class instances), the ordering used by `first/last/next/prev` is deterministic (i.e., repeatable over several simulations) but arbitrary (different simulators may use different ordering).

For `next()` and `prev()`, `index` is effectively an inout argument, in that the method returns the next highest key or previous lower key for the given argument. The value passed in does not have to exist in the array.

The `first()`, `last()`, `next()` and `prev()` methods cannot be used with associative arrays declared using a wildcard index type `[*]`.

## Example: Associative Array Lookup Table

```

logic [7:0] assoc[int];
bit [4:0] rand_a;
logic [7:0] rand_d, rdat;

initial begin
 repeat (32) begin
 success = randomize(rand_a, rand_d);
 write_mem (rand_a, rand_d);
 assoc[rand_a] = rand_d;
 end

 $display("Memory locations to check:%d", assoc.num());

 for (int i=0; i<=31; i++)
 if (assoc.exists(i)) begin
 read_mem(i, rdat);
 if (rdat != assoc[i])
 $display("Error at Addr:%h", i);
 assoc.delete(i);
 end
 $display("array size:%d", assoc.num());
end

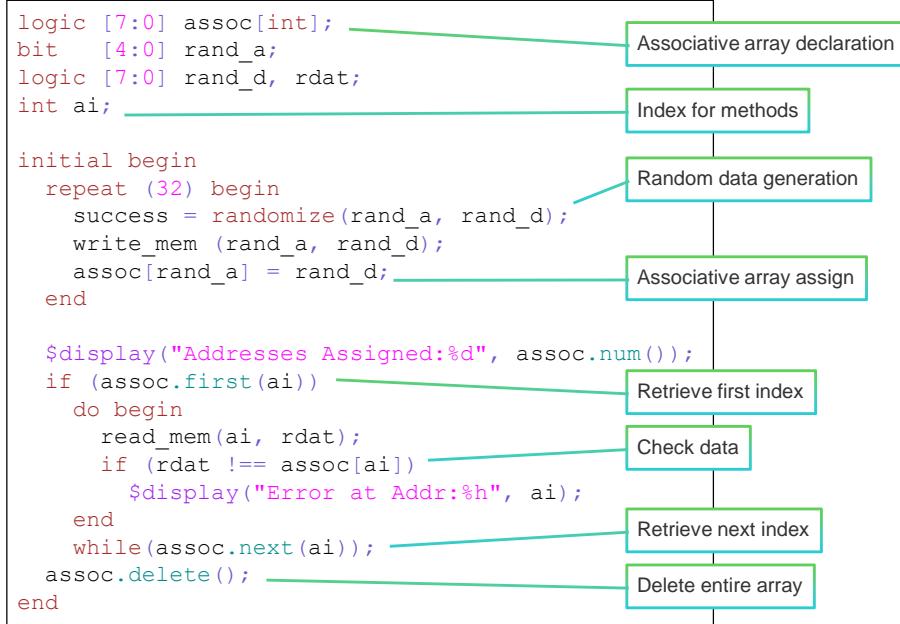
```

303 © Cadence Design Systems, Inc. All rights reserved.



This example declares an associative array to store 8-bit logic vectors with keys of the `int` type. In a loop, it randomizes address and data 32 times, writes the data to a memory component, and stores the address/data pairs in an associative memory. As the address vector is only 5 bits wide, it is very likely that some addresses are written multiple times. In a second loop, the example iterates through the potential address space. For each address, if the associative array has an entry for that address, it verifies that the memory component has the correct data, then deletes the entry for that address. Deleting entries individually is inefficient, but it allows us to verify that all locations have been checked by printing the final size of the array.

## Example: Associative Array-More Efficient Lookup Table



304 © Cadence Design Systems, Inc. All rights reserved.



This algorithm is more efficient. It utilizes associative array methods to iterate through only the assigned entries. It deletes the entire associative array at once when finished with it.

## Queues



IEEE 1800-2012 7.1.0

- Queues are a dynamically sized, ordered collection of elements of a declared type.
- A queue is declared by specifying \$ instead of a size for its one dimension.
  - Can optionally also include a maximum queue size.
- A queue supports access to all its elements as well as insertion and removal at the beginning or the end of the queue.
- Each element is identified by a number defining its position in the queue.
  - 0 represents the first location.
  - \$ represents the last location.

```
integer q_integer[$]; // queue of integers
logic [15:0] q_logic[$]; // queue of 16-bit logic
int q_int[$:2000]; // queue of int - max size of 2000
time q_time[$:10]; // queue of time - max size of 10
```

305 © Cadence Design Systems, Inc. All rights reserved.



Use a queue when the insertion and extraction order are important. Declare a queue by using the dollar (\$) character as the size of its one dimension. You can optionally limit the queue size by appending a colon (:) followed by a constant expression after the dollar character. You can use the \$ to reference the end of the queue and 0 to reference the start.

If a write to a bounded queue would exceed the maximum queue size, then a warning is issued and the write is discarded.

## Queue Methods

| Method       | Description                                    | Syntax                                       |
|--------------|------------------------------------------------|----------------------------------------------|
| size()       | Returns the size of the queue                  | function int size();                         |
| insert()     | Inserts an item at the index location          | function void insert(int index, qtype item); |
| delete()     | Removes item at the specified index            | function void delete(int index);             |
| pop_front()  | Removes an item in front of the queue [0]      | function q_type pop_front();                 |
| pop_back()   | Removes an item from the end of the queue [\$] | function q_type pop_back();                  |
| push_front() | Places an item at the front of the queue [0]   | function void push_front(q_type item);       |
| push_back()  | Places an item at the back of the queue [\$+1] | function void push_back(q_type item);        |

306 © Cadence Design Systems, Inc. All rights reserved.



Queue access is usually by methods.

The `.delete()` method with no index deletes the entire queue.

You can delete the queue contents and set the size to 0 by assigning an empty queue value to the queue variable:

```
qint = {} ; // delete queue
```

Reading an index value that lies outside the current queue bounds or is otherwise invalid returns the default value for the queue type.

Writing to an index value that lies outside the current queue bounds plus one or is otherwise invalid raises a warning and the write is suppressed.

Writing to an index value equal to the end of the queue plus one (`$+1`) is the equivalent of a `push_back()` operation.

## Example: Queues with Its Methods

```
int q_int[$]; // queue of unlimited size
int data;
initial begin
 q_int.push_front(0); // {0}
 q_int.push_back(1); // {0,1}
 q_int.push_front(2); // {2,0,1}
 q_int.insert(1, 3); // {2,3,0,1}
 q_int.insert(3, 4); // {2,3,0,4,1}
 q_int.delete(2); // {2,3,4,1}
 q_int.insert(2,5); // {2,3,5,4,1}
 data = q_int.pop_back(); // {2,3,5,4} data = 1
 data = q_int.pop_front(); // {3,5,4} data = 2
 while (q_int.size() > 0) // checking queue size
 data = q_int.pop_back(); // loop executes 3 times
 q_int = {0,1,2,3,4,5}; // direct load of queue - size = 6
 data = q_int[3]; // direct read - data = 3
 q_int = {}; // delete with empty queue assignment
end
```

307 © Cadence Design Systems, Inc. All rights reserved.



Queue methods support inserting and extracting elements. This example illustrates the use of several of the methods.

## Example: Queue with Indexing

```

int q_int[$]; // queue of unlimited size
int data;

initial begin
 q_int = {0,q_int}; // {0} push_front
 q_int = {q_int,1}; // {0,1} push_back
 q_int = {2,q_int}; // {2,0,1} push_front
 q_int = {q_int[0],3,q_int[1:$]}; // {2,3,0,1} insert
 q_int = {q_int[0:2],4,q_int[3]}; // {2,3,0,4,1} insert
 q_int = {q_int[0:1],q_int[3:4]}; // {2,3,4,1} delete
 q_int = {q_int[0:1],5,q_int[2:3]}; // {2,3,5,4,1} insert
 data = q_int[$]; q_int = q_int[0:$-1]; // {2,3,5,4} data = 1 pop_back
 data = q_int[0]; q_int = q_int[1:$]; // {3,5,4} data = 2 pop_front
 while (q_int.size() > 0) begin // checking queue size
 data = q_int[$];
 q_int = q_int[0:$-1]; end
 ...
end

```



A queue is an array, so if you really want to, you can still use unpacked array indexing and concatenation to insert and extract elements. As this example illustrates, you probably don't really want to. The queue methods are much more friendly.

## Array Manipulation Methods



Array Manipulation Methods are built-in methods that support searching, ordering, and reducing of arrays. These methods apply to any unpacked array, with the following exceptions:

- Locator methods cannot be applied to associative arrays that use the wildcard index type.
- Ordering methods cannot be applied to associative arrays.

- Locator

- Search array for elements or indexes that satisfy an expression.
  - Attached using `with`.
- Return a queue of those elements or indexes.

Locator Method Example

- Extract all elements from `array_int` greater than 5:  
`q_int = array_int.find with (item>5);`

- Ordering

- Reorder an array.
- Cannot be applied to associative arrays.

Ordering Method Example

- Sort `q_int` in ascending order:  
`q_int.sort;`

- Reduction

- Reduce an array of integral values to a single value.

Reduction Method Example

- Extract `xor` reduction of all elements in `q_int`:  
`var_int = q_int.xor;`

The locator methods return a queue of either indexes or elements. The return queue type is of the same time as the array index or element.

The ordering methods affect the array to which they are applied directly.

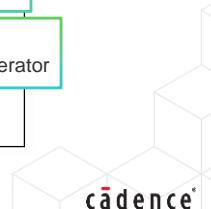
The reduction methods return a single integral value.

## Example: Array Locator Methods

| Locator Method                                            | Description                                                                                |
|-----------------------------------------------------------|--------------------------------------------------------------------------------------------|
| find(),<br>find_first(),<br>find_last()                   | Find element(s) satisfying relational expression<br>with expression is mandatory           |
| find_index(),<br>find_first_index(),<br>find_last_index() | Find index(s) of elements satisfying relational expression<br>with expression is mandatory |
| min(), max(),<br>unique()                                 | Find element(s) satisfying expression<br>with expression is optional                       |
| unique_index()                                            | Find indexes of elements having unique expression<br>with expression is optional           |

```
string qstr[$], res[$]; int idx[$];
initial begin
 qstr = {"A", "D", "C", "B", "A", "E", "C"};
 res = qstr.find(i) with (i>="D"); // {"D", "E"}
 res = qstr.max(); // {"E"}
 idx = qstr.find_last_index with (item>"C"); // {5}
 res = qstr.unique(); // {"A", "D", "C", "B", "E"}
```

310 © Cadence Design Systems, Inc. All rights reserved.



Syntax for methods:

```
function array_type[$] <locator> (array_type iterator = item);
function int_or_index_type[$] <index_locator>(array_type iterator = item);
```

Array locator methods apply to any unpacked array except associative arrays using the wildcard index type. The min(), max(), and both unique methods further require that relational operators be defined for the expression to be evaluated.

A with clause allows you to define an expression for the array method. The expression uses an iterator to refer to the current array element when processing the array. You can define your own iterator or use the default item identifier.

For example, the following are equivalent:

```
q_int = array_int.find with (item>5);
q_int = array_int.find(i) with (i>5);
```

Within the with expression you may refer to the current element's index by using the iterator's index() method and passing to it the dimension number for which you want the current index. The dimension number defaults to 1, that is, the first dimension.

## Example: Array Ordering Methods

| Ordering Method | Description                                                  |
|-----------------|--------------------------------------------------------------|
| reverse()       | Reverse array elements<br>Specifying with is an error        |
| sort()          | Sort array elements (ascending)                              |
| rsort()         | Sort array elements (descending)                             |
| shuffle()       | Randomize array element order<br>Specifying with is an error |

```
string qstr[$];
initial begin
 qstr = {"A", "D", "C", "B", "E"};
 qstr.sort; // {"A", "B", "C", "D", "E"}
 qstr.reverse; // {"E", "D", "C", "B", "A"}
 qstr.shuffle; // {"D", "C", "E", "A", "B"}
 ...

```

Random shuffle

311 © Cadence Design Systems, Inc. All rights reserved.



Syntax for methods:

```
function void ordering_method (array_type iterator = item)
```

Array ordering methods apply to any unpacked array except associative arrays. The sorting methods further require that relational operators be defined for the expression to be evaluated for sorting purposes. As this expression is by default the array element itself, for some array types you will want to specify some other appropriate expression.

## Example: Array Reduction Methods

| Method    | Description                          |
|-----------|--------------------------------------|
| sum()     | Sum of expressions                   |
| product() | Product of expressions               |
| and()     | Conjunction of expressions (bitwise) |
| or()      | Disjunction of expressions (bitwise) |
| xor()     | Parity of expressions (bitwise)      |

```
logic [7:0] narr [3:0] = '{8'ha, 8'h0, 8'hf, 8'h5};
int ia2d[2][2] = '{default:2};
int j;
...
j = narr.xor; // 0
j = narr.sum; // 1e
ia2d = '{'{1,2},{3,4}};
j = ia2d.sum with (item.product); // 14 ((4*3)+(1*2))
```

312 © Cadence Design Systems, Inc. All rights reserved.



Syntax for methods:

```
function expr_or_array_type reduction_method (array_type iterator = item);
```

Array reduction methods can apply to any unpacked array.

The corresponding operation must be defined for the type of the expressions that participate in the reduction. These expressions are by default the array elements, which then by default must be integral, but you can specify any expression for which the corresponding operation is defined.

For the ia2d array example:

- Each element of the ia2d array is itself a one-dimensional unpacked array of int, so cannot directly participate in the product. This example instead uses the value returned from calling the product method of each element in turn.

## Module Summary

Features of SystemVerilog arrays allow efficient and flexible definition, storage and access to large test data sets:

- Dynamic arrays are useful for contiguous data which varies in size during simulation
  - Track an undetermined number of dynamic objects
- Associative arrays are ideal for sparse or unbounded data sets
  - Look-up tables
- Queues are useful where insertion and extraction order are important
  - Synchronize data using FIFO and LIFO (stack) mechanisms



*This page does not contain notes.*

## Quiz



1. I need to validate the architecture of my processor stack...

A. Dynamic array

2. I need to validate the architecture of my packet data temporary storage...

B. Associative array

3. I need to validate the architecture of my instruction cache...

C. Queue



1. I need to validate the architecture of my processor stack... **Answer: Queue**

2. I need to validate the architecture of my packet data temporary storage...**Answer: Dynamic Array**

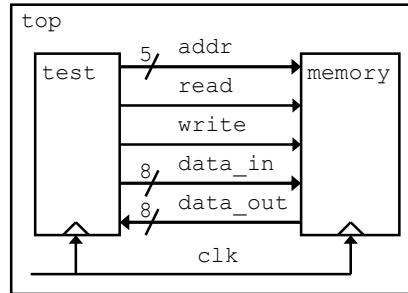
3. I need to validate the architecture of my instruction cache...**Answer: Associative Array**



## Lab

### Lab 17 Using Dynamic Arrays and Queues

- Implement a scoreboard using:
  - Dynamic Array
  - Associative Array
  - Queues



315 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*



## Introduction to Assertion-Based Verification (ABV)

**Module** **20**

Revision **1.0**

Version **21.10**

Estimated time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

- Gain an overview of Assertion-Based Verification (ABV)
- Discuss who writes assertions and for what purpose
- Describe the benefits and issues of using assertions



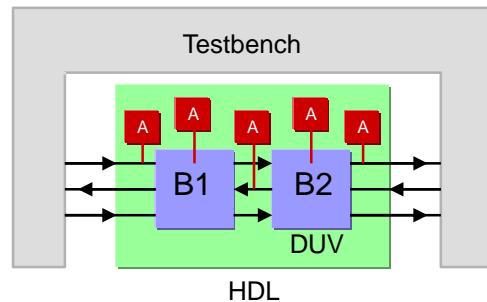
*This page does not contain notes.*

## What Is an Assertion?



An assertion is a directive to an EDA tool to verify that a property is always true.

- A property is a description of design behavior.
- We instruct the tool what to do with the property using verification directives:
  - assert, cover, assume, restrict
- An assertion is a check that a property is always true.
- During verification, assertions continually observe:
  - If a specific condition occurs, or
  - If a specific sequence of events occurs.
- Assertions offer a way to significantly enhance productivity for designers:
  - Find bugs earlier and more easily.



318 © Cadence Design Systems, Inc. All rights reserved.

An assertion is a directive to an EDA tool to verify that a property is always true (the terminology is that it “holds”). The simulator will report any failure of that design property to hold. Stating the assertion is the easy part. Crafting useful design properties is more difficult. However, it is far easier than debugging a broken system without the help of assertions.

The graphic illustrates where you can place an assertion. Note that you can place them on design inputs and design outputs and design interconnect and in the design blocks, as well as in a testbench.

The verification directives assume and restrict are used only for formal verification.

Cover and assert can be used in simulation or formal verification.

## Why Use Assertions?

Improves observability

- Automatically and constantly checks behavior
- Can detect hard-to-find bugs embedded deep in the design
- Isolates problem close to source
- Can be applied non-intrusively to legacy designs

Improves verification efficiency

- Can concisely and unambiguously document design intent
- Encapsulation and reuse
- Can detect bugs earlier in the design cycle
- Trap and exit on errors, saving simulation cycles
- Writing assertions helps designers to better understand their design and do more debugging

Assertions can be embedded in code, to “travel with” design from:

- Project to project – ideal for IP
- Tool to tool – ideal for design methodology integrity



You can place assertions anywhere in your design hierarchy, enabling you to monitor design behavior locally and highlight problems as soon as they occur at the source of the problem. Conventional testbenches need to propagate a problem to the design outputs to detect it, and then you have to trace the error back through the design, and probably back through time as well, to determine the source.

Assertions have the potential to detect problems during the early stages of testbench development, when the testbench might not yet be sufficiently robust to even propagate a problem to the outputs.

You can have the simulation terminate upon failure of critical assertions, thus permitting other projects to use the compute cycles that would otherwise be wasted.

Writing design properties encourages you to think more clearly about your design. This itself can highlight problems before you even finish the property. Overall verification efficiency is improved by designers producing better designs to begin with.

Your embedded assertions travel with your design as it is used in different projects, warning against unexpected or incorrect use of your design block.

Your embedded assertions also travel with your design from tool to tool, for example, from simulation to formal verification, avoiding duplication of the verification effort.

## Who Writes Assertions and Why?

| Who                   | Type of Assertion   | Reason                                                                        | Example                                                                                                                    |
|-----------------------|---------------------|-------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| System Architects     | Abstract properties | Ensure implementation meets functional spec.                                  | An asynchronous FIFO does not indicate full and empty at the same time.                                                    |
| RTL Designer          | Functional Property | Implementation has properties the RTL designer intended it to have.           | When Read and Write pointers both 0, the FIFO indicates empty.                                                             |
| Verification Engineer | Functional Coverage | Ensure test stimulus exercises the corner cases as required by the test plan. | Did you read from full FIFO, write to the last empty FIFO location, can you go from empty to full then back to empty, etc. |
| IP Designer           | Correct use of IP   | Indicate to IP User that the IP is not being used correctly.                  | FIFO was read when it was empty.                                                                                           |

320 © Cadence Design Systems, Inc. All rights reserved.



A system architect develops abstract properties that represent the desired behavior of the system at the functional level. At this point, the actual implementation is not known.

The block designer develops objective properties that represent their intended behaviour of the system at the RTL level.

As an IP designer, the RTL designer will want to embed input assertions in the design to detect future misuse of the design.

The verification engineer focuses on design functional coverage points, which should include that the testbench actually exercised the asserted properties. Thus, functional coverage is a fundamental part of an Assertion-Based Verification methodology.

## Issues with Assertions

- Assertions must be constructed with care.
  - Incorrectly specified assertions can give misleading results.
  - Debugging an assertion can be difficult.
- How do you know when enough assertions have been written?
- It is easy to shoot yourself in the foot using overly complex SVA constructs.
  - The syntax is “sugared”\* – simplicity of the syntax hides complex overlapping behaviors.
- Quality of test stimulus is critical.
  - The simulator can make only those checks which the test exercises.
  - Coverage metrics reveal which checks are exercised.
- Checking assertions consumes CPU cycles.
- Other issues...



The most prominent issue with the Assertion-Based Verification methodology is that you, the user, must craft the design properties. It is easy to miss nuances of design behavior, some corner cases, that really ought to be checked. There is also the issue of “how do you know when your property set is sufficient”.

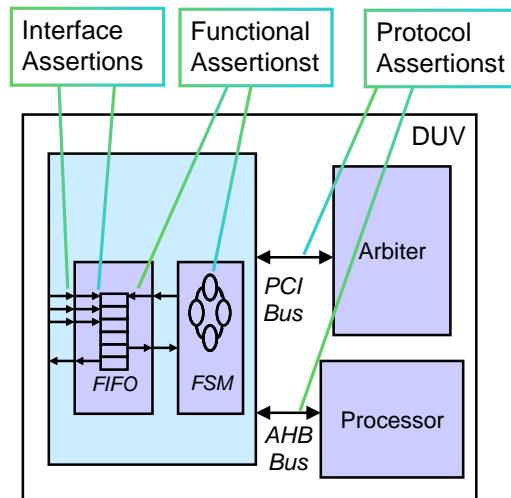
The simulator can check only those properties that the testbench exercises, so testbench quality is critical. This demands coverage metrics, which help to solve the “how do you know when you have tested enough” question, but do nothing for the “how do you know when your property set is sufficient” question.

## What Is Assertion-Based Verification (ABV)?



**Assertion-Based Verification** is the structured use of assertions to describe and verify design properties.

- Assertions monitor and report:
  - Expected behavior
  - Forbidden behavior
- Assertions are used by:
  - Static verification tools
    - No test vectors, formal (mathematical) proof
  - Dynamic verification tools
    - Dependent upon simulation test effectiveness (as measured using functional coverage)
- ABV also encompasses SVA constructs for defining functional coverage.



Assertion-based verification is a verification methodology that utilizes assertions. Assertions are not limited to just simulation tools. Formal verification tools, that do their verification statically, typically with very little helper test stimulus, if any, also use assertions. Coverage is an essential part of any verification activity whether one is using a simulator or a formal tool. SVA includes constructs for defining functional coverage.

ABV implies that we have a verification plan which details which properties are required to verify the design and which technologies the properties should be used with, for example, simulation, formal.

## HDL-Based Assertions (Assertions Are Not New...)

You have been “asserting” correct design conditions all along, using:

```
if (!good_condition) $display (error_message) ;

`ifndef SYNTHESIS
always @GNT
 @(negedge clk)
 if (GNT != 4'h0)
 begin: GNT_CHK
 integer cnt, idx;
 cnt = 0 ;
 for (idx = 0; idx <= 3; idx = idx+1)
 if (GNT[idx] != 1'b0)
 cnt = cnt + 1;
 if (cnt > 1)
 $display ("ERROR: Multiple GNT");
 end
`endif
```

323 © Cadence Design Systems, Inc. All rights reserved.



Verification personnel writing Verilog testbenches have been using an informal type of assertion right along. They write a process, that upon appropriate events, checks that some expression has the desired value, and if not, takes appropriate action. They can even write a check that spans multiple cycles, although that is more difficult, and thus rare.

## Why Is ABV Better Than HDL-Based Assertions?

Assertion-Based Verification goes *well beyond* HDL-based assertions:

- Standard language constructs to concisely express complex temporal behaviors
  - PSL and SVA
- Standard tool support taking advantage of the new constructs
  - Failure messaging
  - Statistics gathering
  - Debug features
- Definition of Functional Coverage



The use of assertions is not new. What is relatively new is the standardization of language constructs with which you can concisely express temporal design behaviors, and consistent support of those constructs, with failure messages, statistics collection and reporting, and user-friendly debug features.

## Module Summary

Complex properties are difficult to write using Verilog.

- A dedicated syntax like SystemVerilog Assertions helps greatly

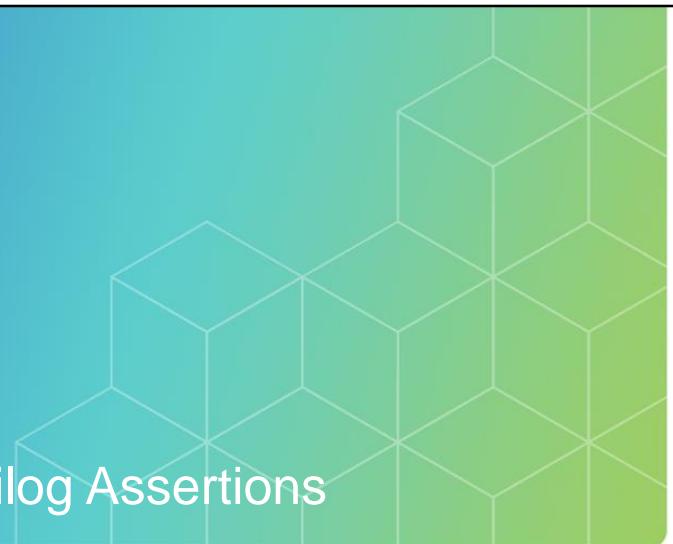
Assertions can reduce debugging time by identifying incorrect design behavior when and where it occurs. The Assertion-Based Verification methodology does the following:

- Captures specifications
- Captures design assumptions
- Documents interfaces
- Provides white-box visibility in simulation
- Checks properties of legacy designs without modifying existing code
- Supports advanced verification technologies:
  - Simulation, acceleration, emulation
  - Functional coverage
  - Static (formal) verification

325 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*



## Introduction to SystemVerilog Assertions

**Module** **21**

Revision **1.0**

Version **21.10**

Estimated time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

- Examine the structure of a concurrent assertion
- Create a simple instantaneous concurrent assertion
- Use implication to create conditional assertions
- Define sequences to create multicycle assertions
- Use simple cycle and sequence repetition to create more efficient assertions



*This page does not contain notes.*

## What Are Concurrent Assertions?



Concurrent assertions describe behavior that spans over time.

Unlike immediate assertions, the evaluation model is based on a clock so that a concurrent assertion is evaluated only at the occurrence of a clock tick.

- Concurrent assertions can be embedded in functional code and ignored by synthesis.
- They are supported by mainstream simulators and formal verification tools.
- You define and assert the functional properties of a design with Concurrent Assertions:
  - Boolean expression about some design behavior.
  - What it should or should not do.
  - Can be single or multiple cycle.
  - Uses Verilog syntax, yet ...
  - Mathematically precise and computationally efficient.
    - Sufficiently expressive to specify “real world” design properties.

```
RW_CHK : assert property (@(negedge clock) ! (wr_en && rd_en));
```

328 © Cadence Design Systems, Inc. All rights reserved.



We have learned that an immediate assertion is a procedural statement that some Boolean condition is true. The assertion is checked when the procedural statement executes.

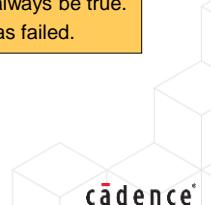
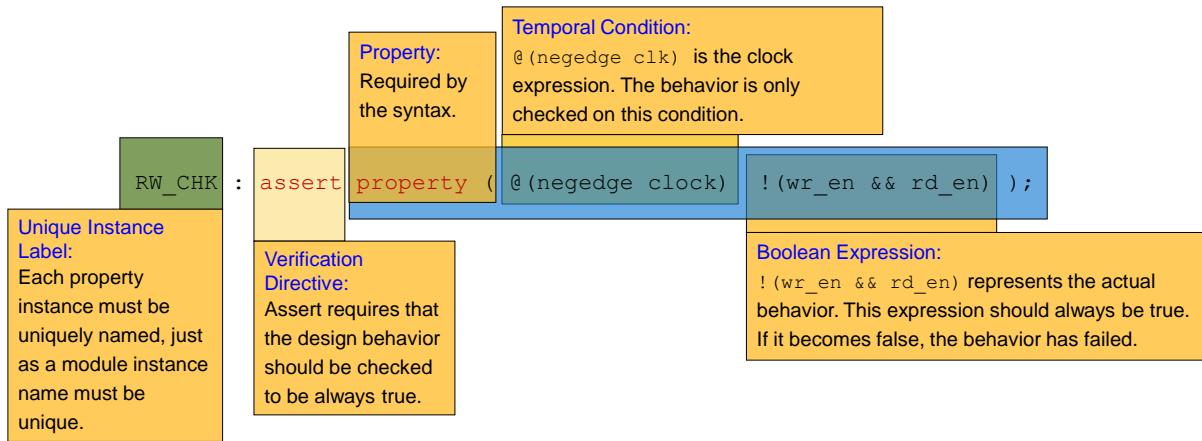
You typically place concurrent assertions outside procedural blocks to execute concurrently with the procedural blocks and utilize their own clock event. In simulation, this means each assertion runs continually in parallel and in the background.

A property is a design behavior that may span multiple cycles. It is a design requirement, similar to a synthesis constraint that the design must work with a 250MHz clock. SystemVerilog has special operators you use to develop concise property expressions to represent complex multicycle design behavior.

## Concurrent Assertion Structure

Let us examine the following required design behavior.

"wr\_en and rd\_en are never both high at the negative edge of the clock"



This is one form of a concurrent assertions. There are other forms as you will see.

- The assertion starts with an optional statement label and the keyword **assert**. As designs typically contain hundreds if not thousands of assertions, a user-defined label will help with assertion management and debugging. The **assert** keyword tells the simulator to check this property.
- The **property** keyword defines this is a concurrent as opposed to an immediate assertion. The two terms of the property must be enclosed in parentheses.
- The first part of the property is the temporal condition which defines the event expression upon which the assertion is evaluated. All properties must be clocked, although the temporal condition can be defined elsewhere.
- The second part of the property is the Boolean expression, defining the design behavior which will be checked at the temporal expression. If the Boolean expression evaluates to 1, this simple assertion passes. If the expression evaluates to any other value (0, X, Z), the assertion fails and an error message is reported.

The assertion label must be unique in the current scope. It is a good practice to use an uppercase label to avoid the likelihood of clashes with any other local identifiers. If you fail to label your assertions, the simulator just reports the scope of the assertion if it fails.

## Concurrent Assertions Placement

- Properties are declared as normal SystemVerilog declarations:
  - Usually in a module or interface.
- Properties are asserted as a concurrent statement:
  - Usually in a module or interface.
  - Also legal inside an always or initial block.
    - Strong recommendation not to do this.



What about tools that cannot compile SystemVerilog?

- Use text macro conditional compilation.
- Or use `bind` construct.

```
module mod1
 (input logic clk,
 ...
 output logic en1, en2);

 always @ (negedge clk)
 begin
 ...
 (en1, en2) <= sel[1:0];
 ...
 end

`ifndef USING_OLD_TOOL
 property RW_CHK_CLK;
 @(negedge clk) en1 || en2;
 endproperty

 A1: assert property (RW_CHK_CLK);
`endif

endmodule
```

330 © Cadence Design Systems, Inc. All rights reserved.



You can declare a property, like a normal SystemVerilog type declaration, in a compilation unit scope, package, interface, module or program.

The property is asserted as a procedural statement, only in an interface, module, or program. You can also assert a property in an always or initial block, although there are many issues with doing this, which are covered in the full SystemVerilog Assertions class.

If your tool flow contains tools which cannot compile SystemVerilog, then there are two common options:

- Declare your own text macro and use conditional compilation (``ifndef`) to conceal the code from the non-compliant tool.
- Use the SystemVerilog `bind` construct. This allows you to place assertions within a module, and then instantiate the assertion module into an existing design module from a higher hierarchical scope, usually the testbench, without modifying the design module. Bind is covered in the full SystemVerilog Assertions class.

Concurrent assertions can also be placed inside of procedural blocks.

However, this makes their behavior very difficult to understand.

It is strongly recommended that you don't place concurrent assertions inside of an `always` or `initial` block.

## Defining a Simple Property



IEEE 1800-2012 16.5

en1 || en2;

- Define a design behavior as a Verilog Boolean expression.
- Any expression allowed in an `if` condition can be used, including:
  - Functions
  - Out Of Module references
- Subject to normal Verilog restrictions, particularly:
  - Case sensitivity
  - Naming rules

((a + b) >= 42) && (c || d);

(num\_set(sel\_vec) == 1) && valid;

num\_set is a  
user-defined  
functionext

A property expression can be simply a Boolean expression. Use parentheses where needed to appropriately group the operands and to enhance readability.

## Naming and Asserting the Property

You have two options:

- Declare and instance the property on the same line.

```
ASSERT1 : assert property (@(posedge clk) en1 || en2);
```

- Declare and instance the property in separate statements:

- Gives more flexibility
- Property can have arguments

Always label assertions

```
property RW_CHK;
 @(posedge clk) en1 || en2;
endproperty

ASSERT1 : assert property (RW_CHK);
```

Parentheses for named property required

Instance labels are used to hierarchically reference the assertion.

332 © Cadence Design Systems, Inc. All rights reserved.



Assertions can be defined in 1 of 2 ways:

- Single statement assertion – Define the Boolean condition as an un-named property and assert the property directly.
- Named property declaration – Declare a property to define the desired behaviour and assert the property in a separate statement. The property must have a name, and the assertion statement can be labelled.

The assertion label and property name can be used by the verification tools to identify the assertion. The name and label must be legal Verilog names that do not conflict with other identifiers in the same scope. Using uppercase identifiers for both label and name is a common convention to avoid name clashes.

The named property declaration form is more flexible for complex assertions. By separately declaring and naming properties, you can:

- Use the properties as building blocks in more complex assertions.
- Reuse properties for assertions, functional coverage and formal verification.
- Declare property arguments, allowing a single design behavior to be applied to different signals.

We will use the named property form throughout the course and assume the property is separately asserted if not explicitly shown.

## Clocking the Property

- Properties are evaluated upon standard Verilog clocking events.
  - `@identifier or @ (event_expression)`
  - Can also use SystemVerilog enhancements (such as `iff`)
- All assertions must be clocked!
- Clocking expression does not need to be:
  - A design hardware clock
  - Periodic
- Good clocking expressions can greatly simplify properties.
- A default clock can be specified.

```
property RW_CHK_CLK;
 @ (negedge clk iff VALID) en1 || en2;
endproperty
```

```
property P3;
 @ (negedge addr_en) addr <= 7;
endproperty
```

333 © Cadence Design Systems, Inc. All rights reserved.



All assertions must be clocked. You can specify the clock in a sequence expression that the property uses or in the property specification.

The clocking event is typically just a Verilog event expression, exactly the same as that used in an RTL `always` procedure, or in embedded event control in verification code. SystemVerilog event expression enhancements, such as `iff` (if-and-only-if) qualifiers, can also be used.

Default clocks can be defined for properties which do not have an explicit clock in their declaration.

This is done with a clocking block and the keyword pair “default clocking”.

The default clock applies only to the scope in which it is declared.

## How Is an Assertion Evaluated?



Assertions are sampled, clocked, and evaluated at strictly defined points in the simulation cycle.

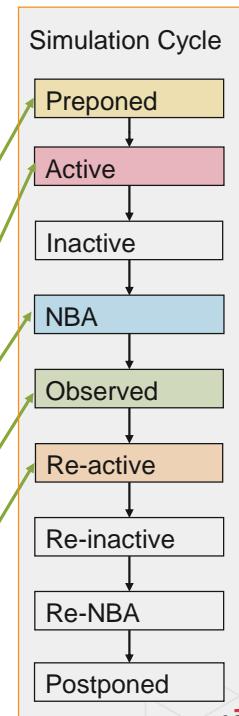
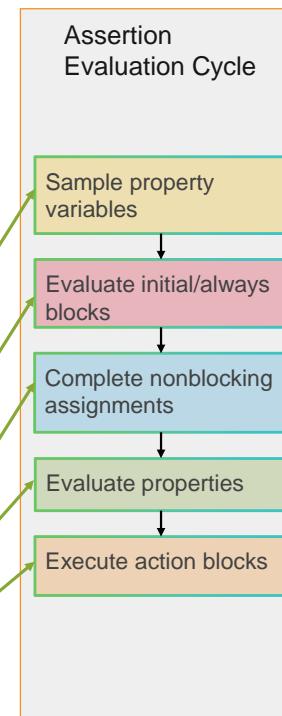
- Property variables are sampled in the *prepended* region.
- Properties are evaluated in *observed* region using *prepended* values.

```
property RW_CHK_CLK;
 @ (negedge clk)
 en1 || en2;
endproperty

always @ (negedge clk)
 (en1, en2) <= sel[1:0];

always #10 clk <= !clk;

A2 : assert property (RW_CHK_CLK)
 else
 $warning("RW_CHK_CLK failed");
```



334 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

A SystemVerilog simulation time instant is divided into several regions. The regions separate various activities to help avoid races between the writing and reading of a variable.

Assertions are sampled, clocked, and evaluated at strictly defined points in the simulation cycle. Variables in a property are sampled at the beginning of the simulation cycle in the *prepended* region. In the Active, Inactive and NBA regions, procedures are evaluated and variables updated. It can take several passes through these regions until you reach a steady state at this point in simulation time.

Then properties are evaluated in the *observed* region. If the temporal condition is currently true, such as a rising edge of `clk` occurred in the NBA region, then the assertion is triggered. However, the property is evaluated using the sampled variable values from the *prepended* region, which might be different from the current value of these variables in the *observed* region, that is, they were updated in the Active or NBA region.

This may seem counter-intuitive, but if you consider gate-level behavior, where the input sampled by a register is that in the setup time before the clock edge, then it makes more sense.

In the example above, `en1` and `en2` are updated on the `negedge` of `clk`, and the property `RW_CHK_CLK` is evaluated on the `negedge` of `clk`. `en1` and `en2` are sampled in the *prepended* region. `clk`, `en1` and `en2` are updated in the NBA region, and if `clk` fell, the property is evaluated in the *observed* region, using the sampled values of `en1` and `en2`. Therefore, assertion evaluation uses the preclock values, and the values of `en1` and `en2` in the *observed* region (post-`clk` values) might be different from the values used in the property evaluation (pre-`clk` values).

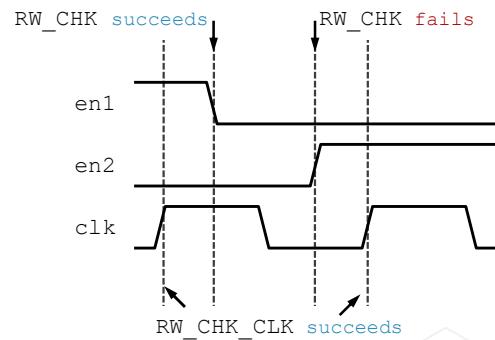
This diagram of the time slots is greatly simplified and omits all the regions used for PLI callbacks.

## Example: Assertion Evaluation

- RW\_CHK evaluated when en1 changes or en2 changes.
  - 2 evaluations, 1 failure
    - Using preponed values before change.
- RW\_CHK\_CLK evaluated on the positive edge of clk.
  - 2 evaluations, 2 successes
    - Using preponed values before rising edge clk.
- There are many things to understand to avoid problems if signals used as clocking signals are also referred to in the property behavior.
- Strongly recommend not to create such properties, for example, RW\_CHK.

```
property RW_CHK;
 @(en1 or en2) en1 || en2;
endproperty

property RW_CHK_CLK;
 @(posedge clk) en1 || en2;
endproperty
```



335 © Cadence Design Systems, Inc. All rights reserved.



Clock asynchronous properties on any change of any signal used in the property expression.

Clock synchronous properties on the appropriate edge of the appropriate clock signal.

Property expression evaluation uses the sampled values, that is, the values before the clock event.

A side effect of using sampled values in assertions rather than instantaneous values is that conflicting behavior is not immediately detected. Notice that the failure of RW\_CHK occurs some time after the incorrect behavior started. Indeed if we end the simulation before en2 changes from 0->1, then we would never be aware that there was a problem.

## What Is Assertion Binding?



Binding allows to specify one or more instantiations of an assertion module, interface, program, or checker without modifying the code of the target with bind construct.

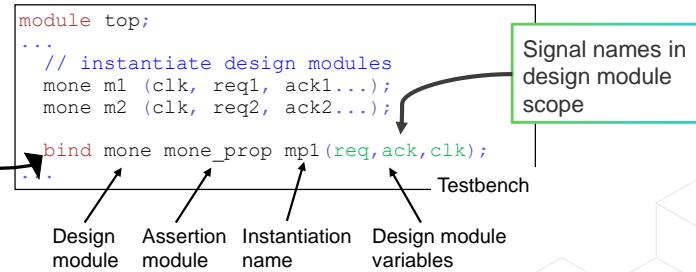
- The bind statement can be placed literally anywhere.
- Instantiates an assertion module in instance(s) of a hierarchical design module.
- Advantages:**
  - No changes to design module
  - Easier to modify and update
  - Enhances reusability
- Can bind to all or specific instances of a design module:
  - To bind to every instantiation of mone  
bind **mone** mone\_prop...
  - To bind only to instantiation m1 of mone  
bind **mone:m1** mone\_prop...

```
module mone_prop(input a,b,clk);
 property P1;
 @(posedge clk) $rose(a) |=> b;
 endproperty
 A1 : assert property (P1);
endmodule
```

Assertion module

```
module mone(input logic clk, req,
 output logic ack,
 ...);
 ...
 mone_prop mp1(req,ack,clk);
endmodule
```

Design module



336 © Cadence Design Systems, Inc. All rights reserved.

Bind allows you to insert an instantiation of an assertion module (or indeed any module) into any design module in the hierarchy, by adding a bind construct to, typically, the top-most module in a design.

Hence, you can add modules into a design hierarchy from the testbench without having to edit the design modules.

Syntax for bind:

- Bind
- mone //design module name/instance
- mone\_prop //assertion module
- mp1 //instantiation of assertion module
- req,ack,clk); //map to variables in design module

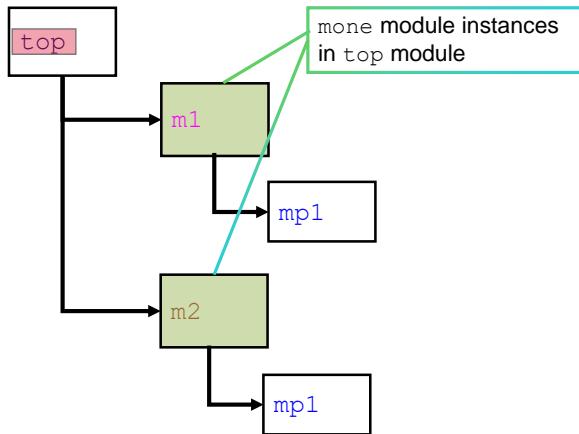
You can bind an assertion module to a specific instantiation of a design module by using the instantiation name instead of the module name:

```
bind m1 mone_prop mp1(req,ack,clk);
```

## Conceptual Illustration of Assertion Binding

Equivalent to an instantiation of the assertion module inside top.m1 and top.m2:

- Linked into design by way of hierarchical path names.



```

module mone_prop(input a,b,clk);
 property P1;
 @(posedge clk) $rose(a) |=> b;
 endproperty
 A1 : assert property (P1);
endmodule

```

Assertion module

```

module mone input logic clk, req,
 output logic ack,
 ...
endmodule

```

Design module

```

module top;
 ...
 // instantiate design modules
 mone m1 (clk, req1, ack1...);
 mone m2 (clk, req2, ack2...);

 bind mone|mone_prop mp1(req,ack,clk);
 ...

```

Bind creates an implicit instantiation of an assertion module and links this to all instances, or a specific instance, of a design module via hierarchical variable names.

## Application: Assertion Binding

Connecting assertion module port signals to internal signals of the design module using bind.

### Additional Advantages

- Internal signals of Design module can be connected to port signals of Assertion module.
- If names don't match, the .name notation can be used for connection.

```
module fifo_check(input clk, data_in,data_out,
half,empty/full);
property FIFO_EMPTY;
...
endproperty
FIFO_EMPTY_A1 : assert property (P1);

property FIFO_FULL;
...
endproperty
...
endmodule
```

Assertion module

```
module dut(input logic clk, data_in,
output logic data_out
...
logic half, empty, full;
...
endmodule
```

Design module

```
module top;
...
// instantiate design modules
dut m1 (clk,data_in,data_out);
bind m1 fifo_check mp1 (clk,data_in,data_out,half,empty/full);
...

```

These signals are internal signals  
of design module



Here, DUT module has internal vectors named 'half', 'empty' and 'full'. And fifo\_check module has ports with same names, but these vectors are not declared in the top module, because these vectors do not exist in top module. These vectors are present only in fifo\_check and dut modules. Since, fifo\_check module is indirectly instantiated into the dut module through the use of the 'bind' mechanism, and does not really exist in top module, there is no need to make the declarations of half, empty and full in top module.

Assume the port names of the fifo\_check module have different names that have to be connected to half, empty and full signals of the dut module. Then, you can simply use dot name notation to connect those signals to all remaining signals using the dot star connection.

Again, it is important to remember that, if it is permitted to instantiate fifo\_check module directly in the dut module, then bind construct is not required at all.

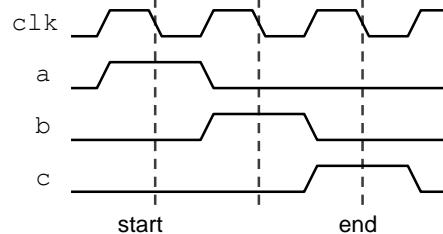
## What Is a Sequence Expression?



**A Sequence Expression** describes a series of one or more cycles of design signal states (each cycle described by a Boolean expression).

- Simple Boolean properties are instantaneous:
  - Single pass/fail test at the evaluation point
- Multi-cycle properties are described using sequences:
  - Series of Boolean equations
  - Evaluated in successive clocking cycles
  - Each cycle separated by `##N`
- Sequences are building blocks of properties.
- There is a wide range of operators and features to construct complex sequences:
  - *if-then* implication
  - Repetition
  - Composition operators

```
Sequence
@ (negedge clk)
a #1 b #1 c;
```



339 © Cadence Design Systems, Inc. All rights reserved.

Simple Boolean properties are instantaneous – they will pass or fail at a single evaluation point. However, most properties are temporal. They require different checks to be performed over different cycles to confirm the design behavior.

A sequence is a series of Boolean conditions over successive cycles. You use `##` (cycle delay operator) to separate one evaluation cycle from the next. The example sequence:

`a ## b ## c`

is evaluated as condition `a` true in the first cycle, followed by condition `b` true on the next, followed by condition `c` true on the next cycle. If each cycle of a sequence is true, then the whole sequence is true. The first condition which is not true causes the sequence to fail on that cycle.

Sequences are used as building blocks of properties and there are a wide range of operators and constructs to define conditional, repeated, overlapping, parallel, alternate and nested sequences to help create properties.

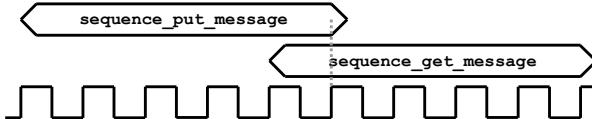
## Defining a Cycle Delay



IEEE 1800-2012 16.7

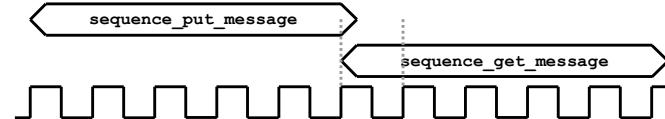
Delay of 0 cycles, AKA **Sequence Fusion**

```
sequence_put_message ##0 sequence_get_message
```



Delay of 1 cycle, AKA **Sequence Concatenation**

```
sequence_put_message ##1 sequence_get_message
```



340 © Cadence Design Systems, Inc. All rights reserved.



You can delay sequence evaluation by a constant number of cycles. The delay can be any integral number between zero and the end of simulation.

A delay of 0 cycles, that is, sequence fusion, places the first cycle of the following sequence coincident with the last cycle of the preceding sequence.

A delay of 1 cycle, that is, sequence concatenation, places the first cycle of the following sequence after the last cycle of the preceding sequence.

## Implication ( $| ->$ : Same Cycle)

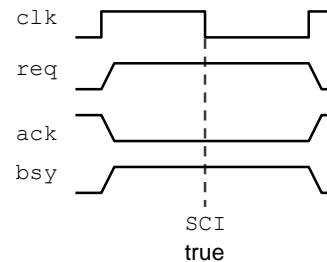


IEEE 1800-2012 16.12.6

- Conditional properties are defined with implication operators:
  - If `exprA` is true, then `exprB` must occur.
  - If `exprA` is not true, `exprB` is not checked.
- For the same cycle implication, `exprB` must be true in the same cycle as `exprA`.
- For implication properties:
  - `exprA` is the antecedent or enabling condition.
  - `exprB` is the consequent or fulfilling condition.
  - Both can be either sequences or Boolean conditions.

```
exprA |-> exprB;
```

```
property SCI;
 @ (negedge clk)
 (req && !ack) |-> bsy;
endproperty
```



341 © Cadence Design Systems, Inc. All rights reserved.

Conditional properties are those where some initial (or enabling) conditions must be true before a series of (fulfilling) conditions must be checked. Conditional properties are defined using implication operators.

SystemVerilog provides two sequence implication operators. This is the “same cycle” implication operator. If the condition on the left of the operator is true, then the condition on the right-hand side of the operator must be true in the same cycle.

The left side is called the antecedent or enabling condition.

The right side is called the consequent or fulfilling condition.

In this example, if `req` is high and `ack` is low at the failing edge of `clk`, then `bsy` must be high in the same cycle. If `req` is low or `ack` is high, then the enabling condition is not true, and `bsy` is not checked. As both sides of the operator are single Boolean conditions, the property can be rewritten as an unconditional property:

```
@ (negedge clk) (req && !ack && bsy) | !req | ack;
```

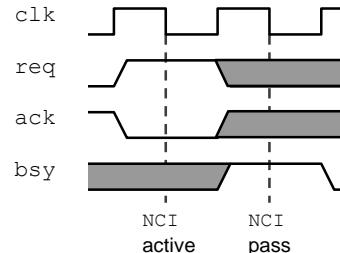
Either of the enabling or fulfilling conditions can be sequences. If an enabling sequence runs to completion, then the first cycle of the fulfilling sequence (or condition) is checked in the same cycle as the enabling sequence completes.

## Implication ( $|=>$ : Next Cycle)

- For the next cycle implication, `exprb` must be true in the cycle after `expra` completes.
- Otherwise, same rules apply:
  - If `expra` is true, then `exprb` must occur.
  - If `expra` is not true, `exprb` is not checked.
  - Both can be either sequences or Boolean conditions.
- If a variable is not included in a condition, its value is “don’t care”:
  - `bsy` is not checked in enabling condition.
  - `req` and `ack` are not checked in the fulfilling condition.

```
expra |=> exprb;
```

```
property NCI;
 @ (negedge clk)
 (req && !ack) |=> bsy;
endproperty
```



342 © Cadence Design Systems, Inc. All rights reserved.



This is the “next cycle” implication operator. If the condition on the left of the operator is true, then the condition on the right-hand side of the operator must be true from the next evaluation cycle.

The left-hand side is called the antecedent or enabling condition.

The right-hand side is called the consequent or fulfilling condition.

At each evaluation point (falling edge of `clk` in these examples), the enabling condition is checked (`req = 1; ack = 0`). If the enabling condition is false, the assertion is ignored for this clock cycle. If the enabling condition is true, then the fulfilling condition is checked at the next evaluation point (`bsy = 1` at next falling edge of `clk`). If the fulfilling condition is true, then the assertion passes. If the fulfilling condition is false, the assertion fails.

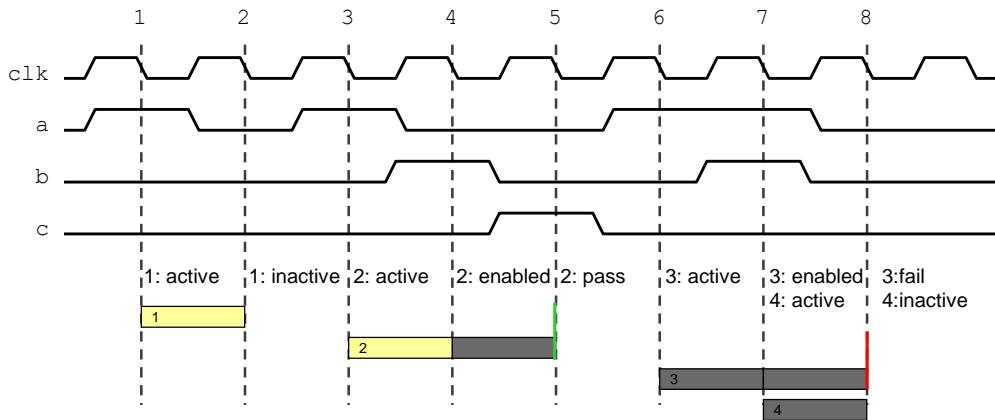
Either of the enabling or fulfilling conditions can be sequences. If an enabling sequence runs to completion, then the first cycle of the fulfilling sequence (or condition) is checked in the next cycle after the enabling sequence completes.

If a variable is not included in the enabling condition or fulfilling condition, then its value is “don’t care”. You must be careful to write design properties which check all parts of the required behavior.

For example, should `bsy` be low in the enabling condition? Should `req` and `ack` stay unchanged in the fulfilling condition?

## Analyzing Sequence Property with Implication

```
property STATE;
 @ (negedge clk) (a ##1 b) |=> c;
endproperty
```



343 © Cadence Design Systems, Inc. All rights reserved.



Remember that the enabling sequence on the left side of the implication operator must complete before the fulfilling sequence on the right side is checked.

Here is an analysis of the evaluation property STATE:

Cycle 1: a is true, so attempt 1 of STATE evaluation becomes active.

Cycle 2: b is false, so attempt 1 of STATE evaluation becomes inactive.

Cycle 3: a is true, so attempt 2 of STATE evaluation becomes active.

Cycle 4: b is true, so attempt 2 of STATE evaluation becomes enabled.

Cycle 5: c is true, so attempt 2 of STATE evaluation succeeds and finishes.

Cycle 6: a is true, so attempt 3 of STATE evaluation becomes active.

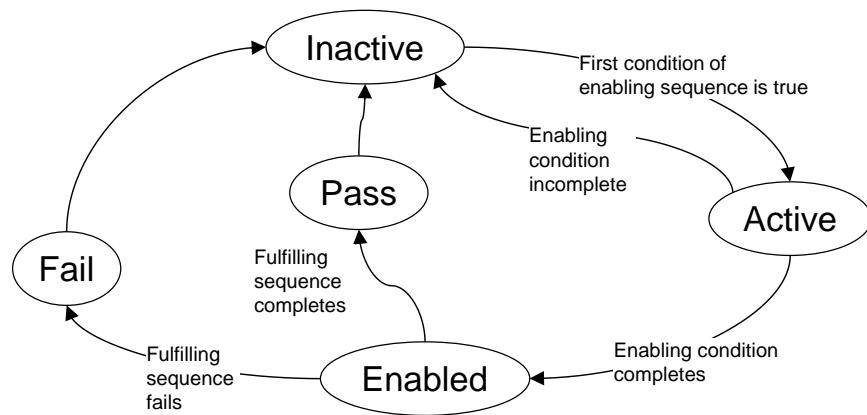
Cycle 7: b is true, so attempt 3 of STATE evaluation becomes enabled. a is still true, so attempt 4 of STATE evaluation becomes active.

Cycle 8: c is false, so attempt 3 of STATE evaluation fails. b is false, and attempt 4 of STATE evaluation becomes inactive.

The key point here is that assertions can overlap. The first condition of the enabling sequence is checked at every evaluation point. If the first condition is true, another copy of the property becomes active, regardless of how many other copies of the property are currently active. There can be any number of copies of a property active at any one time, all at different stages in their evaluation.

## Pictorial View of Assertion Property Evaluation States

An assertion can be in one of several states:



```
@(clocking) disable iff (dis_expr) enable |=> fulfill;
```

344 © Cadence Design Systems, Inc. All rights reserved.



Assertions start in the *inactive* state. At each evaluation cycle, the enabling condition (or the first condition of an enabling sequence) is checked. If the condition is true, a copy of the assertion becomes *active*. In the *active* state, the assertion continues checking the enabling condition or sequence. If the enabling condition(s) do not complete, then the assertion becomes *inactive*. If the enabling condition(s) complete, then the assertion is *enabled*. In the *enabled* state, the assertion checks the fulfilling condition(s). If the fulfilling conditions fail to complete, then in the evaluation cycle the failure was detected, the state of the assertion is *fail*, after which it becomes *inactive*. If the fulfilling condition(s) are true, then the assertion state is *pass* in the evaluation cycle the final fulfilling condition becomes *true*.

A disable can terminate an *active* or an *enabled* assertion. For SystemVerilog2005, a disabled assertion counted as a pass. For formal verification, a disabled assertion is interpreted as a pass. From SystemVerilog2009, a disabled assertion has a new separate disabled state before returning to inactive.

Remember the enabling condition of every assertion is checked on every evaluation cycle, therefore multiple versions of a single assertion can exist in different states. This can make management and debugging of assertions difficult.

## Disabling Properties with disable iff



IEEE 1800-2012 16.15

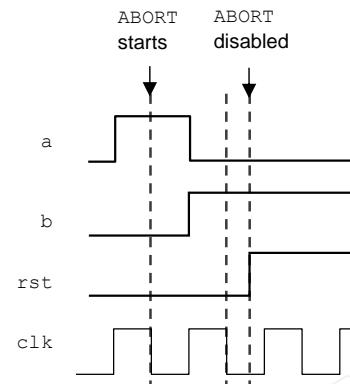
- To terminate an assertion when a condition is true, use disable iff.
- If expr is true at any time, then the property is terminated.
- Disable expression must be bracketed () .
- Useful for cancelling multi-cycle sequence properties.
  - For example, burst on a bus interrupted by reset.
- A default disable can be specified.

```
property NAME;
 @(clocking) disable iff (expr)
 the_property;
endproperty
```

Syntax

```
property ABORT;
 @(negedge clk) disable iff (rst)
 a |=> b ##1 c ##1 d;
endproperty
```

Example



345 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

Disable is essential for terminating long sequence properties if design conditions change, e.g., in this case a reset occurs.

If the disable condition (`rst`) is true, then the assertion is disabled (terminated) immediately, regardless of how the property is clocked.

You can view the disable as being asynchronous from the property sampling clock.

Up to SystemVerilog2009, disabled properties were defined as a pass, just as if the property had successfully completed.. This could possibly lead to misleading results when a simulator reported the number of design properties which have passed or failed. This is why most simulators reported the number of assertions which have completed or finished, rather than the number which have passed.

From SystemVerilog2009, a new completion state for properties was added, in addition to pass or fail. When the disable expression of a property is true, the property is counted as “disabled”, and simulators can now report the number of assertions that are disabled.

In Formal Verification (where assertion languages were first used), disabled assertions are always counted as passes because the absence of a failure is deemed as a pass.

Default disable iff can be defined for properties which do not have an explicit disable iff in their declaration.

The syntax is

default disable <Boolean>, for example:

**disable iff (!rst\_n)**

The default disable applies only to the scope in which it is declared.

## Cycle Delay Repetition



IEEE 1800-2012 16.7

- ## can be used with any constant integral expression to define a number of cycle delays.
- After expr1, count N evaluation points, after which expr2 must be true.
- ##0 is allowed (and useful) for sequences:
  - Zero delay ("fusion")

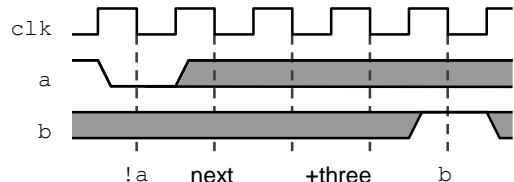
```
expr1 ##N expr2
```



Use with care!

If a is low,  
then b is high  
4 cycles later

```
property A3B_NEXT;
 @ (negedge clk) !a |=> ##3 b;
endproperty
```



Cycle delay ## just counts evaluation cycles without checking any conditions. The cycle delay expression can be an integral constant expression or a range between two integral constant expressions. Here we have a 3-cycle delay. The property succeeds if, after a is low, b is low 4 cycles later. Note that if the cycle delay comes immediately after the next cycle implication operator, then the cycle count starts after the cycle when the enabling condition completes. If same cycle implication is used, then the count starts on the same cycle as the enabling condition completes.

A delay of zero is valid and useful. A zero delay between the end of the left sequence and the start of the right sequence means that the left sequence ends and the right sequence begins in the same cycle. This is called sequence fusion.

### Example

$$(a \ ##1 \ b) \ ##0 \ (c \ ##1 \ d) = (a \ ##1 \ (b \ \&& \ c) \ ##1 \ d)$$

Cycle delay repetition should be used with care. You are not checking any values during the cycle count, and you ask yourself whether you should be checking something. In the example above, should a be high or low during the 3-cycle delay? Can b only be low on the fourth cycle after a, or can it be low on every cycle?

## Consecutive Sequence Repetition

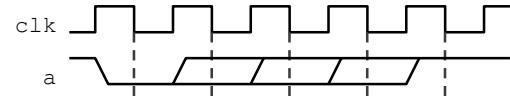


IEEE 1800-2012 16.9.2

You can specify multiple consecutive repetitions of a sequence using `[*N]`:

- Consecutive repetition operator.
- `expr` repeats consecutively `N` times.
- `N` must be an non-negative integer constant expression.

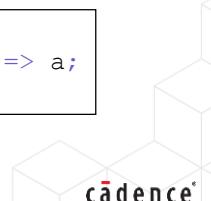
expr [\*N]

**Instead of this...**

```
property SEQ_COUNT;
 @ (negedge clk) !a ##1 !a ##1 !a ##1 !a |=> a;
endproperty
```

**Write this...**

```
property CONSEC;
 @ (negedge clk) !a[*4] |=> a;
endproperty
```



SystemVerilog provides three sequence repetition operators. The consecutive repetition operator simply repeats an expression for a set number of times given by `N`. The consecutive repetition operator can be used with a Boolean expression, a named sequence, or an inline sequence enclosed within parentheses (see below).

Consecutive repetition allows an individual condition or a complete sequence to repeat, consecutively, a set number of times.

### Example

`(a [*4])` is equivalent to `(a ##1 a ##1 a ##1 a)`

`((a ##1 b) [*2])` is equivalent to `(a ##1 b ##1 a ##1 b)`

The count value `N` must be statically computable, that is, known at compilation time and not defined by a variable or expression.

## Consecutive Repetition with Ranges



IEEE 1800-2012 16.9.2

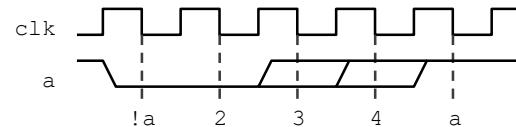
expr [\*min:max]

- You can specify a range for consecutive repetition.
- expr repeats consecutively for:
  - At least min number of times.
  - At most max number of times.
- Range limits must be non-negative integer constant expressions:
  - min can be 0.
    - That is, no repetitions.
  - max can be \$ (unlimited).
    - That is, can repeat forever.
  - min is less than or equal to max.



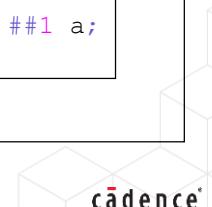
\$ bound can lead to ambiguities

a goes low in between 2 and 4 cycles only



```
property RANGE;
 @(negedge CLK)
 a #1 !a |=> !a[*1:3] #1 a;
endproperty
```

348 © Cadence Design Systems, Inc. All rights reserved.



The consecutive repetition operator can have a range. The range specifies a minimum number of repetitions which must occur and a maximum number which must not be exceeded.

Both bounds of the range must be statically computable (i.e., a constant expression) and must be defined in an ascending direction.

In the example, once the enabling condition of a falling edge on a is found, the property will check that a is false for two cycles (lowest bound of range). We use a same cycle implication operator to begin the count of a low in the same cycle as a low for the enabling condition completion. Then for the next 3 evaluation points, if a is true, the property completes and passes. Otherwise, the property remains active. On the final (fourth) cycle, a must be true (upper bound of range exceeded) or the property fails.

This property specifically looks for a falling edge on a, as we interpret the “a goes low” of the specification to imply this. The property would not catch the case of a starting low at the beginning of simulation, and then remaining low for more than 4 cycles.

A minimum bound of 0 means that the sequence or condition can be missing. For example:

```
a ##1 b*[0:2] ##1 c
```

matches

```
a ##1 c
a ##1 b ##1 c
a ##1 b ##1 b ##1 c
```



## Additional SVA Features

SystemVerilog has many more advanced properties, sequence operators and features:

1. Conditional property operators
2. Nonconsecutive sequence repetition
3. Concurrent, alternate and overlapping sequences

### Advanced SVA features:

- Detecting the end of a sequence
- Parameterized sequences and properties
- Action blocks on property pass or failure
- Functional coverage

The SystemVerilog Assertions class covers all the above including the following:

- Coding guidelines and recommendations
- Practical examples
- Formal Friendly SVA coding
- Use of Verilog helper code to simplify verification using SVA properties



SystemVerilog has many more advanced property and sequence operators and features. The *SystemVerilog Assertions* training covers these plus coding guidelines and recommendations, functional coverage and working examples.

## Module Summary

- SystemVerilog assertions are based on Verilog Boolean conditions
- Constructing a simple Boolean assertion is straightforward:
  - Write a Boolean condition using Verilog syntax
  - Define when the condition is checked with a clock expression
  - Name the condition using the `property` statement
  - Assert the property using an `assert` statement
- With sequences you can easily express complex, multi-cycle properties:
  - Boolean expressions separated by cycle delay `##N`
- SVA has many features for defining multi-cycle and repeated sequences:
  - For example: consecutive repetition `[ *N ]`

350 © Cadence Design Systems, Inc. All rights reserved.



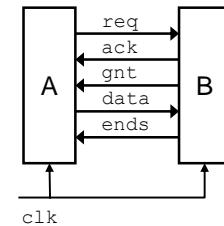
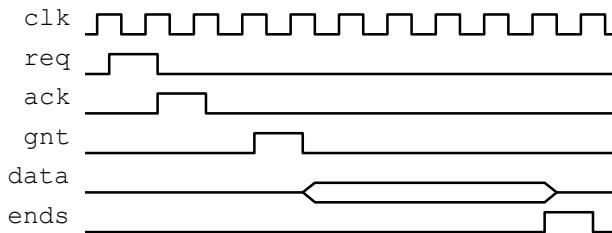
Pause here for a moment and review what you have learned about SystemVerilog assertions.

## Quiz

Code this property and assert it.

If the req-ack handshake occurs then:

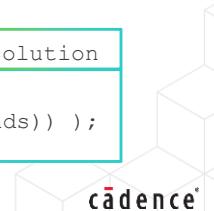
- gnt is true on only the 2nd cycle after the req-ack handshake.
- ends is true on only the 6th cycle after gnt is true.



Solution

```
A1 : assert property (@(posedge clk)
 (req #1 ack) |=> (!gnt #1 gnt ##1 (!gnt && !ends)[*5] ##1 (!gnt && ends)));
```

351 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*

## Labs



### Lab 18 Simulating Simple Implication Assertions

- Create simple implication assertions, track assertion failures in simulation and debug the assertions.

### Lab 19 Sequence-Based Properties

- Create sequence-based assertions, track multiple, overlapping assertions in simulation and debug issues.



*This page does not contain notes.*



## Direct Programming Interface (DPI)

**Module** **22**

Revision **1.0**

Version **21.10**

Estimated time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

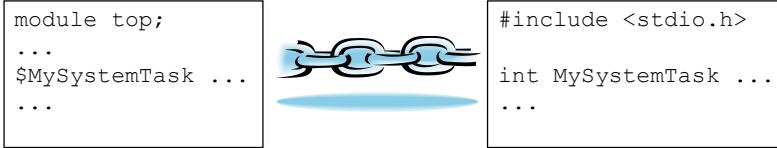
- Understand the concepts of Direct Programming Interface (DPI)
  - Data types
  - Imported tasks and functions
  - Pure and context definitions
  - Exported tasks and functions
  - Compilation
- Use the DPI to import C functions into your SystemVerilog code



*This page does not contain notes.*



## Quick Reference Guide: The VPI



The current interface of VPI is the traditional interface between Verilog and C code.

| Advantages                           | Disadvantages                                    |
|--------------------------------------|--------------------------------------------------|
| Powerful                             | Difficult, even for simple operations            |
| Safe handling of type conversion     | Linking is different for each simulator          |
| Full visibility of design hierarchy  | Multiple applications across multiple simulators |
| Synchronization to simulation events | Create tool management problems                  |

The current interface of VPI is a true application programming interface. Through it you can access and navigate the full design hierarchy, and can synchronize a C application to any point of simulation time or to any simulation event. With this power comes difficulty of use, which typically limits its use to a handful of experts in any company.

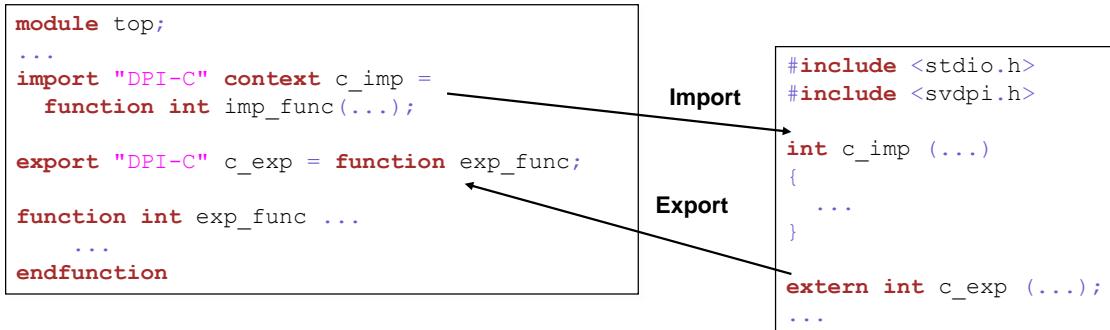
## What Is SystemVerilog DPI?



DPI is an interface between SystemVerilog and a foreign programming language. It consists of two separate layers: the SystemVerilog layer and a foreign language layer. Both sides of DPI are fully isolated.

DPI Maps between SystemVerilog subroutines and external routines in following two ways:

1. **Import to SV:** SystemVerilog calls subroutines defined in foreign language layer.
2. **Export from SV:** SystemVerilog defines subroutines called in foreign language layer.



356 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

The direct programming interface is a lightweight interface compared to the PLI, designed to interface between SystemVerilog and any foreign programming language. It allows SystemVerilog subroutine calls to be mapped to foreign language implementations, and SystemVerilog subroutine declarations to be mapped to foreign language calls.

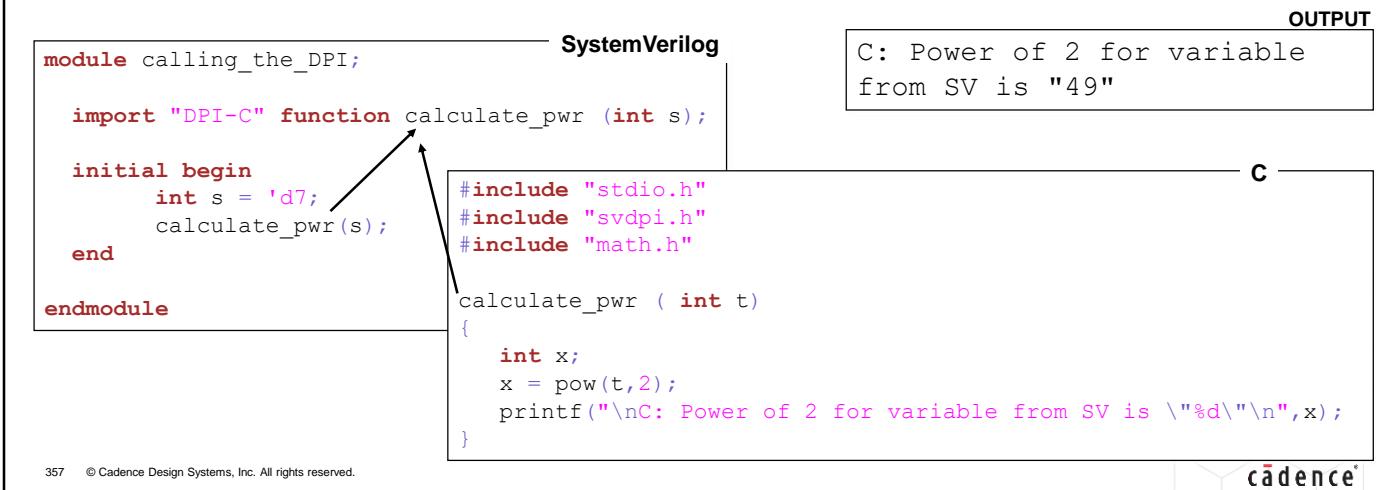
The standard clearly separates the specification of the SystemVerilog layer from the foreign language layer so that, in theory, alternate foreign languages can be swapped in without changing the SystemVerilog view of the interface.

So, any language which follows the normal C linkage conventions can be used to link with SystemVerilog using DPI.

# 1. Import to SV: Importing Tasks and Functions

The import prototype declares how SystemVerilog views the C subroutine.

- Argument and return types must be compatible with that in the C layer.
- You call the imported subroutine exactly as if it was in SystemVerilog.



An import declaration imports a C function as a SystemVerilog task or function. The import declaration maps the C function name to a SystemVerilog task or function name. After you import a C function, you call it as if it were a SystemVerilog task or function.

-----

```

import "DPI-C" [context | pure] [c_identifier =] function function_data_type
function_identifier ([argument_list]);

```

```

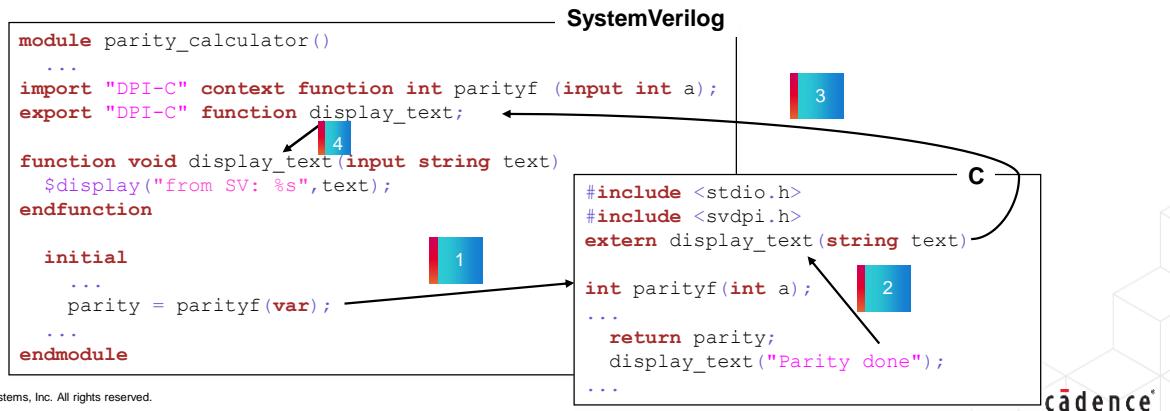
import "DPI-C" [context] [c_identifier =] task task_identifier
([argument_list]);

```

## 2. Export from SV: Exporting Tasks and Functions

The export declaration makes a SystemVerilog subroutine visible in the C global name space.

- Argument and return types must be compatible with those in the C layer.
- You call the exported subroutine exactly as if it was in C.
- An imported subroutine *must* be declared as context if it contains a call back into the SystemVerilog layer through a call to an exported subroutine.



```
export "DPI-C" [c_identifier =] function function_identifier;
export "DPI-C" [c_identifier =] task task_identifier;
```

An export declaration exports a SystemVerilog subroutine as a C function. The export declaration maps the SystemVerilog task or function name to a C function name.

In the C layer, you declare the function with `extern` and call it as if it were a normal C function.

As with imported subroutines, the exported SystemVerilog subroutine and C `extern` declaration must match, specifically for the following:

- Name (although this can be mapped with a linkage name – see next slide)
- Return type
- Number of arguments
- Type of arguments
- Direction of arguments

## SystemVerilog DPI Characteristics

The SystemVerilog and foreign language layers are independent.

- Each side is unaware of the implementation of the other side.
- Each layer does its own data interpretation and conversion.
- Each layer follows its own syntax and semantic rules:
  - For instance, for declaring and calling subroutines.

Only SystemVerilog data types can be transferred across the interface.

- 2-state or 4-state data types can be handled.
- Simple types have a direct mapping in both layers:
  - Complex types may require custom conversion.



These are the key concepts of DPI:

- The two layers are independent of each other. You declare subroutines and make subroutine calls in each layer as if the other layer does not exist. You follow the syntax and semantics of the host language for all declarations and statements and data type management.
- This requires that the data types passed between the layers be restricted to the subset that map between the layers. This in particular excludes dynamic types such as classes and class-like types such as covergroups and virtual interfaces, dynamic and associative arrays, and queues, events, mailboxes and semaphores.

## Mapping Data Types



IEEE 1800-2012 7.1.0

- The mapping between the basic SystemVerilog data types and the corresponding C types is defined as shown in the table.
- The DPI also supports the SystemVerilog and C unsigned integer data types that correspond to the mappings.
  - The table shows their signed equivalents.

requires  
svdpi.h

No long type

| SystemVerilog | C             |
|---------------|---------------|
| byte          | char          |
| shortint      | short int     |
| int           | int           |
| longint       | long          |
| real          | double        |
| shortreal     | float         |
| string        | const char *  |
| chandle       | void *        |
| bit †         | unsigned char |
| logic †       | unsigned char |

360 © Cadence Design Systems, Inc. All rights reserved.



The mapping between SystemVerilog and foreign language data types is specified by the foreign layer. Here we present the mapping specific to the C foreign language.

The mapping of simple SystemVerilog data types such as byte, int, real and string is straightforward, as they have obvious C counterparts. The logic type maps to an unsigned char on the C side to accommodate the four states. For consistency, the bit type also maps to an unsigned char. Your C code should not use the character value directly, but should instead include svdpi.h and use the text macros representing the four states. The SystemVerilog LRM describes the name and core functionality of svdpi.h and permits vendors to differ in their implementations.

```

typedef signed __int8 int8_t;
typedef uint8_t svScalar;
typedef svScalar svBit;
typedef svScalar svLogic;
#define sv_0 0
#define sv_1 1
#define sv_z 2
#define sv_x 3
```

**void\* - 1800\_2010-H.7.4**

## Mapping Data Types (continued)

- C include file `svdpi.h` defines C types for simple SystemVerilog data types.

| <b>svdpi.h</b> | <b>Scalar Types</b> |         | <b>Packed Array Types</b> |               |
|----------------|---------------------|---------|---------------------------|---------------|
| <b>SV</b>      | bit                 | logic   | bit[n:0]                  | logic [n:0]   |
| <b>C</b>       | svBit               | svLogic | svBitVecVal               | svLogicVecVal |

- SystemVerilog enumerated types map to their base type (default `int`).
  - Enumerated names are not available in the C layer.
- Unpacked types having no packed elements map to corresponding C types:
  - Unpacked arrays and unpacked structures of simple types.

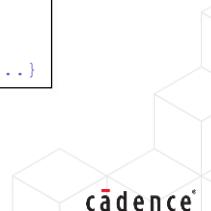
```
module top;
 import "DPI-C" function int c_func (input bit bin1);
 ...

```



```
#include <stdio.h>
#include <svdpi.h>

int c_func (svBit b1) { ... }
```



SystemVerilog enumerated types are accessible on the C side as the enumeration base type, but the enumeration value names are not available.

SystemVerilog packed types are accessible on the C side as arrays of the `svBitVecVal` type or `svLogicVecVal` type as described in `svdpi.h`.

SystemVerilog unpacked arrays, structures and unions having only the simple types map to their corresponding C arrays, structures and unions of their corresponding C types.

`svBitVecVal` and `svLogicVecVal` are arrays of 32-bit elements.

```

```

```
typedef unsigned __int32 uint32_t;
typedef uint32_t svBitVecVal;
typedef struct vpi_vecval {
 uint32_t a;
 uint32_t b;
} s_vpi_vecval, *p_vpi_vecval;
typedef s_vpi_vecval svLogicVecVal;
```

## Example 1: Passing a String to C

```
module pass_string;
 import "DPI-C" function void print_message (string s);
 initial
 begin
 string s = "Hello DPI";
 print_message(s);
 end
endmodule
```

### OUTPUT

C: Message got from SV is  
"Hello DPI"

```
#include "svdpi.h"

print_message(char *t)
{
 printf("\nC: Message got from SV is \"%s\"\n",t);
```



*This page does not contain notes.*

## Example 2: Passing a String from C to SV

```
module pass_string;
 import "DPI-C" function string print_message ();
 initial begin
 string s = print_message ();
 $display("%s", s);
 end
endmodule
```

### OUTPUT

```
HI from C World
```

```
#include "svdpi.h"

char * print_message ()
{
 return "HI from C World";
}
```

363 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*

## Example 3: Passing 1D Unpacked Bit Array to C

```
module pass_array;
 import "DPI-C" function void print_array (input bit[15:0] arr[5:0]);
 initial begin
 bit[15:0] arr[5:0] = '{1,2,3,4,5,6};
 print_array(arr);
 end
endmodule
```

```
#include "svdpi.h"

void print_array(svBitVecVal arr[6])
{
 for(int j=0; j<6; j++)
 {
 printf("\nC: Value in C-code of nums[%d] is %d\n",j,arr[j]);
 }
}
```

### OUTPUT

```
C: Value in C-code of nums[0] is 1
C: Value in C-code of nums[1] is 2
C: Value in C-code of nums[2] is 3
C: Value in C-code of nums[3] is 4
C: Value in C-code of nums[4] is 5
C: Value in C-code of nums[5] is 6
```

364 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*

## Example 4: Passing Logic Value to C

```

module pass_logic_value;
 import "DPI-C" function void print_statement (input logic[7:0] v);

 initial begin
 print_statement (8'b0011XXZZ);
 end
endmodule

#include "svdpi.h"

void print_statement(svLogicVecVal* v)
{
 io_printf("v= %x\n", *v);
 io_printf("aval = %x\n", v->a);
 io_printf("bval = %x\n", v->b);
}

```

Canonical Representation

| Actual Value | 0 | 0 | 1 | 1 | X | X | Z | Z |
|--------------|---|---|---|---|---|---|---|---|
| a value      | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| b value      | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

OUTPUT

v= 3c  
aval= 3c  
bval= f

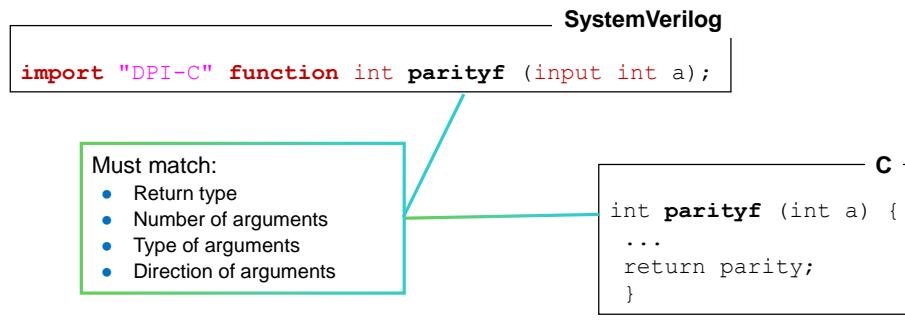


365 © Cadence Design Systems, Inc. All rights reserved.

*This page does not contain notes.*

## Mapping Data Types Compatibility Issues

- The compilers (C and SystemVerilog) cannot communicate.
- The linker is aware of symbols, but not how they are interpreted.
- YOU assume responsibility for argument direction and type compatibility!
  - Declaration mismatches can lead to elaboration errors.
  - Type incompatibility can lead to unpredictable behavior.



366 © Cadence Design Systems, Inc. All rights reserved.



As the SystemVerilog and C layers are compiled and elaborated or linked separately, there is no opportunity for the DPI to check function declarations or verify data type compatibility. It is therefore critical that you match the import declaration to the C function with respect to the function name, return type and argument types.

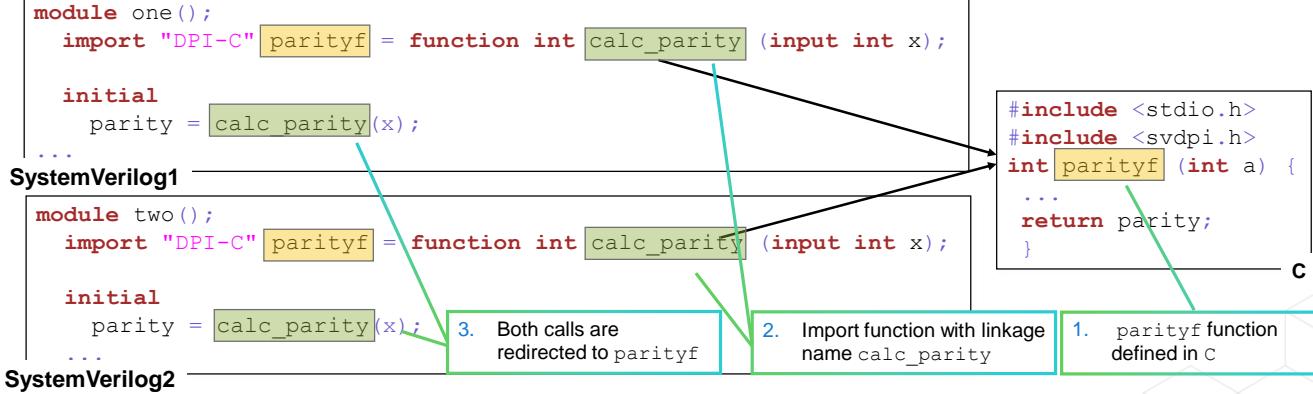
A name mismatch will probably result in only a “symbol not found” error. A type mismatch error can be more serious and can lead to unpredictable effects as the C layer misinterprets the SystemVerilog arguments.

## Import Linkage Name: How to Import a Function with a Different Name



You can specify a different SystemVerilog name for the C function, if you provide the C linkage name in the import declaration.

- The SystemVerilog name and C function name are by default identical.
  - You can optionally specify a different SystemVerilog name.
- You can import a C function multiple times if the SystemVerilog names do not clash.
  - Use different names or different scopes.



367 © Cadence Design Systems, Inc. All rights reserved.



```
import "DPI-C" [property] [c_identifier =] prototype;
```

By default, you call an imported subroutine using the same name as the C function. You can specify a different SystemVerilog name for the C function if you provide the C linkage name in the import declaration. This is useful when the C function name is a reserved word or illegal identifier in the SystemVerilog namespace. It also allows you to import a C function multiple times with different SystemVerilog names.

The C linkage name must of course be a legal C identifier, and if not a legal SystemVerilog identifier, must be appropriately escaped.

Verilog supports the use of escaped identifiers, which allow any printable ASCII character to be used in an identifier. Escaped identifiers must begin with a backslash (\) and end with a whitespace.

### Example

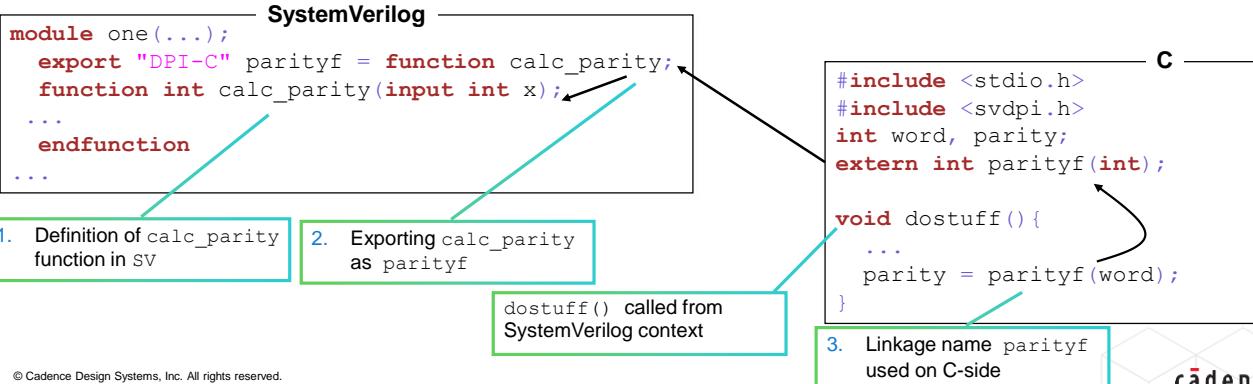
\unit@32 \unit-32 \16-bit-bus are all legal escaped identifiers.

## Export Linkage Name: How to Export a Function with a Different Name



You can specify a different SystemVerilog name for the C function, if you provide the C linkage name in the export declaration.

- The SystemVerilog subroutine name and C function name are by default identical.
- You can optionally specify a different SystemVerilog name.
- You can export a SystemVerilog task or function only once.



368 © Cadence Design Systems, Inc. All rights reserved.

```
export "DPI-C" [c_identifier =] {function | task} name;
```

By default, you call an exported subroutine using the same name as the SystemVerilog subroutine. You can specify a different SystemVerilog name for the C function if you provide the C linkage name in the export declaration. This is useful when the SystemVerilog subroutine name is a reserved word or an illegal identifier in the C namespace.

The C linkage name must of course be a legal C identifier, and if not a legal SystemVerilog identifier, must be appropriately escaped.

Verilog supports the use of escaped identifiers, which allow any printable ASCII character to be used in an identifier. Escaped identifiers must begin with a backslash (\) and end with a whitespace.

### Example

\unit@32 \unit-32 \16-bit-bus are all legal escaped identifiers.

## Example: Imported and Exported Functions

```

module top;
bit b1 = 1'b1, b2 = 1'b0;

import "DPI-C" context c_func =
 function void import_c_func (input bit bin1, inout bit bin2);

export "DPI-C" sv_func = function export_sv_func;
function void export_sv_func(input bit eb1,
 inout bit eb2);
 if (eb1 != eb2) begin
 eb2 = ~eb2;
 endfunction

initial begin
 import_c_func(b1, b2);
end

```

```

#include <stdio.h>
#include "svdpi.h"

void c_func (svBit b1, svBit *b2) {
 int res;
 printf(" b1:%d, b2:%d\n", b1, *b2);
 sv_func(b1, b2);
 printf(" b1:%d, b2:%d\n", b1, *b2);
}

extern void sv_func(svBit, svBit*);

```

**OUTPUT**

```
xcelium> run
b1:1, b2:0
b1:1, b2:1
```

369 © Cadence Design Systems, Inc. All rights reserved.



The SystemVerilog initial block calls the `import_c_func`, which is linked to the C function named `c_func` by the `import` declaration. This function is imported with the `context` property so that it can in turn call exported SystemVerilog subroutines.

The C implementation of the `c_func` function calls the `sv_func` function that it has previously declared to be `extern`. This function is linked to the SystemVerilog function named `export_sv_func` in the `export` declaration.

## How to Include Linked C-Object Code During Compilation



To integrate the foreign language code into a SystemVerilog application, compiled object code is required for cases where the compilation and linking of source code are fully handled by the user.

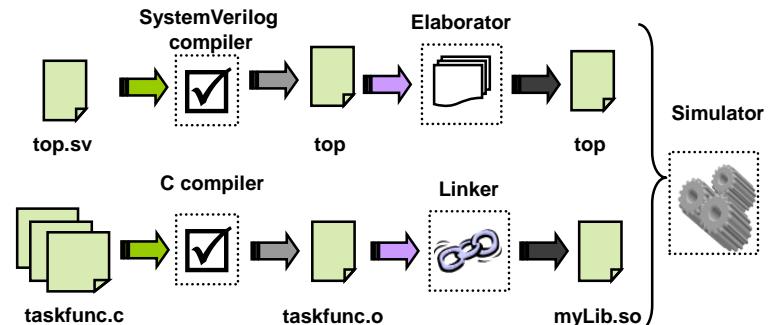
SystemVerilog recommends two alternatives for including the linked C object code:

1. By directly providing the library name with the `-sv_lib` option.

Example 01

```
<sim> mvDesign -sv_lib myLib
```

2. By entry in a bootstrap file whose name is provided with `-sv_liblist`.
  - Optionally providing a path prefix with `-sv_root`.



Alternatively, the Cadence® simulator can compile and link C code automatically.

The actual usage methodology is vendor dependent, but is likely to be something like this:

- You separately compile your C code and link it into a shared object library with a platform-specific extension, such as `.sl` or `.so` for UNIX variants and `.dll` for Windows.
- You compile, elaborate and simulate your SystemVerilog code as usual, and provide the shared-object library to the simulator.

The standard recommends that command-line options be provided for this purpose, and even suggests names for them:

- The `-sv_lib` option provides a pathname for the shared object library minus its extension – the tool provides the correct extension for the platform.
- The `-sv_root` option provides a single directory path that is prefixed to any relative path specified by the `-sv_lib` option.

A nonstandard, but widely supported technique is to compile your C code into a shared object library with the name `libdipi` (e.g., `libdipi.so`). A file with this name is automatically linked into the simulator.

For a simple example, the Cadence simulator can compile and link the C code directly, for example:

```
xrun taskfunc.c top.sv
```

## Module Summary

In Summary, SystemVerilog offers DPI to communicate easily with foreign language such as C. The table below summarizes its advantages and potential issues.

| Advantages                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | Potential Issues                                                                                                                                                                                                                                                                                                                                                                                                                     |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Easy to use:</p> <ul style="list-style-type: none"> <li>• Standard interface: <ul style="list-style-type: none"> <li>▪ If no packed types</li> </ul> </li> <li>• Doesn't require new simulator executable</li> <li>• Look and feel of standard SystemVerilog subroutines</li> <li>• Easy access to standard C functions</li> <li>• Bidirectional interface: <ul style="list-style-type: none"> <li>▪ SystemVerilog can call C</li> <li>▪ C can call SystemVerilog</li> </ul> </li> </ul> | <ul style="list-style-type: none"> <li>• No arbitrary data types: <ul style="list-style-type: none"> <li>▪ SystemVerilog types only</li> <li>▪ Packed array and struct layout is dependent upon platform and implementation</li> </ul> </li> <li>• Does not replace PLI: <ul style="list-style-type: none"> <li>▪ No access to simulation data structure</li> <li>▪ No synchronization with simulation events</li> </ul> </li> </ul> |

371 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*

## Quiz

 Write a function import statement to import the `exp2f` function (from the `cmath` library) with the `base2exp` SystemVerilog identifier. Assume the C function has no side effects. The C function prototype is:

```
float exp2f(float x);
```

```
import "DPI-C" pure exp2f = function shortreal base2exp (input shortreal x) ;
```

 True or False? The standard supports import and export of class methods.

False: The standard does not support import or export of class methods.

 How do you make an imported task (implemented in C) consume time?

The imported task calls an exported task (implemented in SystemVerilog) that consumes time.

## Lab



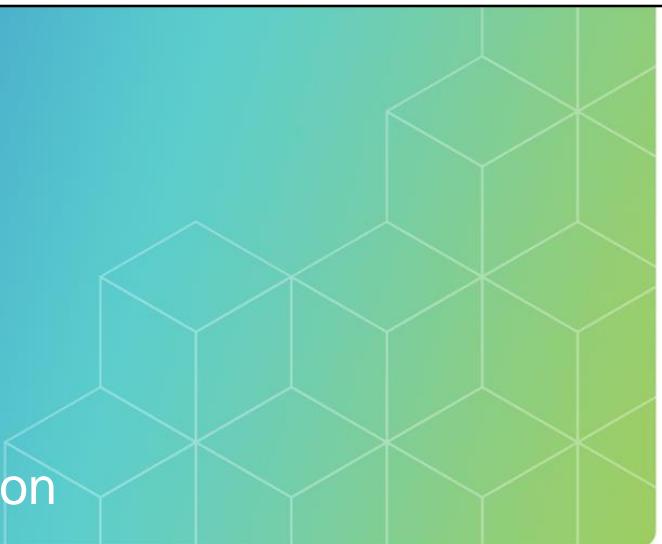
### Lab 20 Simple DPI Use

- Import simple C functions from standard and maths C libraries.

373 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*



## Interprocess Synchronization

**Module** **23**

Revision **1.0**

Version **21.10**

Estimated time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

- Apply SystemVerilog interprocess synchronization mechanisms
  - Nonblocking event trigger
  - Event sequences
  - Mailboxes
  - Semaphores



*This page does not contain notes.*



## Review Blocking/Named Event Trigger ( -> )

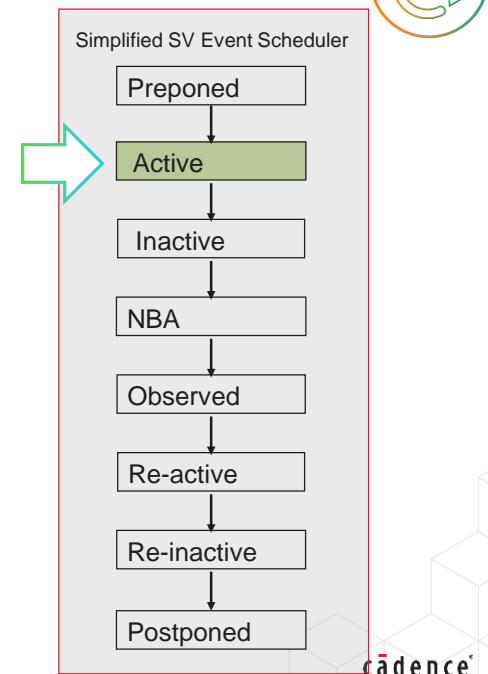
- Traditional Verilog event trigger is blocking:
  - Triggered instantaneously in Active region when executed.
- Synchronization can be difficult:
  - Processes to be triggered must be waiting for the event when it occurs.

```
event e;
integer i = 0;

always @e $display("i:%0d", i);

initial begin
 i <= 1;
 -> e; // blocking i:0
 ...

```



376 © Cadence Design Systems, Inc. All rights reserved.

Traditional Verilog event triggered are blocking. The event is triggered instantaneously in the Active region of the event scheduler. This can make synchronization difficult. A process to be triggered must be waiting for the event at the instant in the Active region when the event trigger is executed. If the triggered process needs to consume time, then you have race conditions between the event trigger and process event control.

Another issue with traditional events is demonstrated by the example. As the event is triggered in the Active region, any nonblocking behavior of the design cannot be captured by a process triggered by the event.

## Nonblocking Event Trigger ( ->> )



IEEE 1800-2012 15.5.1

- You can trigger SystemVerilog events with a nonblocking ->> operator:
  - Executes without blocking
  - Schedules an event for the NBA region
  - Helps prevent races between processes
  - Can include optional embedded event control

```

event e;
integer i = 0;

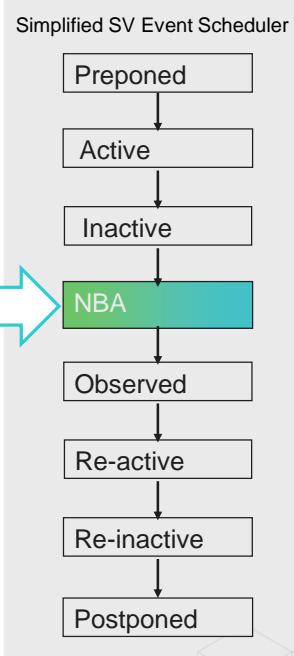
always @e $display("i:%0d",i);

initial begin
 i <= 1;
->> e; // nonblocking i:1

->> #3 e;
->> @ (posedge clk) e;
...

```

377 © Cadence Design Systems, Inc. All rights reserved.



cadence®

SystemVerilog adds a nonblocking event trigger (->>) that schedules the event for the NBA region. You can also use intra-assignment delays just as you do with the nonblocking assignment operator.

The nonblocking event trigger prolongs the visibility of the event and allows nonblocking behavior to be captured by processes triggered by the event. It also allows us to schedule an event for some time or event in the future.

The nonblocking event trigger serves the same general purpose as the nonblocking assignment – to prevent a race between a block that triggers the event and a block that waits for the event in the same delta cycle.

## Persistent Event Trigger ( triggered )



IEEE 1800-2012 15.5.3

- A process can wait for the `triggered` property of the event:
  - The `triggered` state persists to the end of the time step.

```
event e1, e2; "Hangs"
initial begin
 $display("fork");
 fork
 @e1;
 -> e1;
 -> e2;
 @e2;
 join
 $display("join");
...

```

output `fork`

```
event e1, e2; "Completes"
initial begin
 $display("fork");
 fork
 @e1;
 ->> e1;
 -> e2;
 wait(e2.triggered);
 join
 $display("join");
...

```

output `fork`  
`join`

378 © Cadence Design Systems, Inc. All rights reserved.



An event is not persistent. If a process waits for an event after it occurs, the process can go on waiting forever.

The first example illustrates this. The forked blocks can be scheduled in any order. It is very likely that one of the event controls is too late – its associated event has already occurred.

The second example utilizes the nonblocking event trigger, which schedules the event for the NBA region, after the event control has had a chance to wait for that event. The example also illustrates use of the `triggered` event property, which persists to the end of the time step. If the `e2` event has already occurred when the `wait` statement executes, the `triggered` property is still true.

## Defining a Mailbox

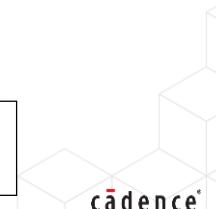


IEEE 1800-2012 15.4

- Mailboxes are a message-based synchronization mechanism.
  - Used for passing messages where order is important (FIFO).
  - Intended for interprocedure communication and synchronization.
- Mailboxes can block procedure execution:
  - Writing to a full mailbox blocks until the mailbox is no longer full.
  - Reading from an empty mailbox blocks until the mailbox is no longer empty.
- Mailboxes can be bound (to a set size) or unbound:
  - Unbounded mailboxes can never be full.
- Mailboxes can be type-less or parameterized:
  - Each type-less (default) mailbox can contain data of different types.
  - Parameterized mailboxes can contain only one type of data.
  - Mailbox connections.

```
mailbox hugebox = new; // mailbox of unlimited size
mailbox fourbox;
fourbox = new(4); // mailbox of size 4
```

379 © Cadence Design Systems, Inc. All rights reserved.



Mailboxes are a class-like, message-based process synchronization mechanism.

A mailbox is conceptually like a mailbox, with a delivery slot on one side and a door on the other slide for retrieving mail.

When you create the mailbox, you pass to its constructor the maximum capacity of the mailbox. If you do not pass a number, the mailbox has unlimited capacity and can never be full.

A mailbox by default accepts messages of any type. Later slides show you how to restrict a mailbox to just one type.

Any process can place messages in the mailbox. Message placement (`put` operation) can be blocking or nonblocking. A blocking put will suspend its process if the mailbox is full. The process will be suspended until the mailbox is no longer full, i.e., another process retrieves a message from the mailbox. A nonblocking put will not suspend if the mailbox is full, but simply returns a 0 to indicate the put has failed.

Messages become available for retrieval in the same order that they are placed in the mailbox.

Any process can retrieve messages from the mailbox. Message retrieval (`get` operation) can be blocking or nonblocking. A blocking get will suspend its process if there the mailbox is empty. The process will be suspended until the mailbox is no longer empty, i.e., another process puts a message in the mailbox. A nonblocking get will not suspend if the mailbox is empty, but simply returns a 0 to indicate the get has failed.

## Mailbox Methods

| Method     | Description                                                                                                                     | Syntax                                                          |
|------------|---------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| new()      | Mailbox constructor which optionally specifies maximum size                                                                     | function new(int bound = 0);<br>Note by default no maximum size |
| num()      | Returns the number of messages currently in the mailbox                                                                         | function int num();                                             |
| put()      | Places a message in the mailbox<br>• Blocks if mailbox full                                                                     | task put (<message>);                                           |
| try_put()  | Places a message in a mailbox<br>• Returns 0 if mailbox full                                                                    | function int try_put (<message>);                               |
| get()      | Retrieves a message from the mailbox<br>• Blocks if mailbox empty<br>• Error if type mismatch                                   | task get (ref <variable>);                                      |
| try_get()  | Retrieves a message from the mailbox<br>• Returns +int if successful<br>• Returns 0 if empty<br>• Returns -int if type mismatch | function int try_get (ref <variable>);                          |
| peek()     | Copies a message from the mailbox<br>• Blocks if mailbox empty<br>• Error if type mismatch                                      | task peek (ref <variable>);                                     |
| try_peek() | Copies a message from the mailbox<br>• Returns +int if successful<br>• Returns 0 if empty<br>• Returns -int if type mismatch    | function int try_peek (ref <variable>);                         |

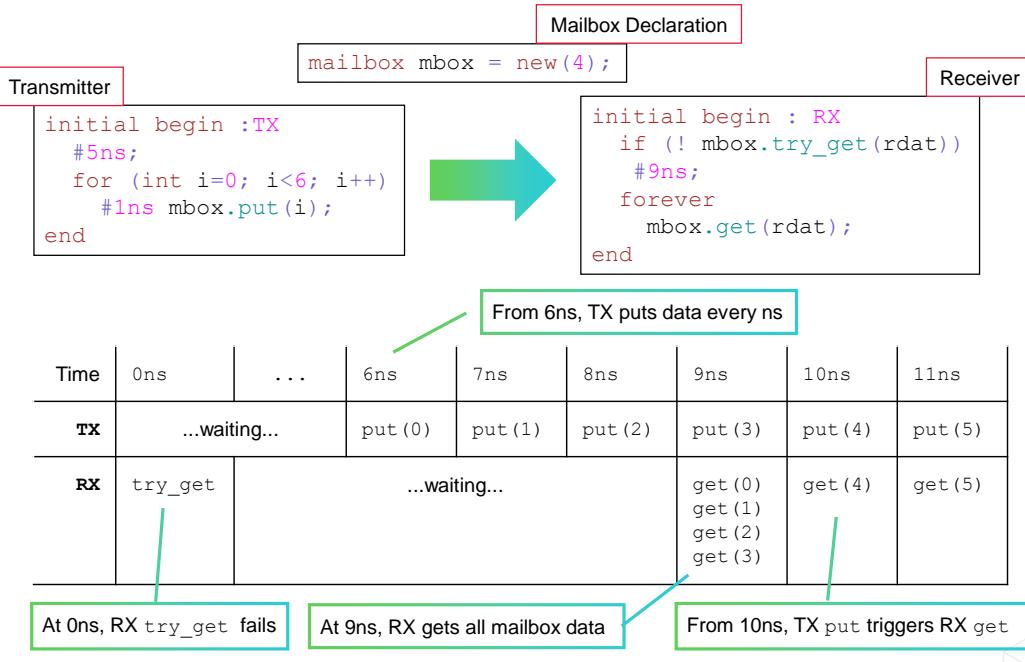
380 © Cadence Design Systems, Inc. All rights reserved.



- The new () constructor constructs a mailbox of an optionally limited size.
- The num () function returns the number of messages currently in the mailbox.
- The blocking put () task attempts to place a message in the mailbox and blocks if the mailbox is full.
- The nonblocking try\_put () function attempts to place a message in the mailbox and returns 0 if the mailbox is full.
- The blocking get () task attempts to retrieve a message from the mailbox into the variable argument. It blocks if the mailbox is empty. It is an error if the type of the message at the head of the mailbox queue is incompatible with the type of the argument variable.
- The nonblocking try\_get () function attempts to get a message from the mailbox into the variable argument. It returns 0 if the mailbox is empty and returns a negative integer if the type of the message at the head of the mailbox queue is incompatible with the type of the argument variable.
- The peek () task and try\_peek () function operate similarly to the get () task and try\_get () function but copy the message instead of removing it.
- Use the try\_get () and try\_peek () functions when you do not know what type of message is at the head of the mailbox queue.

Although you might never use a mailbox, the communication concepts are very similar to Transaction Level Modeling (TLM), which is the standard class communication protocol used by SystemC and SystemVerilog *Universal Verification Methodology* (UVM).

## Example: Process Synchronization with a Mailbox



381 © Cadence Design Systems, Inc. All rights reserved.



The example uses a mailbox `mbox` of size 4.

At 0ns, the RX process executes a `try_get` on `mbox`. This returns 0 as `mbox` is empty and so RX waits for 9ns.

After 5ns, TX starts putting data into `mbox` every ns (from 6ns to 11ns).

After the 9ns delay, RX executes `get` in a forever loop with no delays. This allows RX to read the entire contents of `mbox`, including data 3 which is put by TX at 9ns. This transfer is independent of the execution order of the TX and RX procedures. If TX is executed first, RX can get all data from 0 to 3 in a single execution. If RX is executed first, it can get data 0 to 2. Then when TX puts data 3, this retriggers RX to get data 3 in a second execution in the same timeslot. Hence data transfer is immune to race conditions in procedural execution.

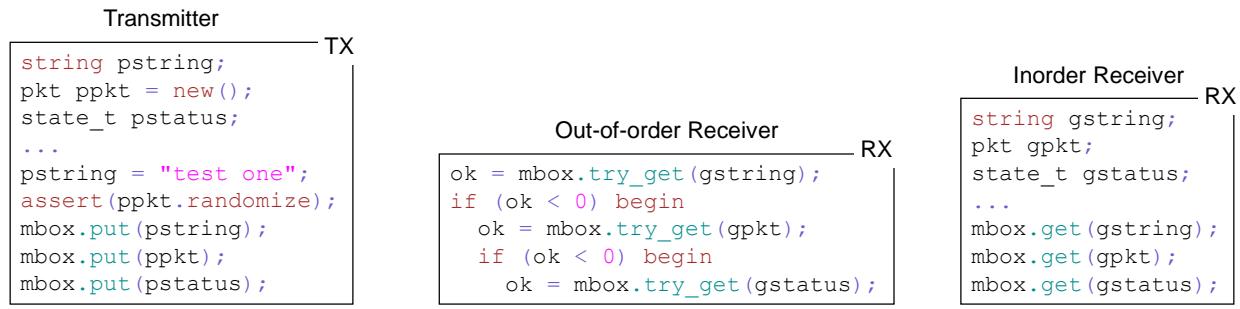
From 10ns, every TX put triggers the RX get.

## Example: Use of Typeless Mailbox

A mailbox can hold messages of different types.

- Blocking get/peek into incompatible type gives a runtime error.
- Nonblocking `try_get/try_peek` into incompatible type returns a negative integer.
  - Possible solution – try all types?

```
Common declarations
class pkt;
 rand bit [4:0] addr;
endclass
typedef enum {pass, fail} state_t;
mailbox mbox = new;
```



382 © Cadence Design Systems, Inc. All rights reserved.



A mailbox, by default, accepts messages of any type.

When retrieving a message using blocking `get()` or `peek()`, the type of the variable passed as the method argument must be compatible with the type of the message at the head of the mailbox queue. Otherwise, a runtime error occurs. Unless you always put and get messages in a preordained type order, it is unlikely you could avoid such errors.

One option is to use the nonblocking `try_get()` or `try_peek()` methods. These return a negative integer if the argument type is incompatible with the type of the message at the head of the mailbox queue. By checking for a negative return integer, you could iterate through all the expected types the mailbox could contain until a match is found.

A more simple alternative is to use a separate mailbox for each message type.

## Parameterized Mailboxes



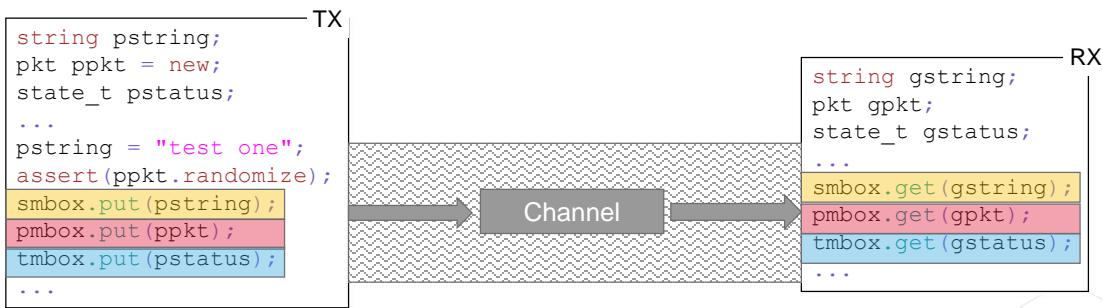
IEEE 1800-2012 15.4.9

You can type-parameterize a mailbox:

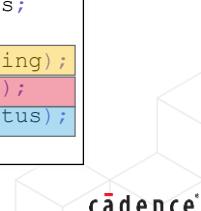
- Define type upon declaration.
- Holds messages of that and equivalent types.
- Compiler detects type mismatch.

```
Common declarations
class pkt;
 rand bit [4:0] addr;
endclass
typedef enum {pass, fail} state_t;

mailbox #(string) smbox = new;
mailbox #(pkt) pmbox = new;
mailbox #(state_t) tmbox = new;
...
```



383 © Cadence Design Systems, Inc. All rights reserved.



You can simply use a separate mailbox for each message type.

You specify the one type a mailbox may accept by overriding its type parameter when you declare the mailbox variable.

The compiler can now detect any attempt to store or retrieve messages of an incompatible type.

## Defining a Semaphore



IEEE 1800-2012 15.3

- Semaphores are a key-based synchronization mechanism.
- They are intended for process synchronization, mutual exclusion, and controlling access to a shared or limited resource.
  - A procedure requests semaphore key(s) before accessing the resource.
  - A procedure returns semaphore key(s) after accessing the resource.
- Semaphores can block procedure execution.
  - If the procedure requests more keys than the semaphore holds, execution is blocked until sufficient keys are returned.
- Key management is the responsibility of the designer.

```
semaphore keybox = new(5); // semaphore with 5 keys

semaphore sync;
sync = new(4); // semaphore with 4 keys
```

384 © Cadence Design Systems, Inc. All rights reserved.



Semaphores are a class-like key-based process synchronization mechanism.

A semaphore is conceptually like a basket of keys.

When you create the semaphore, you pass, as an argument to the constructor, the initial number of keys you want the semaphore to hold. If you do not pass an argument, the semaphore has no keys.

When a process needs to access a resource, it requests a number of keys, by default one, from the semaphore.

Requests can be blocking or nonblocking. A blocking request will suspend the process if there are insufficient keys in the semaphore, until sufficient keys are available. A nonblocking request will return 0 if insufficient keys are available. The order in which processes block is the order in which they are later serviced.

The process returns some or all of its keys when it no longer needs a portion, or any, of the resource.

This is all the semaphore does. It is up to the user to define, manage and utilize the resource.

## Semaphore Methods

| Method                 | Description                                                                                                                                                 | Syntax                                                                                 |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| <code>new()</code>     | Semaphore constructor which specifies number of keys                                                                                                        | function <code>new</code> (int keyCount = 0);<br>Note default number of keys set is 0. |
| <code>get()</code>     | Extracts a set number of keys from the semaphore <ul style="list-style-type: none"> <li>Blocks if the keys are not available</li> </ul>                     | task <code>get</code> (int keyCount = 1);                                              |
| <code>try_get()</code> | Extracts a set number of keys from the semaphore without blocking <ul style="list-style-type: none"> <li>Returns 0 if the keys are not available</li> </ul> | function int <code>try_get</code> (int keyCount=1);                                    |
| <code>put()</code>     | Returns a set number of keys to the semaphore                                                                                                               | function void <code>put</code> (int keyCount=1);                                       |



- The `new()` constructor constructs a semaphore with the specified number of keys. The default value of its argument is 0 – no keys.
- The blocking `get()` requests one or more keys and suspends the calling process if the requested number of keys are not available. The blocked process resumes when sufficient keys are put into the semaphore.
- The nonblocking `try_get()` returns 0 if the requested number of keys are not available.
- The `put()` method puts keys into the semaphore. The user is responsible for key management, for example, to ensure that a process returns only those keys that it previously retrieved.

## Example: Process Synchronization with Semaphore

- sem1 is a single-key semaphore.
- Processes access the resource between calls to get() and put():
  - Process P1 requests a key at time 0, gets it, and accesses the resource.
  - Process P2 requests a key at time 1 and blocks waiting for a key.
  - Process P1 completes its resource access and returns the key.
  - Process P2 unblocks, gets the key, and accesses the resource.
  - Process P2 completes its resource access and returns the key.

```
initial begin : P1
sem1.get();
writemem(addr, data);
#2ns;
sem1.put();
...
```

```
semaphore sem1;
sem1 = new(1);
```

```
initial begin : P2
#1ns;
sem1.get();
writemem(addr, data);
#1ns;
sem1.put();
...
```

386 © Cadence Design Systems, Inc. All rights reserved.



In this example:

- At time 0, process P1 requests a key, gets it, and accesses the resource.
- At time 1, process P2 requests a key and blocks, waiting for the key.
- At time 2, process P1 completes its resource access and returns the key.
  - Process P2 now unblocks, gets the key, and accesses the resource.
- At time 3, process P2 completes its resource access and returns the key.

Remember the user is responsible for defining, managing and utilizing the resource. Common user errors include:

- Accessing the resource without first obtaining the requisite number of keys.
- Failing to return keys after completing the resource access.
- Returning more keys to the semaphore than were taken out of the semaphore.



## 1. Event Variables

A SystemVerilog event variable is a handle pointer to a synchronization queue.

- You can assign and compare these handles to each other.
  - Assignment causes both handles to *point* to the same queue.

```
...
event e1, e2;
initial
 fork
 #1 e2 = e1; // e1, e2 both now have e1 synchronization queue
 #2 @e1 $display ("e1 triggered");
 #2 @e2 $display ("e2 triggered");
 #3 -> e2; // trigger e2 (and also e1)
 join
...

```

Output

```
e1 is triggered
e2 is triggered
```

387 © Cadence Design Systems, Inc. All rights reserved.


A SystemVerilog event variable is a handle that *points* to a synchronization queue. The queue is a list of processes *waiting* for the event.

This example declares two event variables and then assigns one to the other. This makes both event variables *point* to the same list of processes. Processes can access this synchronization queue through either variable.

## 2. Merging Events

Merging events merges only the event *variables*.

- Both now “point” to the RHS synchronization queue.

```
...
event e1, e2;
initial
 fork
 #1 @e1 $display ("e1 triggered");
 #1 @e2 $display ("e2 triggered");
 #2 e2 = e1; // e1, e2 both now have e1 synchronization queue
 // e2 synchronization queue is lost
 #3 -> e2; // trigger e2 (and also e1)
 join
...
...
```

Output e1 is triggered



Event merging is really just the merging of the event variables. The source synchronization queue is assigned to the target event variable, so that both variables “point” to the same queue. If no other variable still “points” to the synchronization queue of the target variable, processes on that queue can wait forever.

### 3. Reclaiming Events

You can assign the `null` value to an event.

- Triggering a null event shall have no effect.
- The effect of waiting on a null event is undefined.

```
module test;
 event e1;
 initial
 fork
 #1 @e1 $display ("e1 triggered once");
 #2 -> e1; // trigger e1
 #3 e1 = null; // e1 synchronization queue lost
 #4 @e1 $display ("e1 triggered twice");
 #5 -> e1; // nothing happens
 join
 endmodule
```

Output `e1 is triggered once`

Event `e1` is set to `null`  
synchronization queue is released

At time 4 `e1` is `null`  
`@e1` may not block or  
may block forever

Triggering null event  
`e1` has no effect

Assigning the null value to an event variable assigns a “null” synchronization queue. This disassociates the variable from its previous synchronization queue. When a synchronization queue is no longer associated with any event variable, the simulator can reuse the queue. The effect of waiting on a null event is undefined. An implementation may choose to wait forever or not wait at all, without warning, and triggering the null event has no effect.

## Module Summary

This module explored interprocess synchronization:

- Enhanced events
  - Help prevent race conditions
  - Provide more control over order of execution
- Mailboxes
  - Multiple-type FIFO operations with built-in blocking synchronization
- Semaphores
  - Multi-purpose synchronization mechanism with built-in blocking and request “weighting” with keys



*This page does not contain notes.*

## Quiz

Where would you use the nonblocking event trigger?

Use a nonblocking event trigger anywhere that a process can potentially “wait” for the event at the same simulation moment that another process triggers the event, thus causing a “race” condition.

What does a blocking `get()` or `peek()` do if the message at the head of the mailbox queue is the wrong type?

This is a runtime error. You should instead use the nonblocking versions.

What happens if a process puts more keys in a semaphore than the process got from the semaphore?

The semaphore capacity increases. SystemVerilog does not check!

*This page does not contain notes.*

## Labs



### Lab 21 Synchronizing Processes with a Mailbox

- You synchronize multiple processes by use of a mailbox.

### Lab 22 Synchronizing Processes with a Semaphore

- You synchronize multiple processes by use of a semaphore.

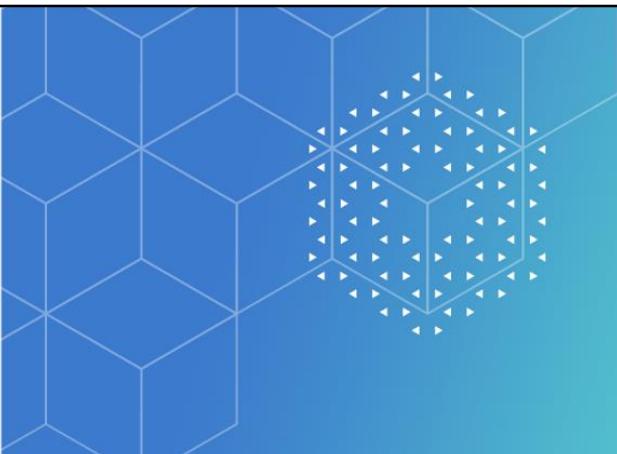
### Lab 23 Synchronizing Processes with Events (Optional)

- You evaluate the SystemVerilog enhanced event constructs.

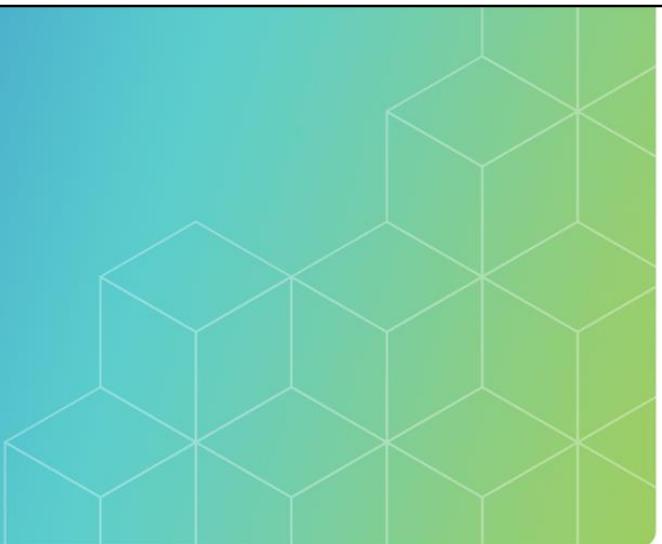
392 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*



## Next Steps



**Module** **24**

Revision **1.0**  
Version **1.0**

Estimated Time:  
• Lecture **5 minutes**  
• Lab **NA**

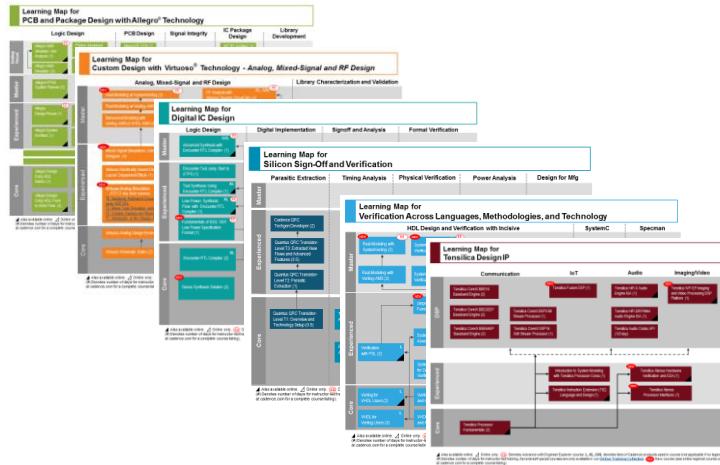
**cadence®**

*This page does not contain notes.*

## Learning Maps

Cadence® Training Services learning maps provide a comprehensive visual overview of the learning opportunities for Cadence customers.

Click [here](#) to see all our courses in each technology area and the recommended order in which to take them.



394 © Cadence Design Systems, Inc. All rights reserved.



Go here to view the learning maps:

[http://www.cadence.com/Training/Pages/learning\\_maps.aspx](http://www.cadence.com/Training/Pages/learning_maps.aspx)

## Cadence Learning and Support

The screenshot shows the Cadence Learning and Support website. At the top, there's a navigation bar with links for Cases, Tools, IP, Resources, Learning, Software, My Support, and Contribute Content. To the right of the navigation are a bell icon and a user profile icon. The main header features the Cadence logo with the tagline "LEARNING & SUPPORT". Below the header is a search bar with a play button overlay. The search bar includes dropdowns for "All Content" and "Start your search here...", and buttons for "View History" and "Documents Liked". The background of the page has a blue and green circuit board design. A large play button is overlaid on the search bar. Below the search bar, there's a section titled "Know more about a product: Choose a product..." with a dropdown menu. Further down, there are six icons with labels: "Installation & Licensing", "Product Manuals", "Training Courses", "What's New", "Troubleshooting Information", and "Video Library". A text overlay states: "Customer Support now includes over 2000 product/language/methodology videos ("Training Bytes")!". The bottom left corner contains copyright information: "395 © Cadence Design Systems, Inc. All rights reserved." The bottom right corner features the Cadence logo.

Click the play button in the figure on this slide to view the demo of Cadence Learning and Support.

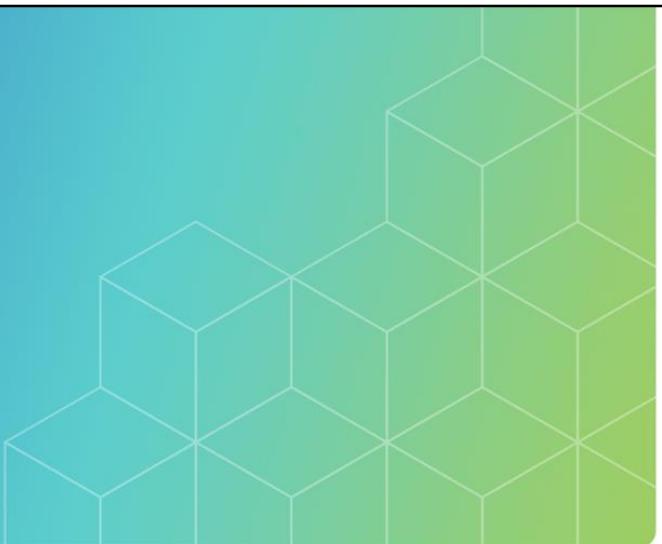
## Wrap Up

- Complete Post Assessment, if provided
- Complete the Course Evaluation
- Get a Certificate of Course Completion

# Thank you!



*This page does not contain notes.*



## Verilog-2001 Summary

### Appendix

### A

Revision

1.0

Version

21.10

Estimated Time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*



## Appendix Objectives

In this appendix, you summarize key Verilog-2001 features, including the following:

- Sensitivity list enhancements
- Input/output lists
- localparam
- Subroutine enhancements
- Variable initialization
- Multidimensional arrays and array selects
- Configurations



*This page does not contain notes.*

## Sensitivity List Enhancements



IEEE 1800-2012 23.3.2

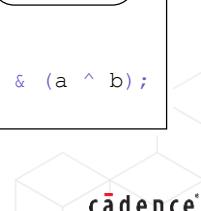
- Verilog-1995: Sensitivity list signals are separated with `or`.
  - Confused with the logical OR (`|`) operator.
- Verilog-2001: You can separate sensitivity list signals with a comma.
- Verilog-2001: You can define a wildcard sensitivity list with `*`.
  - Automatically sensitive to any signal read.
  - Purely combinational logic *only*.

```
Verilog-1995
always @(a or b or cin)
begin
 sum = a ^ b ^ cin;
 carry = (a & b) | cin & (a ^ b);
end
```

```
Verilog-2001
always @(a, b, cin)
begin
 sum = a ^ b ^ cin;
 carry = (a & b) | cin & (a ^ b);
end
```

```
Verilog-2001
always @(*)
begin
 sum = a ^ b ^ cin;
 carry = (a & b) | cin & (a ^ b);
end
```

399 © Cadence Design Systems, Inc. All rights reserved.



In Verilog-1995, the signals in the event list of an `always` block are separated with the keyword `or`. This can be confused with the logical or operator (`|`). Therefore in Verilog-2001, the signals can be separated with commas instead of `or`.

In Verilog-2001, you can define a wildcard sensitivity list for combinational logic only. A wildcard sensitivity list replaces the signal list with a single asterisk. This makes the `always` block automatically sensitive to every signal read within the block. There is one exception to wildcard sensitivity, however. If a combinational `always` block contains a call to a function, which reads a signal which is not defined as a formal argument to the function (i.e., it is accessed via side effects), then this signal will not be included in the sensitivity list. In other words, when the list of signals for the wildcard sensitivity list is created, only the arguments of any functions calls are considered, not the contents of the function.

## Initialization and Local Declarations



IEEE 1800-2012 9.3.4

- Verilog-1995: Declaration and initialization of a variable are separate.
- Verilog-2001: You can combine declaration and initialization of a variable.
- Verilog-2001: You can declare local variables in `initial` or `always` blocks.
  - For example, loop variable.
  - Restrictions:
    - Variable is not visible outside block.
    - Block must be named.

|                                                           |                                            |
|-----------------------------------------------------------|--------------------------------------------|
| Verilog-1995<br><pre>reg clk; initial   clk = 1'b1;</pre> | Verilog-2001<br><pre>reg clk = 1'b1;</pre> |
|-----------------------------------------------------------|--------------------------------------------|

|                                                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Verilog-2001<br><pre>initial begin : IBLK // named block // local declaration integer i; for(i = 0; i&lt;=7; i = i+1)   if (avec[i])     count = count + 1; end</pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|

400 © Cadence Design Systems, Inc. All rights reserved.



In Verilog-1995, if you want to initialize a variable at the start of the simulation, then this must be done in a `initial` block, separate from the declaration. In Verilog-2001, the initial value can be combined into the declaration.

In Verilog-2001, you can also have local variable declarations in an `initial` or `always` block. These are declared after the `begin` of the block, and before any executable statements. All local declarations must be grouped together at the beginning of the block – once the first executable statement is found, any further local declarations will cause compiler errors.

Local declarations are visible only within their declaration block, and override any identifiers with the same name declared in higher scopes, e.g., the enclosing module.

It is a requirement in Verilog-2001 that a block containing local declarations must be named. This is done by defining a name after the `begin` keyword of the block, with a colon separator between name and `begin`.

A good use for local declarations is `for` loop variables. In Verilog-1995, for loop variables can only be declared at the module level, giving a risk of multiple loops in different procedural blocks all sharing the same loop variable. With local declarations, each block can have their own, independent loop variable.

```
`default_nettype none
```



IEEE 1800-2012 22.8

Verilog-1995: Undeclared identifiers in instance port maps default to a single bit `wire`.

- Default net type can be overridden using the `default_nettype` compiler directive.

```
`default_nettype wand
```

Verilog-2001: You can disable default:

```
`default_nettype none
```

- Use of undeclared identifier generates a compiler error.
- Prevents connectivity problems from errors in variable names.
- `none` is not a reserved word.

```
module mone (ip, ...);
 input ip;

 assign undec = 1'b0; // wire
 ...

```

```
`default_nettype tri Verilog-1995
module mone (ip, ...);
 input ip; // tri

 assign undec = 1'b0; // tri
 ...

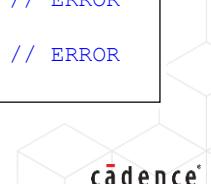
```

```
`default_nettype none Verilog-2001
module mone (ip, ...);
 input ip; // ERROR

 assign undec = 1'b0; // ERROR
 ...

```

401 © Cadence Design Systems, Inc. All rights reserved.



Undeclared identifiers have an implicit declaration of a single bit `wire` by default. There is a compiler directive which can change the default net type – ``default_nettype`. With this directive, the default net type can be changed from `wire` to another net type, for example `tri`.

The compiler directive must be placed outside of a module and applies across file boundaries until overridden by another default net type compiler directive or a ``resetall` is executed.

Multiple net type directives are allowed, but subsequent directives overwrite previous ones and so default net type for the whole design will be defined by the last directive encountered.

In Verilog-2001, the option `none` can be applied to the directive. In this case, undeclared identifiers will generate compiler errors. The aim of the directive is to help debug connectivity problems where a misspelled identifier name results in a new wire declaration, which breaks a connection.

However, implicit net types are used extensively in traditional Verilog code, so setting the default net type to `none` can result in a great many error messages. For example, the following would generate compiler errors with a default net type of `none`:

```
module mone (ip1, ip2, ...);

 input ip1, ip2;
 ...

```

The ports would have to be explicitly defined as wires to avoid errors:

```
module mone (ip1, ip2, ...);

 input wire ip1, ip2;
 ...

```

## Input/Output Declarations



IEEE 1800-2012 22.8

```
module mux4 (a, b, sel, op);
 input [3:0] a, b;
 input sel;
 output [3:0] op;
 reg [3:0] op;
 wire [3:0] a, b;
 wire sel;
 ...
endmodule
```

Verilog-1995

```
module mux4 (a, b, sel, op);
 input wire [3:0] a, b;
 input wire sel;
 output reg [3:0] op;
 ...
endmodule
```

Verilog-2001

```
module mux4 (input wire [3:0] a,
 input wire [3:0] b,
 input wire sel,
 output reg [3:0] op);
 ...
endmodule
```

Verilog-2001

ANSI C style port declaration

cadence®

- You can combine port and type declarations.
- You can use ANSI C style input/output declarations:
  - For modules, tasks and functions.

402 © Cadence Design Systems, Inc. All rights reserved.

In Verilog-1995, each port must be first named in the module header, then sized and given direction in a separate declaration and finally the data type must be defined. In these examples, we are explicitly declaring the net data types as well as the registers.

In Verilog-2001, you can combine the direction, size and data type declarations into one statement. You can also combine the name, direction, size and type declarations into the module header. This is called an ANSI-C-style input/output declaration, and can also be used for subprogram argument lists.

It greatly simplifies the port or argument declarations.

## Parameterized Input/Output Declarations

```
Verilog-2001
module mux_size
 #(parameter SIZE=3)
 (input wire [SIZE-1:0] a,
 input wire [SIZE-1:0] b,
 input wire sel,
 output reg [SIZE-1:0] op);
 ...
endmodule
```

```
Verilog-1995
mux_size #(5) u1
(.a(a), .b(b),
 .sel(sel), .op(op));
```

```
Verilog-2001
mux_size #(.SIZE(5)) u1
(.a(a), .b(b),
 .sel(sel), .op(op));
```

- Verilog-2001: You can declare module parameters before the ANSI-C-style port list.
- Verilog-2001: You can override module parameter by name in the instance parameter map.

403 © Cadence Design Systems, Inc. All rights reserved.



To parameterize port sizes for an ANSI-C-style port list, declare the module parameters before the port list, enclosed in separate parentheses and prefaced with a hash #.

For Verilog-1995, you map parameters by position, using commas as placeholders for parameters you do not override:

```
param_mod #(1, 2, 3) U1 (.a(a), .b(b), .c(c));
param_mod #(, , 3) U2 (.a(a), .b(b), .c(c));
```

You must know the parameter declaration order. This method can lead to errors with badly ordered parameter passing.

For Verilog-2001, you can map parameters by name similarly to mapping ports by name:

```
param_mod #(.P1(1), .P2(2), .P3(3)) U2 (.a(a), .b(b), .c(c));
```

This method reduces the likelihood of error.

## Local Parameters and Parameter Passing

- Localparams are true constants.
- Unlike parameters, localparams cannot be overridden from the next level of hierarchy.
- Verilog-2001 provides three ways to override module parameters:
  - Redefinition using defparam.
  - Positional parameter override during instantiation.
  - Named parameter override during instantiation.
    - New in Verilog-2001.

Verilog-2001

```
module us_mult (a,b,product);

parameter width_a = 5;
parameter width_b = 5;
localparam op_width = width_a + width_b;

input [width_a-1:0] a;
input [width_b-1:0] b;
output [op_width-1:0] product;

assign product = a * b;

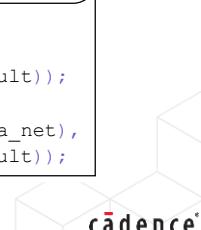
endmodule
```

Verilog-2001

```
defparam u2.width_b = 7;

us_mult #(5,7) u1 (.a (a_net),
 .b (b_net), .product (a_b_mult));

us_mult #(.width_a(5)) u1 (.a (a_net),
 .b (b_net), .product (a_b_mult));
```



*This page does not contain notes.*

## Signed Vectors



IEEE 1800-2012 A.2.2.1

- signed defines vector types as signed quantities.
  - 2's complement
- Also for function returns.
- Signed arithmetic for vectors is easier.
- Assignments between signed reg and integer types maintain sign information.
- Vector types can also be defined as unsigned.
  - Default behavior

```
reg [3:0] usreg; // 4-bit vector in range 0 to 15
reg signed [3:0] sreg; // 4-bit vector in range -8 to 7
```

```
function signed [15:0] add_2_signals
 ...
endfunction
```

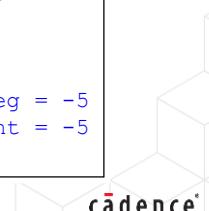
```
module smult (input signed [7:0] opa, opb;
 output signed [15:0] product);
 assign product = opa * opb;
endmodule
```

```
reg signed [3:0] areg;
integer aint = -5;

initial begin
 areg = aint; // areg = -5
 aint = areg; // aint = -5
 ...

```

405 © Cadence Design Systems, Inc. All rights reserved.



For Verilog-2001, you can declare signed vectors. Vectors are by default unsigned. An unsigned vector is treated as signed for accesses through a signed port.

## Signed Support Features

- Arithmetic shift operators maintain sign information.
  - Right >>>
  - Left <<<
- Literal values can be qualified as signed with s.
  - 4'sb1001
- New system functions cast a value as signed or unsigned:
  - \$signed()
  - \$unsigned()

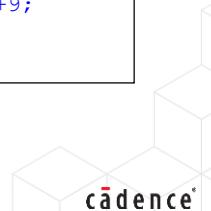
```
reg signed [7:0] a,b;
reg signed [3:0] c;
reg signed [4:0] d;

initial
begin
a = 8'b11001101;
// logical shift right 2 places
b = a >> 2; // b = 8'b00110011

// arithmetic shift right 3 places
b = a >>> 3; // b = 8'b11111001

c = 4'sb1001; // c = -7
d = $unsigned(c); // d = +9;

end
```



*This page does not contain notes.*

## Automatic Width Extension



IEEE 1800-2012 A.2.2.1

Verilog-1995

```
reg [63:0] data;
data = 'bz; //'h00000000zzzzzzzz
data = 64'bz; //'hzzzzzzzzzzzzzzzz
```

Verilog-2001

```
reg [63:0] data;
data = 'bz; //'hzzzzzzzzzzzzzzzz
```

### Verilog-1995

- Assigning an unsized x or z value (e.g., 'bz) to a bus greater than 32 bits only sets the lower 32 bits (unsized literals are assumed to be 32 bits).
- Upper bits are set to 0.
- Setting the entire bus to x or z requires explicitly specifying the number of bits in the value.

### Verilog-2001

- An unsized z or x value automatically expands to fill the full width of the target vector.

407 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*

## Automatic Tasks



IEEE 1800-2012 13.3.1

- Verilog-1995: All tasks and functions are static.
- Concurrent task call and recursive function calls must be avoided.
  - Can cause conflict with internal variables and arguments.
- Verilog-2001: You can declare tasks and functions to be dynamic.
  - Using keyword `automatic`.
  - Allows concurrent task calls and recursive function calls.
- Automatic task declarations cannot be accessed by hierarchical references.
- Automatic arguments cannot be updated with a nonblocking assignment.

Verilog-2001

```

task automatic neg_clocks
 (input [31:0] number_of_edges);
begin
 repeat(number_of_edges)
 @ (negedge clk);
end
endtask

initial begin
 neg_clocks(6);
 ...
end

always @ (posedge trigger)
begin
 neg_clocks(10);
 ...
end

```

408 © Cadence Design Systems, Inc. All rights reserved.



For Verilog-1995, all subroutines are static. Only one copy of a subroutine arguments and variables exists. Multiple concurrent task calls and recursive function calls all use the same copy of inputs, local variables, and outputs.

For Verilog-2001, you can declare an automatic subroutine. A separate copy of the subroutine arguments and variables exists for each invocation. Multiple concurrent task calls and recursive function calls all use their own copy of inputs, local variables, and outputs. As the automatic arguments and variables exist only for the duration of the subroutine execution, you cannot reference them hierarchically from outside the subroutine, cannot assign to them with nonblocking assignments, and cannot use `$monitor` or `$strobe` with them.

## Constant Functions



IEEE 1800-2012 13.4.3

- Verilog-1995 allows simple constant expressions for defining limits for vectors, replicates, etc.
- Verilog-2001 allows limits to be defined by constant functions.
  - Function value must be calculable at elaboration.
    - Usually inputs are constant.
  - Greater flexibility for scalable reusable models.
  - Multiple limits can be derived from a single constant .

Verilog-2001

```
parameter addw = 5;
parameter datw = 8;

reg [addw-1:0] address;
reg [datw-1:0] mem [1:memsize (addw)];

function integer memsize;
 input [15:0] width;
begin
 case (width)
 ((width%2)==0) : memsize = 1024;
 default: memsize = 512;
 endcase
end
endfunction
```

409 © Cadence Design Systems, Inc. All rights reserved.



Verilog-1995 module parameter values must be constant expressions, which are limited to operations on literals and previously declared module parameters.

Verilog-2001 permits you to use a constant function call anywhere you are required to use a constant expression.

The standard lists restrictions upon constant function calls:

- Cannot be placed within any generate scope.
- Cannot contain hierarchical references.
- Cannot contain system function calls.
- Ignores system task calls (except will execute \$display in simulator but not elaborator).
- Cannot themselves make constant function calls in any context requiring constant expressions.
  - Can otherwise make constant function calls to functions local to containing module.
- Can access only functions and module parameters and nothing else declared outside the function definition.
- Module parameters must be previously assigned use of defparam can produce undefined results.

## Multidimensional Arrays and Part Selects



IEEE 1800-2012 H.11.2

### Verilog-1995:

- One dimensional array only of integer, reg, reg[range], time.
- You *cannot* directly select a bit or part of an array element.

### Verilog-2001:

- Multidimensional arrays integer, reg, reg[range], real, realtime, time and nets.
- You *can* directly select a bit or part of an array element.

```
// a 16 x 256 array of 32 bit words
reg [31:0] array2 [0:255][0:15];

// a part select of the most significant byte of
// a 32 bit word with address 100,7
reg [7:0] out2 = array2[100][7][31:24];

// bit select of word at address 100,7,15
reg out1 = array2[100][7][15];
```

Verilog-2001

410 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*

## Indexed (Variable) Array Selection

- Verilog-2001 allows an array to be indexed with a variable.
- But only one bound of a slice can be a variable
- Must use an index part select for a variable slice.
- An indexed part select contains the following:
  - Base expression (variable)
  - Width expression (constant)
  - Offset direction:
    - Positive +:
    - Negative -:
- Offset indicates whether the width is added or subtracted from the base.

```
reg [39:0] str = "abcde";
integer i;

initial begin
 for (i=0;i<5;i=i+1)
 // chop str into 8 bit slices
 $display("%s", str[i*8+7:i*8]);
 ...

```



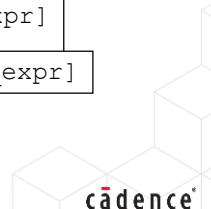
Error

Can only use variable `i`  
in one of the slice bounds

[base\_expr +: width\_expr]

[base\_expr -: width\_expr]

411 © Cadence Design Systems, Inc. All rights reserved.



Verilog lets you index an array with a variable, but if you want to select a slice of an array, only one of the bounds can be a variable expression.

To index a variable slice of an array, you need to use the index part select construct in Verilog.

An indexed part select contains:

- A base expression, which can be a variable expression.
- A width expression, which must be a constant expression.
- An offset direction, which defines whether the width is added to or subtracted from the base.
- [base\_expr +: width\_expr] //positive offset
- [base\_expr -: width\_expr] //negative offset

## Indexed Part Selection in String Array

- Width expression must be constant.
  - Base expression can be a variable expression.
  - Offset indicates whether the width is added or subtracted from the base.
  - For  $i = 4$ , the expression gives:
- ```
$display("%s", str[39:32]);
```
- Lower bound 32 is derived from the base expression.
 - Upper bound 39 comes from the positive offset and width expression.

[base_expr +: width_expr]
 [base_expr -: width_expr]

```
reg [39:0] str = "abcde";
integer i;
initial begin
  for (i=0;i<5;i=i+1)
    $display("%s", str[i*8 +: 8] );
  ...

```

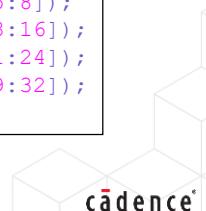
Verilog-2001

Loop expansion

```
reg [39:0] str = "abcde";
integer i;
initial begin
  $display("%s", str[7:0]);
  $display("%s", str[15:8]);
  $display("%s", str[23:16]);
  $display("%s", str[31:24]);
  $display("%s", str[39:32]);
  ...

```

412 © Cadence Design Systems, Inc. All rights reserved.



Offset direction

```
[base_expr +: width_expr] //positive offset
[base_expr -: width_expr] //negative offset
```

```
for (i=0;i<5;i=i+1)
$display("%s", str[i*8 +: 8]);
```

Expands to:

```
$display("%s", str[7:0]); // i = 0, output "e"
$display("%s", str[15:8]); // i = 1, output "d"
$display("%s", str[23:16]); // i = 2, output "c"
$display("%s", str[31:24]); // i = 3, output "b"
$display("%s", str[39:32]); // i = 4, output "a"
```

Generated Instantiation



IEEE 1800-2012 A.4.2

Create multiple instances of:

- Modules, primitives, variables, nets, tasks, functions, continuous assignments, initial blocks, and always blocks.
- Generating module instantiations is the primary use model.

New keywords:

- generate, genvar, endgenerate

Three forms of generate:

- case
- if...else
- for

You must label a generate for block:

- All instances and wires have unique names based on generate lab.

413 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Loop Generate: generate for

Verilog-2001

```
genvar i; //required index type in or outside generate

generate //required keyword
  for (i=0; i<4; i=i+1)
    begin : gendevice //required label for scope naming
      wire n1,n2,n3;
      myxor g1 (n1 , a[i] , b[i]);
      myxor g2 (sum[i] , n1 , c[i]);
    end
  endgenerate //required keyword
```

Label is required to create the generate for scope.



This page does not contain notes.

Generated Hierarchical Names

Hierarchical names are created by adding the index number to the generate block name.

```
Verilog-2001
generate
  for (i=0; i<4; i=i+1)
    begin : gendevice
      wire n1,n2,n3;
      myxor g1 (n1 , a[i] , b[i]);
      myxor g2 (sum[i] , n1 , c[i]);
    end
  endgenerate //required keyword
```

gendevice[1].g1	Hierarchical path of instance of module
gendevice[2].param	Hierarchical path of parameter
gendevice[3].n1	Hierarchical path of net
gendevice[2].g2.out	Hierarchical path of device internal net

This page does not contain notes.

Conditional Generate: generate if...else

```
module multiplier (a, b, product);
    parameter a_width = 8, b_width = 8;
    input [a_width-1:0] a;
    input [b_width-1:0] b;
    output [(a_width+ b_width)-1:0] product;

    generate // if expression must be known at elaboration
        if ( (a_width < 8) || (b_width < 8) )
            CLA_mult #(a_width, b_width) u1 (a, b, product);
        else
            WALLACE_mult #(a_width, b_width) u1 (a, b, product);
    endgenerate

endmodule
```

Verilog-2001

Label is not required to create generate if scope.

416 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Conditional Generate: generate case

```
generate
    case (WIDTH) // expression must be known at elaboration
        1: adder_1bit x1(co, sum, a, b, ci);
        2: adder_2bit x1(co, sum, a, b, ci);
        // others - scalable carry look-ahead adder
        default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);
    endcase
endgenerate
```

Verilog-2001

Label is not required to create generate case instance.



This page does not contain notes.

Nesting Generate Structures

Verilog-2001

```
generate
  for (j=0; j<18; j=j+1)
    begin: nestgen
      if (j==0)
        count1[j] = count0[j];      // genblk1 (implicit name)
      else
        count1[j] = count0[j] & count1[j];
      case (j)                      // genblk2
        1,2:   alu1 U1(a[j], b[j], c[j]);
        default: alu2 U2 (a[j], b[j], c[j]);
      endcase
    end
  endgenerate
```

418 © Cadence Design Systems, Inc. All rights reserved.



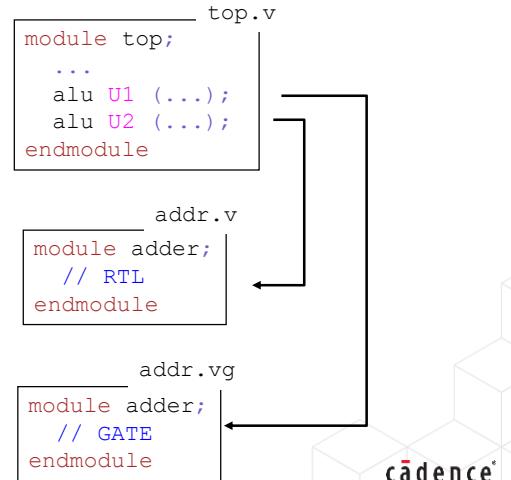
The *generate...for* construct requires an explicit block name. The *generate...case* and *generate...if* constructs do not.

What Are Configurations?



SystemVerilog provides the ability to specify design configurations, which specify the binding information of module instances to specific SystemVerilog source code. Configurations utilize libraries.

- Verilog-1995: No mechanism for module library management:
 - People typically use different module definition names and conditional compilation.
- Verilog-2001: Configurations enable design management:
 - **Virtual library structures** for holding and referencing compiled code.
 - Portable mapping of virtual libraries to specific directories.
 - Binding of an instance to a specific module from a specific library.
 - Identification of top-level design block.

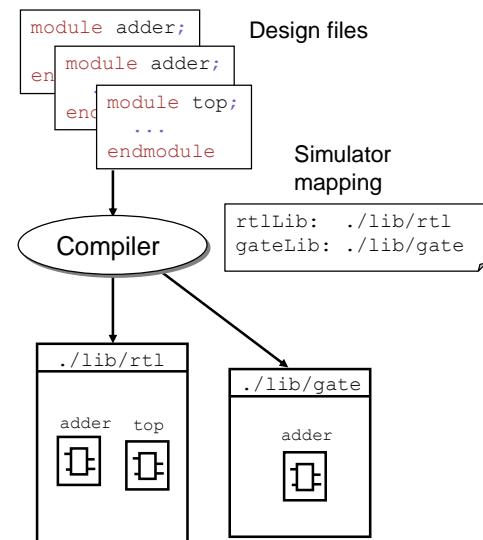


Virtual Compilation Libraries



Compilation Library is a place where the compiler places compiled design objects.

- Can compile design files into one or more virtual libraries:
 - Stores compiled design objects.
 - Large projects can use many.
 - May physically exist in the file system.
 - Vendor-specific implementation maps a file system location to a logical library name.
- Reference library by logical name:
 - Allows portable pathname to any compiled object.
 - Example: `rtlLib.adder`.
- Compiler stores compiled objects according to library declarations in a standard library map file.



420 © Cadence Design Systems, Inc. All rights reserved.



The compiler places compiled design objects in a compilation library.

The compilation library is almost invariably a file system directory of some sort.

The vendor provides a means to specify the physical compilation libraries and map them to logical library names.

The IEEE standard library map file refers to only the logical names and has no knowledge of the actual library implementation.

You reference compiled design elements using the hierarchy separator, i.e., `libName.cellName`.

For the Cadence® Xcelium™ simulator, you provide a file typically called `cds.lib` in which you define the libraries the tool may access and where in the file system they are physically located. You make these statements, one to a line, in the form:

- Define `logicalLibraryName filesystemLocation`

Library Map File (lib.map)

- The map file controls which design files are compiled into which library.
- Referenced at compilation:
 - Command-line option
- All specified files compiled to the defined library.
- Files can be matched explicitly by a wildcard name or directory.
- Precedence should resolve multiple matches.
 - Otherwise, it is an error
- Unmatched files are compiled to a default library named work.



421 © Cadence Design Systems, Inc. All rights reserved.



The compiler stores compiled objects according to library declarations in standard library map file.

Library declaration syntax:

```
library library_identifier file_path_spec [ { , file_path_spec } ]
  • [ -inmdir file_path_spec [ { , file_path_spec } ] ] ;
```

File system path specifications might use wildcards:

- ? matches any single character
- * matches any number of characters in a directory or file name
- ... matches any number of hierarchical directories
- .. matches the parent directory
- . matches the directory containing the library map file

Paths which end in / shall include all files in the specified directory (identical to /*).

Paths which do not begin with / are relative to the directory in which the current lib.map file is located.

Note: if you use a path with // or /* in it, you may need to enclose the path between double quotes ("").

Example

```
library gateLib ./mylib/*.vg;
```

All files with the .vg extension in the ./mylib directory are compiled into the *gateLib* library.

Configurations and Library Declarations

The order of library declarations implies a search order for module bindings.

- rtlLib searched first.
- gateLib then searched for remaining bindings.
- Default worklib searched last.

```
library rtlLib "./rtl/*.v";
library gateLib "./gate/*.v";
include ./gatelibmap.lib;
```

An explicit configuration can:

- Identify top-level module with `design` keyword (compulsory).
- Redefine library search order with `default liblist`.
- Bind all instantiations of a module with `cell`.
- Bind specific instantiations of a module with `instance`.

```
config cfg;
  design rtlLib.top;
  default liblist gateLib rtlLib;
  cell adder use gateLib.adder;
  instance top.U2 use rtlLib.adder;
endconfig
```



The elaborator by default searches libraries for instance definitions in the order you declare the libraries.

You can in a cell configuration declaration specify an alternative default library search order.

You can also in that configuration specify for any cell type or for a specific instance:

- A library search order
- A specific library
- A specific configuration

Configuration Example

library lib1 "./rtl/*.v"; library lib2 "./gate/*.v";	cfg.v
lib.map	config vectest; design lib1.vectest; default liblist lib2 lib1; instance vectest.dut1 liblist lib1 lib2; instance vectest.dut2 use lib1.cpu; cell foo use lib1.foo; endconfig

Library declarations imply the lib1->lib2 search order.

Configuration default liblist specifies the lib2->lib1 search order.

Top-level design element is vectest from lib1.

Library search order for instance dut1 is lib1->lib2.

Bind instance vectest.dut2 to cpu from lib1.

Any instance of cell foo bound to foo from lib1.

All other instances will be bound by searching in the order of the default liblist.

423 © Cadence Design Systems, Inc. All rights reserved.



Cadence Xcelium simulator instance binding rules (highest to lowest precedence):

- Verilog configuration.
- `uselib compiler directive in source code to specify library and/or view for all instances to which the directive applies.
- *-binding* elaborator option to specify library and/or view for all instances of a cell type.
- *-libname* elaborator option to provide an ordered list of libraries to search.
- LIB_MAP and VIEW_MAP environment variables to provide an ordered list of libraries and/or views to search.
- Search for module, then udp views in the order the libraries are defined.
- Search the library of the parent cell for any view of the instance.
- Search all defined libraries for any view of the instance.

Hierarchical Configurations

```
config bottom;
    design lib1.bottom;
    default liblist lib1 lib2;
    instance bottom.a1 liblist lib3;
endconfig
config top;
    design lib1.top;
    default liblist lib2 lib1;
    instance top.bot use lib1.bottom:config;
endconfig
```

- Multiple configurations are allowed in the same libmap file.
- Specific instances can be bound to a configuration.
- The higher level `config` cannot specify binding rules for instances in another configuration.



This page does not contain notes.

Configuration Keywords

```
config <config_name> endconfig
```

- Defines the start and end of a configuration along with the configuration name.

```
design [lib.]<design_unit>;
```

- Specifies top-level design element; one and only one design statement must exist.

```
default liblist <lib1> <lib2>;
```

- Selects all instances not matched with more specific selection clauses.

```
liblist <lib1> <lib2>;
```

- Defines ordered set of libraries to be searched for binding.
- Appears with default, instance, or cell keywords.
- Liblists are inherited hierarchically as instances are bound.
- Overrides order of libraries specified in the libmap file.

```
use [lib.]<design_unit>;
```

- Defines exact library and cell to used to bind a cell or instance.
- Can only be used with instance or cell selection clauses.

425 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Cell and Instance Configuration

Configuration by instance:

- Binds specific instance to specific cell or library search order

```
instance <hierarchy_path> liblist <lib1> <lib2>;
```

or

```
instance <hierarchy_path> use [lib].<design_unit>;
```

Configuration by cell:

- Binds all object instantiations to a specific module or library search order

```
cell <design_unit> liblist <lib1> <lib2>;
```

or

```
cell <design_unit> use [lib.]<design_unit>;
```



The IEEE Std. 1364-2001 does not mention whether the cell clause or the instance clause should have precedence. To ensure portability of your configuration between vendors, you cannot assume that you can override a cell clause for a specific instance of that cell type.

Enhanced File I/O



IEEE 1800-2012 21.3

\$fclose, \$fopen, \$fdisplay[b|h|o], \$fstroke[b|h|o], \$fmonitor[b|h|o], \$fwrite[b|h|o], \$fmonitor[b|h|o], \$readmem<b|h>
\$fgetc, \$ungetc, \$fflush, \$ferror, \$fgets, \$rewind, \$swrite, \$swriteb, \$swriteo, \$swriteh, \$format, \$fscanf, \$sscanf, \$fread, \$ftell, \$fseek

Verilog-1995

Verilog-2001

- New type argument to `$fopen` allows for different modes of opening.

"r" or "rb" - read from existing file
"w" or "wb" - write to new file
"a" or "ab" - append to existing file
"r+", "r+b", "rb+" - read/write (existing)
"w+", "w+b", "wb+" - read/write (new)
"a+", "a+b", "ab+" - append (new or old)

427 © Cadence Design Systems, Inc. All rights reserved.



Verilog-1995 syntax for a file open command is:

```
multi_channel_descriptor = $fopen ( " file_name " );
```

where `multi_channel_descriptor` is an integer with 1 bit set for each open file, leading to a limit of 32 files open at any one time.

Verilog-2001 syntax is:

```
fd = $fopen ( " file_name ", type );
```

where `fd` is an integer with each value representing an open file, and `type` is a character string defining how the file should be opened. The character `b` in the type string defines a binary file which may be important for some operating systems (e.g., Windows).

Simplified description for other file operations: (for full details, see *Verilog-2001 Language Reference Manual*)

`c = $fgetc(fd)` – Reads a byte from the file `fd` into `c`. `c = -1` if read unsuccessful.

`vec = $ungetc(c, fd)` – Pushes byte `c` into buffer for file `fd` so the next `$fgetc` will read `c`. The file itself is not changed, only the read buffer is. `vec = -1` if push unsuccessful.

`$fflush(fd)` – Any remaining data in buffer for file `fd` is written to the file.

`eno = $ferror(fd, str)` – Error number for the latest I/O error on file `fd` written to `eno` and string description of error written to `str` (`str` must be at least 640 bits wide).

... continued on next slide

Example: Reading from a File

```

module testbench;
reg [8:1] char;
integer data_chan;

initial
begin
  data_chan = $fopen("source.dat", "rb");

  while (char !== 255)
    begin : get_char
      char = $fgetc(data_chan);
      if (char == 255)
        begin
          $display ("Finished!");
          //Prevent $display printing out EOF
          disable get_char;
        end
      $display("%s %0d %b", char, char, char);
    end

  $fclose(data_chan);
end
endmodule

```

_!@%^&
abcZ
source.dat

```

95 01011111
! 33 00100001
@ 64 01000000
% 37 00100101
^ 94 01011110
& 38 00100110

10 00001010
a 97 01100001
b 98 01100010
c 99 01100011
z 90 01011010

10 00001010
Finished!

```

`$fgetc` returns -1 (EOF) if an error occurs

428 © Cadence Design Systems, Inc. All rights reserved.



... continued from previous slide

`code = $fgets(str, fd)` – Copies line of data (including newline character) from file *fd* into *str* (or fills *str*) and returns the number of characters read into *code*. *code* = zero on read error.

`code = $rewind(fd)` – Sets the next file operation to the start of the *fd* file.

`$fwrite(b|o|h)` as `$fwrite` but write data as a string to a `reg` variable.

`$sformat` is similar to `$fwrite` except the second argument is interpreted as a format string.

`$fscanf(fd, format, args)` – Reads characters from the *fd* file and interprets them according to the control string *format*. `$sscanf` works similarly with strings instead of files.

`$fread(obj, fd)` – Reads binary data from the *fd* file into *obj* which is a `reg` or memory object. If *obj* is a memory, additional arguments can define a starting address and location count in the memory.

`$ftell(fd)` – Returns the number of the next byte to be read from the *fd* file.

`code = $fseek(fd, offset, op)` – Sets the file pointer (position of the next file operation) for the *fd* file, *offset* from the beginning. The current position or end of the file according to the value of *op* is as shown here:

- *op* = 0 – Sets pointer to beginning of file plus offset
- *op* = 1 – Sets pointer to current position plus offset
- *op* = 2 – Sets pointer to end of file plus offset

offset is a signed value. *code* = -1 if reposition fails (e.g., *op* = 2 and *offset* is positive), 0 if successful.

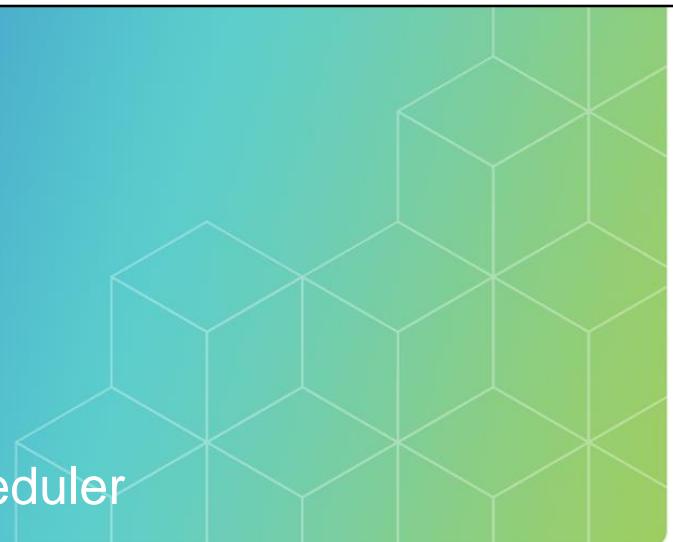
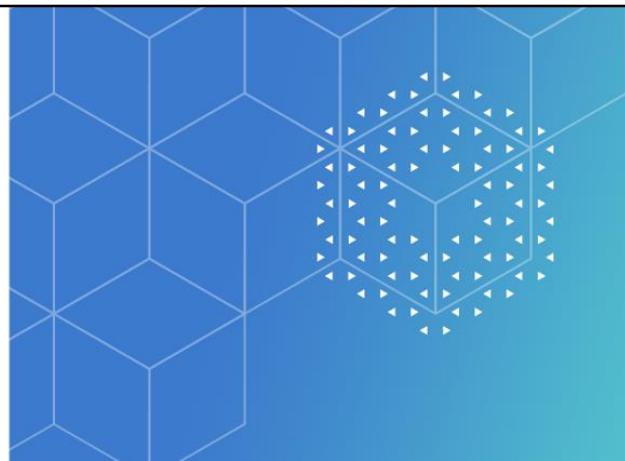
Quick Reference Guide: Verilog-2001 Feature List

- Enhanced file I/O
- Signed arithmetic support
- Comma-separated sensitivity lists
- Combinational sensitivity list (i.e., `@*`)
- ANSI-C-style module declarations
- ANSI-C-style task and function declarations
- Re-entrant tasks and functions
- PLI access to re-entrant tasks
- Combined port/net declarations
- ``ifndef`, ``elsif` and ``undef` directives
- Enhanced VCD syntax
- Named parameter association
- Reg declaration initial value assignments
- `$value$plusargs`
- ``default_netttype none` (disabling implicit net declarations)
- Power (exponentiation) operator
- Localparam
- Implicit net declarations with continuous assigns
- PLI system task extensions
- Automatic width extension of x and z constants beyond 32 bits
- Verilog configuration capability
- Generates
- Attributes
- ``line` directive
- Multidimensional arrays of variables and nets
- Indexed part selects and part selects within multidimensional arrays
- Indexed (variable) part selects
- Arrays of reals and events
- Sized/typed parameters
- `$nochange`
- VPI enhancements
- Skew timing checks of section 15 in
- Fork-join in re-entrant tasks/functions
- Automatic variables in event controls

For more information, see the *IEEE 1364 Language Reference Manual*.



This page does not contain notes.



SystemVerilog Event Scheduler

Appendix

B

Revision

1.0

Version

21.10

Estimated Time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Appendix Objective

In this appendix, you

- Analyze Event Scheduler

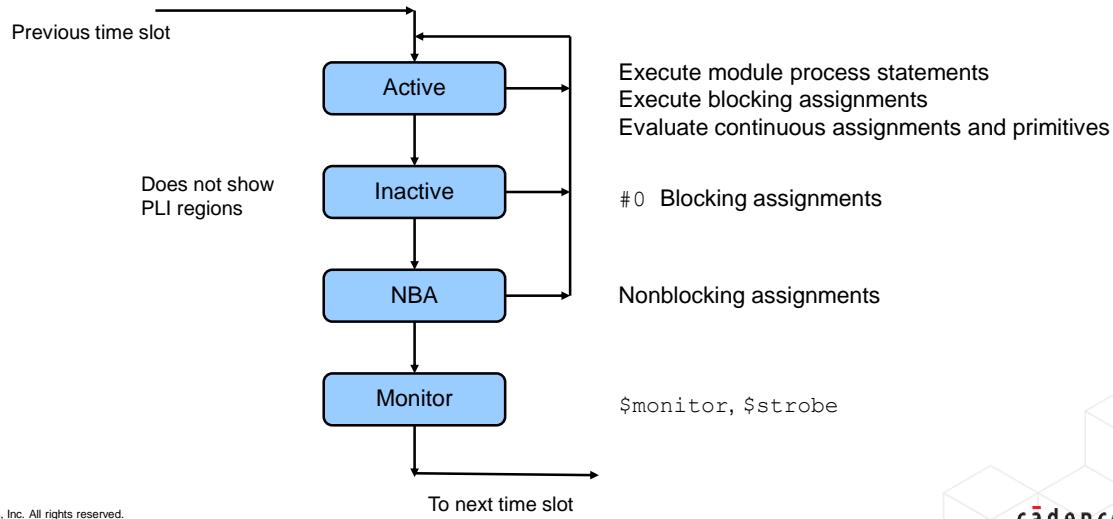


This page does not contain notes.

Simplified Verilog Event Scheduler



A compliant Verilog/SystemVerilog simulator will maintain some form of data structure that allows events to be dynamically scheduled, executed, and removed as the simulator advances through time. The data structure is normally implemented as a time-ordered set of linked lists, which are divided and subdivided in a well-defined manner.



432 © Cadence Design Systems, Inc. All rights reserved.



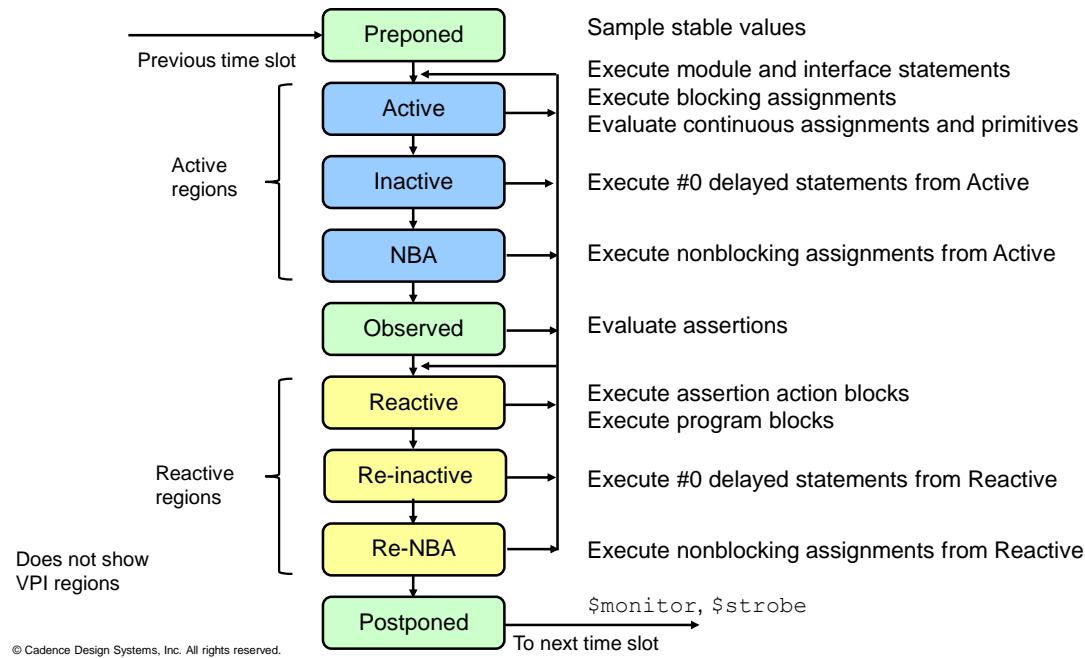
A simulation timeslot is divided into ordered regions to provide a predictable interaction between design constructs.

The Verilog event schedule has four regions for each simulation time:

- The Active region is for executing process statements.
- The Inactive region is for executing process statements postponed with a zero (#0) procedural delay.
- The NBA region is for updating nonblocking assignments.
- The Monitor region is for executing \$monitor and \$strobe and for calling user routines registered for execution during this read-only region. You cannot create additional events within this region.

The first three of these regions are iterative – they can schedule events that require return to the Active region. When no more events exist for the current simulation time, the simulator executes Monitor statements and then advances simulation time to the next time for which events are scheduled. The simulation terminates when no such future events exist.

Simplified SystemVerilog Event Scheduler



SystemVerilog adds regions to provide a predictable interaction between assertions, design code and testbench code.

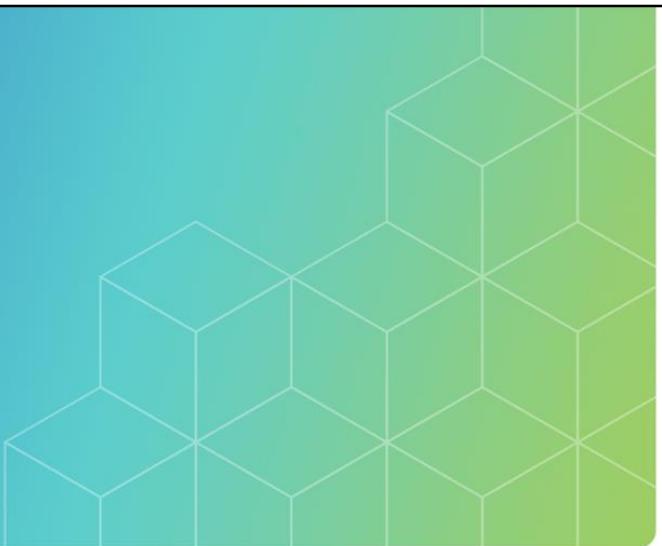
The SystemVerilog event schedule has 13 regions for each simulation time. Some regions used only by the VPI are not shown on the slide and not described here:

- The Preponed region is for sampling signal values before anything in the time slice changes their values. It is equivalent to the #1step sampling delay used in clocking blocks.
- The Active region is for executing interface and module statements (same as Verilog).
- The Inactive region is for executing statements postponed with zero (#0) procedural delay from the Active region (same as Verilog).
- The NBA region is for updating nonblocking assignments from the Active region (same as Verilog).
- The Observed region is for assertion evaluation.
- The Reactive region is for executing program block statements, blocking delay assignment in program blocks and assertion action blocks.
- The Re-inactive region is for executing statements postponed with a zero (#0) procedural delay from the Reactive region.
- The Re-NBA region is for updating nonblocking assignments from the Reactive region.
- The Postponed region is for system tasks that record signal values at the end of the time slice.

The Active through Re-inactive regions are iterative. Most of these regions can schedule events that require return to the Active region.



Programs



Appendix

C

Revision

1.0

Version

21.10

Estimated Time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Appendix Objective

In this appendix, you

- Identify the features of programs for verification code



This page does not contain notes.

Programs



IEEE 1800-2012 24

- A program is very similar to a module, but intended for testbench code.
 - Allows separation of design and test.
 - Is executed in a separate time region to avoid design/test races.
 - Programs execute in the Reactive region.
 - Modules execute in the Active region.
- Program blocks have special features and restrictions for testbench use.
- In particular, a program cannot instantiate hierarchy.
 - Programs are leaf elements.
 - Must be instantiated in a module.

```
program memtest (
    output wire [7:0] data;
    output bit [4:0] addr;
    output bit read, write);
    ...
initial begin
    ...
end
endprogram : memtest
```

```
module top;
    wire [7:0] data;
    wire [4:0] address;
    wire       read, write;

    memtest test (*.);
    memory mem8x32 (*.);

endmodule : top
```

436 © Cadence Design Systems, Inc. All rights reserved.



A program is a design element similar to a module, interface or program. It is usually declared in a separate file, compiled separately and then instantiated in a module or interface.

A program is intended to define testbench code. In a verification methodology using programs, all testbench code would be contained in programs, and all design (synthesizable) code in modules.

The key advantage of a program is that it is executed in a different time region than modules or interfaces. Programs are executed in the Reactive region (see Appendix B – SystemVerilog Event Scheduler). Modules and interfaces are executed in the Active region. By using programs only for verification code and modules or interfaces only for design code, race conditions between design and testbench can be reduced. For example, if design inputs are assigned from multiple program blocks, then all the inputs will be updated before the design is executed.

To enforce and help with testbench definition, program blocks have special restrictions and features.

In particular a program cannot instantiate any hierarchy – it is a leaf element. This means a program cannot contain declarations or instantiations of modules, interfaces, other programs, UDPs or primitives. Therefore, a program must be instantiated in a module, typically in parallel with the top module of the DUT.

Program Construct Restrictions

Allowed:

- Local data declarations
 - Variables
 - User-defined types
 - Classes
- initial and final blocks
- generate blocks
- Task and function declarations
 - Only visible in program block
- Continuous assignments
- Clocking blocks
- Concurrent assertions
- Functional coverage groups (covergroups)

Not Allowed:

- always blocks
- Declaration or instantiation of:
 - Interface
 - Module
 - Primitive
 - Program
- Anything specific to modules
 - Parameter overrides (`defparams`)
 - `specify` blocks
 - `specparam` declarations

437 © Cadence Design Systems, Inc. All rights reserved.



This table lists most of the restrictions on specific construct use in programs. Constructs which are not allowed will generate compile-time errors.

As a general rule, constructs that clearly represent design rather than verification are not allowed.

Note that concurrent assertions can be declared and executed in programs. However, assertions have a clearly defined scheduling in that they are sampled in Preponed, evaluated in Observed and their action blocks executed in Reactive. Therefore, they do not use program scheduling of execution in the Reactive region.

Example: Program Access Restrictions

- Only programs can access these program declarations:
 - Signals (including ports)
 - Subroutines
- Hierarchical references from one program to another are legal.
- A program can access any design declarations, including:
 - Signals
 - Subroutines
 - Executed in reactive region

```
program test1 (output reg t1);
  bit [3:0] addr;
  ...
endprogram : test1
```

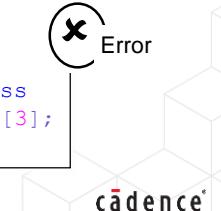
```
program test2 (output reg t2);
  wire [3:0] save;
  // good program access
  assign save = top.t1.addr;
  ...
endprogram : test2
```

```
module top;
  wire top_a, temp_a;
  dut d1 (top_a);
  test1 t1 (top_a);
  test2 t2 (top_a);

  // illegal program access
  assign temp_a = t1.addr[3];
endmodule: top
```

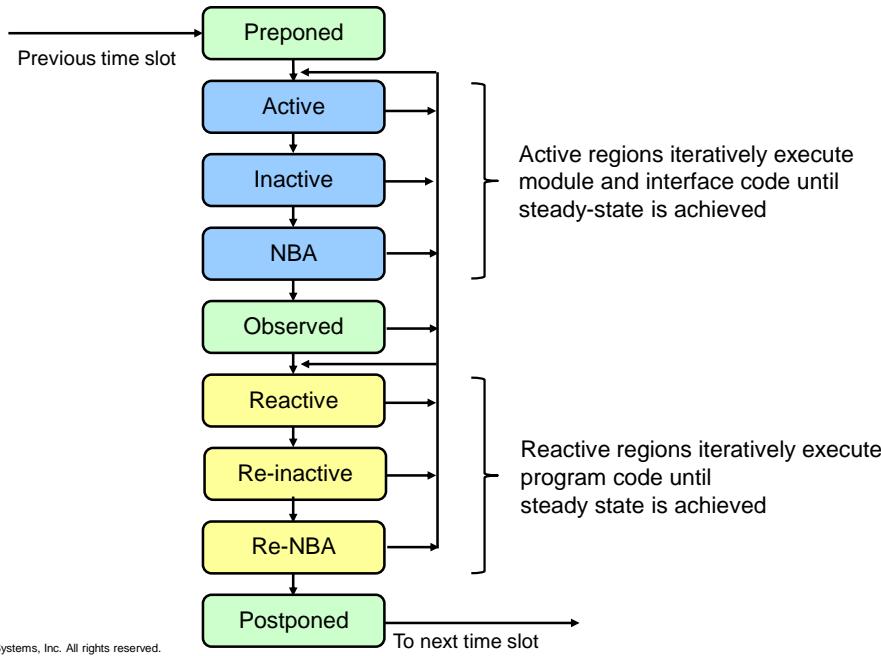


Error



A SystemVerilog philosophy is that the testbench should know about the design, but the design should not be aware of the testbench. Therefore, only programs can access variables or nets declared in a program (collectively called program signals). Likewise, only programs can call subroutines declared in programs. Programs can access signals or subroutines declared in other programs via hierarchical references. Programs can access signals or subroutines anywhere in the design via hierarchical references. When a program calls a subroutine defined in a module or interface, it is executed according to the program scheduling – i.e., in the Reactive region.

Program Block Scheduling



439 © Cadence Design Systems, Inc. All rights reserved.



The active region is for executing design code defined in modules and interfaces:

- The Active region is for executing interface and module statements.
- The Inactive region is for executing statements postponed with zero (#0) procedural delay from the Active region.
- The NBA region is for updating nonblocking assignments from the Active region.

The Reactive region is for executing testbench code defined in programs:

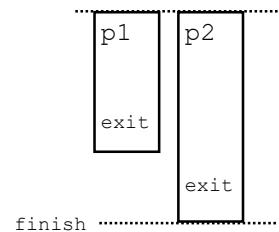
- The Reactive region is for executing program block statements, blocking delay assignment in program blocks and assertion action blocks.
- The Re-inactive region is for executing statements postponed with a zero (#0) procedural delay from the Reactive region.
- The Re-NBA region is for updating nonblocking assignments from the Reactive region.

The Active and Reactive regions are iterative. Most of these regions can schedule events that require return to the Active region.

Program Block: \$exit System Task

- You can call \$exit only from a program.
- Terminates all processes in the current program.
 - Initial blocks and their subprocesses.
 - Not continuous assignments.
 - Not tasks called from other programs.
- Also called implicitly after all initial blocks in a program are complete.
 - In other words, they have executed their last statement.
- \$finish is called when all program blocks have exited.
 - Either explicitly through \$exit or normally (implicit \$exit).

```
program test1;
  ...
initial begin
  ...
  if (timeout_error)
    $exit();
  ...
endprogram : test1
```



A SystemVerilog program can call \$exit to immediately terminate its initial blocks and their child processes and remove any pending assignments from the update queue. The \$exit does not apply to continuous assignments and does not apply to subroutines declared within the program but invoked from other programs. Only a program can call \$exit.

The \$exit system task differs from the \$stop and \$finish system tasks, in that \$exit immediately applies only to the program that starts it. The simulation terminates when all programs have terminated, either naturally or through \$exit calls.

Implicitly Instantiated/Anonymous Programs

- Programs can be nested in modules/interfaces.
- Nested programs without ports need not be instantiated.
 - Each is implicitly instantiated once.
 - Instance name is the same as definition name.
- Top-level programs can also be implicitly instantiated.
- Anonymous programs are unnamed.
 - Declared in compilation unit scope or package only.
 - Contain declarations only (a restricted subset) – no processes.
 - Declarations follow program rules.

```
module test(...);
  program p1;
    ...
  endprogram
  program p2;
    ...
  endprogram

  ...
endmodule : top
```

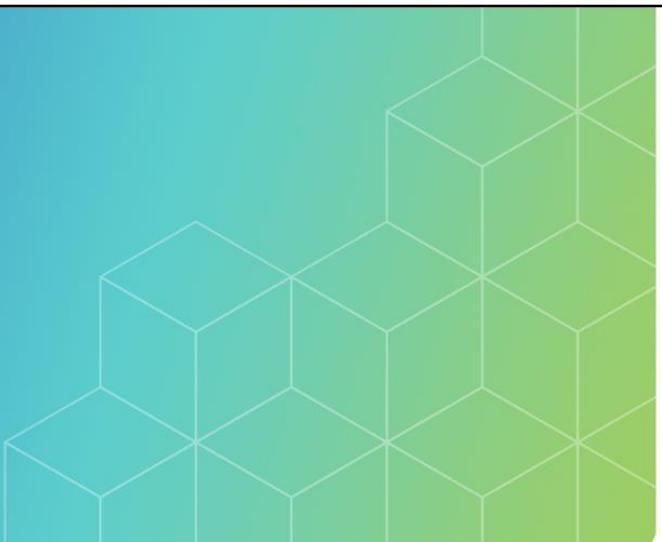
```
package program_decls;
  program;
    task do_stuff;
    ...
  endtask
endprogram
endpackage
```

441 © Cadence Design Systems, Inc. All rights reserved.



A program can be nested inside a module or interface declaration. If the nested program has no port connections, then you do not need to directly instantiate the program. If you do not explicitly instantiate the program, then it will be implicitly instantiated once with an instantiation name matching the program declaration name. Likewise, a top-level program is also implicitly instantiated if there is no explicit instance declared.

You can define anonymous programs in packages and in compilation unit scopes. Anonymous programs can declare a restricted set of program items without declaring a new scope, i.e., the declarations share the same name space as the package or compilation unit scope in which the program. Declarations in an anonymous program cannot be referenced from other programs via a hierarchical pathname.



Miscellaneous Features

Appendix

Revision

D

Version

1.0

21.10

Estimated Time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Appendix Objectives

In this appendix, you

- Analyze miscellaneous SystemVerilog constructs, including
 - final blocks
 - Unions
 - Fine-grain process control
 - \$root
 - Nested modules
 - \$urandom and \$urandom_range
 - randsequence

443 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Final Procedures: final



IEEE 1800-2012 9.2.3

- Procedural blocks that execute once at the end of simulation:
 - After explicit or implicit call to \$finish.
 - Cannot invoke scheduler (no scheduled assignments or delays).
 - Can be used to calculate and display simulation statistics.

```
final begin
    if (timeout_error)
        $display ("ERROR: %0t: Test Timed Out", $time);
    else
        $display ("INFO: %0t: Test Complete", $time);
        $display("Error Count: %d", error_count);
        $display("Fifo Overflow Count: %d", fifo_overflow);
end
```

444 © Cadence Design Systems, Inc. All rights reserved.



A final block cannot consume simulation time, but is otherwise similar to an initial block. You can place a final block anywhere you can place an initial block. SystemVerilog executes all final blocks exactly once at the end of simulation in an unspecified but repeatable order.

SystemVerilog executes the final blocks upon encountering \$finish or upon running out of events to process. Any final block that itself calls \$finish or calls a user-defined system task or function that terminates the simulation aborts execution of the final blocks. After SystemVerilog completes or terminates execution of the final blocks, it calls any PLI routines you have scheduled for execution at the end of the simulation.

Unions



IEEE 1800-2012 7.3

```
union {
    logic [31:0] udata; // unsigned data
    integer sdata;     // signed data
} inbus, outbus;      // 32-bit variables

initial begin
    inbus.udata = 32'h00f3;
    outbus.sdata = -44;
    ...
}
```



Fields of a packed union must be all the same size.

- You use a union to store different data types at the same memory location.
- The union size is the size of its largest field.

```
typedef union { int i;
                shortreal f; } num; // named union type

num n1, n2; // instances of union type

initial begin
    n1.f = 0.0; // set n1 in floating point format
    n2.i = -44; // set n2 in integer format
    ...
}
```



445 © Cadence Design Systems, Inc. All rights reserved.

The union construct provides a way to view the same memory location as different data types without the need to cast the data types. The SystemVerilog union is identical to the C union, with two exceptions:

1. You cannot provide a tag name for a SystemVerilog structure or union. You can alternatively use the `typedef` keyword to declare a type name for the structure or union.
2. You can further qualify the SystemVerilog union as “tagged”. A SystemVerilog tagged union is type-checked. The next slide has an example of a tagged union.

Union Example

Structures can contain unions and unions can contain structures.

```
typedef struct { union { int i;
                           shortreal f;
                           } n; // anonymous union type
               bit isfloat;
             } tagged_s; // named struct type

tagged_s a[9:0]; //array of structure containing union
```

```
typedef union tagged { int i;
                       shortreal f;
                     } tagged_u;
tagged_u value;
int j;
shortreal g;

initial begin
  value = tagged f 3.14156;
  g = value.f; // legal
  j = value.i; // illegal
```

Union variable
tagged as field f

Access to other fields
illegal without
retagging variable

446 © Cadence Design Systems, Inc. All rights reserved.



The top example declares an array of a named structure type that contains a field that is a union. The other structure field is a bit that indicates whether the structure union field is currently being used as an int type or as a float type.

SystemVerilog provides the tagged union qualifier to serve exactly that purpose. A tagged union stores both its current value and its current tag. Subsequent expressions accessing the tagged value must be of the current tag type. This type checking is generally done during runtime.

Fine-Grain Process Control



Fine-Grain Process Control is a method where, a built-in process class allows, one process to access and control another process, once it has started.

- Every static and dynamic thread creates an instance of the class `process`.

```
class process;
  enum state { FINISHED, RUNNING, WAITING, SUSPENDED, KILLED };
  ...

```

- This gives access to fine-grain process control and query methods:

Method	Description
<code>static function process self()</code>	Returns handle of calling process
<code>function state status()</code>	Returns state of process
<code>task await()</code>	Waits for a process to terminate. Note: Illegal to wait for self
<code>function void suspend()</code>	Suspends a process
<code>function void resume()</code>	Resumes a suspended process
<code>function void kill()</code>	Forcibly terminates a process

```
process ptr;
initial begin
  ptr = process::self();
  if (ptr.status == process::SUSPENDED) ...

```

Get handle to
access members



447 © Cadence Design Systems, Inc. All rights reserved.

The `process` class is built in. SystemVerilog creates an instance of `process` for every process created, either statically, for *initial* and *always* blocks, or dynamically, for *fork* blocks. You obtain the handle of a process by calling its `self()` method in the block and assigning to a handle of the `process` type. By declaring that handle somewhere accessible to other processes, they can then manipulate the state of that process.

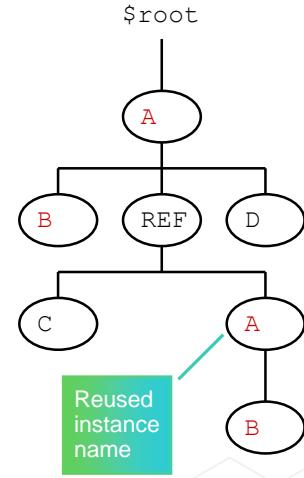
You cannot create your own instances of the `process` class. Nor can the `process` class be extended with sub-classes.

Instance Tree Root: \$root



\$root is used to unambiguously refer to a top-level instance or to an instance path starting from the root of the instantiation tree. \$root is the root of the instantiation tree.

- \$root is a reference to top-most design element in hierarchical paths.
- In Verilog, local paths take precedence:
 - From REF, relative path A.B is always absolute path A.REF.A.B.
- \$root can be used to start path references at topmost design element:
 - From anywhere, path \$root.A.B is always the absolute path A.B.
- Remember hierarchical paths can be used:
 - To read or write remote variables or events.
 - In task or functions calls.



448 © Cadence Design Systems, Inc. All rights reserved.

Verilog does not differentiate between an absolute hierarchical path name and a relative hierarchical path name. If the path names happen to be identical, Verilog assumes you mean the relative path. To avoid such confusion, most users name their top-level module something obvious like `top` and do not reuse that instance name lower in the hierarchy. For SystemVerilog, you can use “dollar root” (\$root) to refer to a virtual simulation root, thus unambiguously specifying an absolute path.

Module References



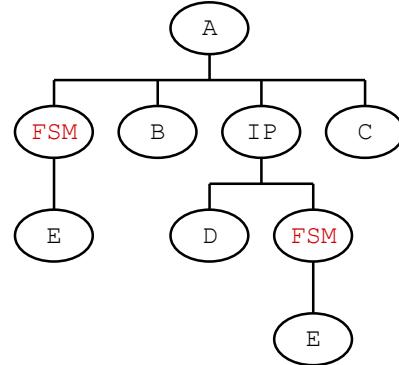
For Verilog, you need to declare your modules in the definitions name space. All module declarations are globally visible and can be instantiated in any other module.

Issues:

- Different modules with the same name cause naming conflicts:
 - E.g., two modules named `FSM`.
 - Verilog-2001 configurations can help.
- Module access cannot be restricted:
 - All modules of an IP block can be externally referenced.

Solution:

SystemVerilog adds nested module declarations (more on next page).



- Two different modules named `FSM`
- Using IP module `E` externally

449 © Cadence Design Systems, Inc. All rights reserved.

For Verilog, you need to declare your modules in the definitions name space. All module declarations are globally visible and can be instantiated in any other module. As a developer of intellectual property, you might want to prohibit a module of your design to be instantiated anywhere outside the IP hierarchy. Or you may want to simply use the same module name that someone else on your team has already used.

Nested Modules

- You can declare (nest) a module in a module declaration.
- Nested modules can further declare nested modules.
- Can degrade readability.
 - You might want to use `include.
- Declarations in a local module:
 - Override higher-level declarations of the same name.
 - Are visible only where declared and in lower (nested) module declarations.

```
module FSM (...);
...
endmodule : FSM

module IP(...);

D   m1 (...);
FSM m2 (...) ; // local FSM

module FSM (...);
...
endmodule : FSM

`include "dmod.v"

endmodule : IP
```

```
module D (...);
...
endmodule : D
```

dmod.v



450 © Cadence Design Systems, Inc. All rights reserved.

A nested module definition is visible only in the module where it is declared or in any sub-hierarchy of the module where it is declared. You can declare modules in the top level of your own design component, and thus restrict their use to just that component.

Such local declarations override declarations in the definitions name space. This allows the use of different module definitions of the same name.

Deeply nesting module definitions can make your code less readable. You may want to judiciously utilize include (`include) directives and place each module definition in a separate file.

The FSM module in the example may be instantiated before it is declared. This is consistent with such declarations in the definitions name space.

Unsigned Random Numbers Using \$urandom



IEEE 1800-2012 18.13.1

- \$urandom generates a 32-bit random unsigned integer.
- \$urandom_range generates a 32-bit random unsigned integer in a specified range.

```

int value;
bit [63:0] addr;

value = $urandom();           // 32 bit random number
value = $urandom(8);          // 32 bit random number with seed
addr = {$urandom,$urandom};   // 64 bit random number

value = $urandom_range(10,2); // 32 bit random number from 2 to 10
value = $urandom_range(16);   // 32 bit random number from 0 to 16
value = $urandom_range(1,9);  // 32 bit random number from 1 to 9

```

Ranges are automatically
re-ordered to (max, min)

451 © Cadence Design Systems, Inc. All rights reserved.



If all you want is a random integer value, you can use the SystemVerilog \$urandom() system function to obtain a random unsigned 32-bit integer, or \$urandom_range() system function to constrain the value to a range of values. These functions differ from the Verilog \$random() function in that:

1. They return unsigned values rather than signed values.
2. The random sequences are independent of each other, i.e., they use the SystemVerilog randomization implementation. For backward compatibility, the Verilog \$random still uses the Verilog randomization implementation where all randomization calls are part of the same single sequence.

```

function int unsigned $urandom [ (int seed) ] ;
function int unsigned $urandom_range(int unsigned maxval, int unsigned
minval=0) ;

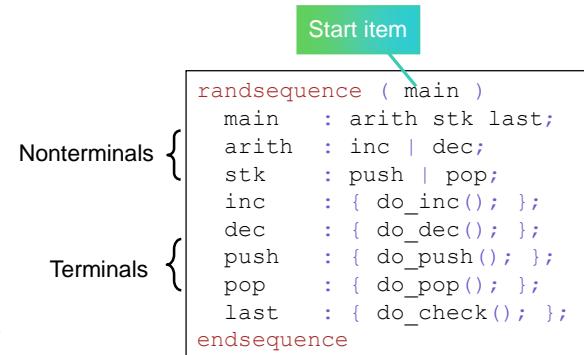
```

Random Sequence Generation: `randsequence`



IEEE 1800-2012 18.17

- You can use `randsequence` to execute a random sequence of statements.
- Based on rules called a production list.
- Production list items are either:
 - Terminal: an indivisible item that needs no more definition.
 - Nonterminal: Defined in terms of terminals and other nonterminals.
- All productions must ultimately decompose into terminal productions.



Start item

Valid Combinations

do_inc()	do_push()	do_check()
do_dec()	do_push()	do_check()
do_inc()	do_pop()	do_check()
do_dec()	do_pop()	do_check()



452 © Cadence Design Systems, Inc. All rights reserved.

You can use the `randsequence` keyword to execute a random sequence of statements based on rules called a production list. The argument to the `randsequence` is the identifier of the production to start with. This defaults to the first listed production.

Each production consists of a function declaration, with optional argument and return type declarations, followed by a colon, followed by rules for the production, where rules are separated by the bitwise OR (`|`) operator.

Each rule is a production list of one or more productions, and may have a weight associated with it, and may have a code block associated with it.

A code block is data declarations and statements between curly brackets (`{ }`).

Each `randsequence` construct creates its own automatic scope, which means that production declarations are not visible outside of the `randsequence` construct. Furthermore, each code block creates an anonymous automatic scope, so its declarations are also not visible outside the code block.

The sequence in this example starts with the `main` production, which defines one production rule consisting of the arithmetic (`arith`), stack (`stk`), and `last` productions, in sequence. The arithmetic production defines the increment (`inc`) and decrement (`dec`) alternative rules, each equally likely to be selected. The stack production defines the `push` and `pop` alternative rules, each also equally likely to be selected.

randsequence : Random Production Weights

You can apply weights to production items to influence randomization.

- The default weight is 1.
- Apply weights with the `:=` operator.
- Weights apply to alternatives separated with `|`.

```
randsequence ( main )
  main   : arith stk last;
  arith  : inc:=2 | dec:=1;
  stk    : push:=3 | pop;
  inc    : { do_inc(); };
  dec    : { do_dec(); };
  push   : { do_push(); };
  pop    : { do_pop(); };
  last   : { do_check(); };
endsequence
```

inc is twice as likely as dec

push is thrice as likely as pop



You can weight production alternatives. These are the relative weights of production rules within a production list, and do not weight the production list itself relative to any other production list.

Production Statements: case, if-else, repeat in randsequence

```
randsequence ( main )
    main   : repeat(5) arith stk last;
    arith  : if (incr)
              inc;
              else
              dec;
    stk    : case (which)
              0: push;
              1: pop;
              endcase;
    inc    : { do_inc(); };
    dec    : { do_dec(); };
    push   : { do_push(); };
    pop    : { do_pop(); };
    last   : { do_check(); };
endsequence
```

Very limited constructs in productions

Less limited constructs in code blocks



A production list can contain a restricted set of procedural constructs. It can contain the `case` and `if...else` constructs to make the production conditional, and can contain the `repeat` construct to repeat the production a number of times.

Aborting Productions: `break`, `return` in `randsequence`

You can use `break` and `return` in code blocks to abort productions:

- `break` terminates the entire `randsequence` production.
- `return` aborts the generation of current production only.

```
randsequence(test)
  test: run1 run2;
  run1: A B;
  run2: B C;
  A: { if(i==1)
        // exit item A
        return;
        $display("A"); };
  B: { if(i==2)
        // exit randsequence
        break;
        $display("B"); };
  C: { $display("C"); };
endsequence
```

i == 1	produces	B B C
i == 2	produces	A

455 © Cadence Design Systems, Inc. All rights reserved.



Your code blocks can use the `break` keyword and the `return` keyword to abort productions. These keywords have special semantics within a `randsequence` construct. Within a `randsequence` construct, the `break` keyword terminates the entire `randsequence` production, and not just any loop it happens to be in, and the `return` keyword terminates only the current production. As every production declaration is a local function declaration, and every production item is a task call, the code block of a production item can utilize those arguments and can return a value.

randsequence : Passing Values Between Productions

- A production definition is similar to a function definition.
- A production item is similar to a task call.
- You can access passed arguments throughout the production scope.
- You can access returned values after the production item (in the scope).

```

randsequence ( main )
    main      : arith($urandom) {$display(arith);} stk last;
    int arith(int i) : inc(i) {return inc;} |
                      dec(i) {return dec;};
    stk      : push | pop;
    int inc(int i) : { return do_inc(i); };
    int dec(int i) : { return do_dec(i); };
    push   : { do_push(); };
    pop    : { do_pop(); };
    last   : { do_check(); };
endsequence

```

456 © Cadence Design Systems, Inc. All rights reserved.



A production definition is similar to a function definition:

```
[function_data_type] production_identifier
  [(tf_port_list)]:rs_rule{|rs_rule};
```

A production item is similar to a task call:

```
production_identifier [ ( list_of_arguments ) ]
```

The syntax for declaring a production is similar to the syntax for declaring a function. However, a production is not a function; the production code block can consume time.

The syntax for passing data to a production is similar to a task call.

A production for which you have declared a type can return data. You return the data in the return statement. The data is available as the production item name after the production item “call” in the “calling” production.

This example passes a random number to the arithmetic production and displays the returned value. The arithmetic production further exchanges this value with either the increment or decrement production. The increment and decrement productions each call a function to implement their operation.

Event Sequencing: `wait_order`



IEEE 1800-2012 15.5.4

A process can wait for a specific sequence of events:

- Upon each sequential event, subsequent events must not yet have triggered.
- The first event can alternatively be the triggered property:
 - Has an associated action block.
- Construct is similar to a concurrent assertion in a procedural block.

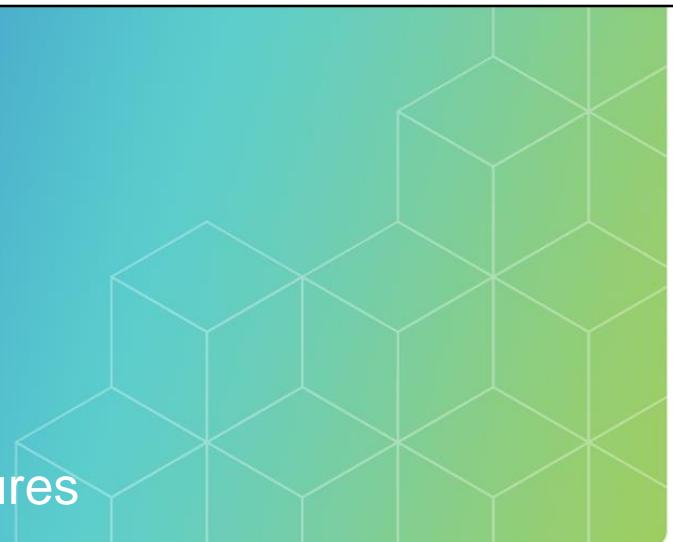
```
event e1, e2, e3;
always
  wait_order(e1.triggered, e2, e3)
    $display("events in order");
  else
    $warning("events out of order");
```

Equivalent

```
event e1, e2, e3;
always begin
  wait(e1.triggered);
  @(e2 or e3);
  if (!e3.triggered) begin
    @e3;
    $display("events in order");
  end
  else
    $display("events out of order");
end
```



A process can wait for a specific sequence of events. The first listed event can optionally be the triggered state of an event. The `wait_order` statement has characteristics very similar to a concurrent assertion. Occurrence in turn of each event implies that the next event in the list has not yet occurred and will occur before any of the later events in the list. Re-occurrence of an event has no additional effect. Events out of order constitute an error by default. You can change this in an associated action block.



SystemVerilog 2012 Features

Appendix

Revision

E

Version

1.0

21.10

Estimated Time:

- Lecture
- Lab

cadence®

This page does not contain notes.

Appendix Objectives

In this appendix, you

- Analyze key SV2012 features which are enhancements for addressing verification
 - Interface classes
 - Remove restrictions on NBA assignments to class members
 - Soft constraints
 - Constraint to generate unique elements



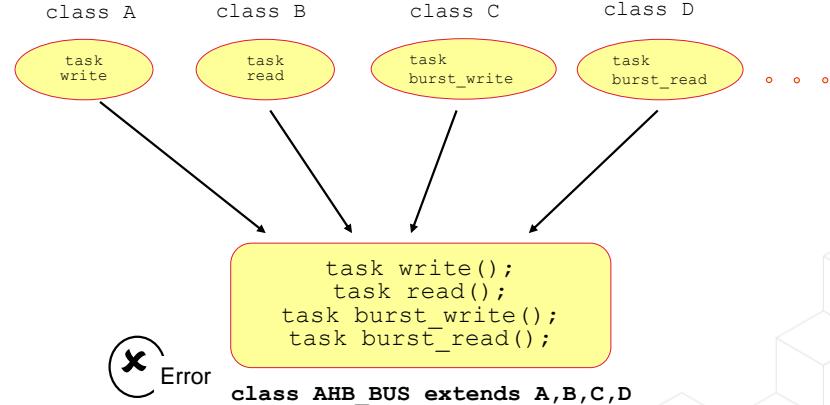
This page does not contain notes.

SystemVerilog Single Inheritance



- SystemVerilog 2009 allows only a single inheritance:
 - A class can only extend from a single class.
- This makes some inheritance forms complicated.

- **Problem:** If we want to utilize methods from multiple classes, it is not possible with a single inheritance.
- **Solution:** Interface classes (see next slide).



460 © Cadence Design Systems, Inc. All rights reserved.



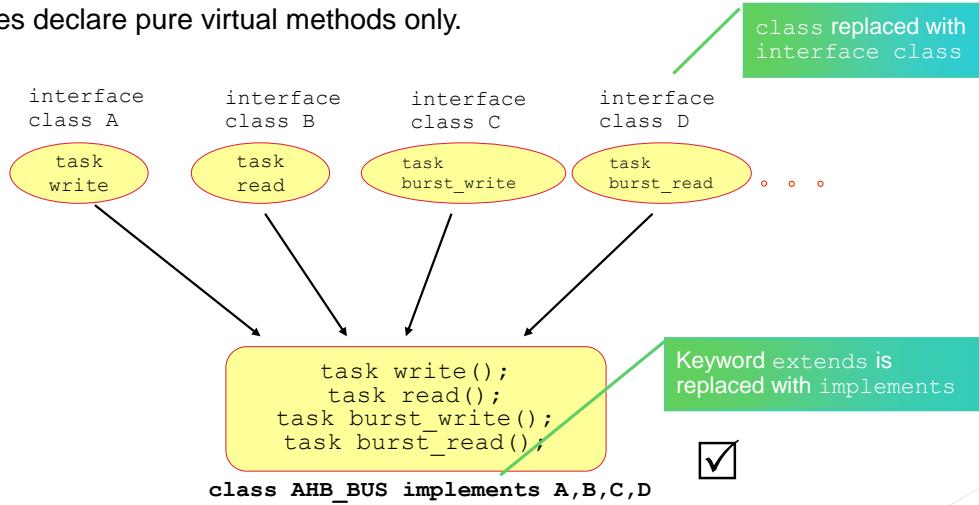
This page does not contain notes.

SystemVerilog Interface Classes

SystemVerilog 2012 allows a form of multiple inheritance:

- Through Java-like interface classes.

Interface classes declare pure virtual methods only.



461 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Interface Classes



IEEE 1800-2012 8.26

SystemVerilog 2012 allows a class to implement one or more interface classes:

- Enables code reuse.
- A non-interface class can implement combined functionality of all interface classes.

```
class child implements base1, base2;
    virtual function void iambase1();
        $display("This is base1");
    endfunction
    virtual function void iambase2();
        $display("This is base2");
    endfunction
endclass
```

interface classes may declare pure virtual prototypes

```
interface class base1;
    pure virtual function void iambase1();
endclass
```

```
interface class base2;
    pure virtual function void iambase2();
endclass
```

Class should implement pure virtual prototypes from base1, base2

462 © Cadence Design Systems, Inc. All rights reserved.



Code reuse – code that is already written can be better utilized.

A list of interface classes with unique functionalities can be created and a non-interface class can implement combined functionality of all these interface classes.

Interface Class – Rules

- An interface class can inherit from one or more interface classes with the keyword `extends`.
- An interface class does not provide implementation for its prototypes.
- An interface class can contain only:
 - Pure virtual methods.
 - Type declarations.
 - Parameter declarations.
- Classes which extend or implement the interface class can only reference the virtual methods:
 - Type and parameter declarations are visible only through the method declarations.
 - Cannot be referenced directly.
- An interface class:
 - Cannot be declared inside of another class (nested class).
 - Shall not be nested within another class.



This page does not contain notes.

Interface Class – Type Access

- Extending an interface class allows parameters and typedefs to be inherited.
- Implementation does not allow visibility of parameters and typedefts by default.
 - However, as static properties they can be accessed via a scope resolution operator.

```
class imp implements ext;
  logic [WIDTH-1:0] y2 = 55;
  logic [ex::WIDTH-1:0] y2 = 55;

  virtual function return_ex();
    ...
  endfunction

  virtual function bit[ex::WIDTH-1:0] return_ex1();
    ...
  endfunction
endclass
```

```
interface class base #(type t = logic);
  parameter WIDTH = 32;
  pure virtual function return_ex();
endclass
```

Interface class with one parameter and one prototype

```
interface class ext extends base #(bit);
  pure virtual function bit[WIDTH-1:0] return_ex1();
endclass
```

Parameter WIDTH is inherited since this is extended from another interface class

Compilation error since implementation will not inherit parameters/typedefts

Interface class parameter is static and can be accessed in implementation class with ::



- Parameters and typedefs within an interface class are inherited by extending interface class and not by implementing them.
- Parameters and typedefs defined in an interface class are static.
- They can be accessed by the scope resolution operator but not by a dotted reference.

Interface Class Inheritance and Implementation

An interface class can extend zero or more interface classes but:

- May not implement an interface class.
- May not extend a class or virtual class.
- May not implement a class or virtual class.

```
interface class ifbase;  
  ...  
endclass
```

```
class cbase;  
  ...  
endclass
```

```
interface class base3 implements ifbase;  
  ...  
endclass
```



Error

```
interface class base4 extends cbase;  
  ...  
endclass
```



Error

```
interface class base5 implements cbase;  
  ...  
endclass
```



Error



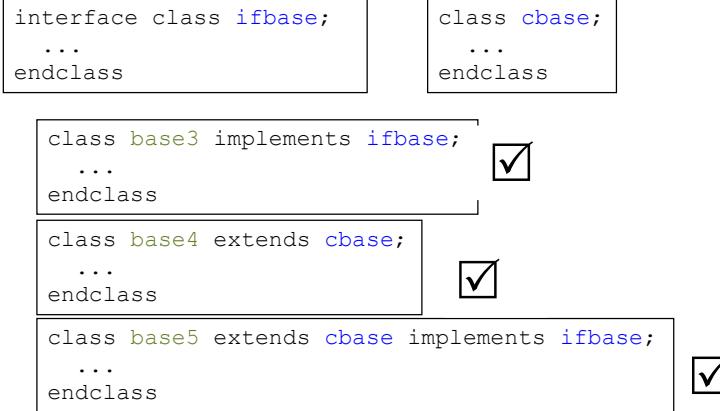
extends: Mechanism to add to or modify the behavior of a superclass.

implements: Requirement to provide implementations for the pure virtual methods in an interface class.

Interface Class Inheritance and Implementation (continued)

A class or virtual class can only:

- Implement zero or more interface classes.
- Extend at most one other class or a virtual class.
- May simultaneously extend a class and implement interface classes.



466 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Interface Class Inheritance and Implementation (continued)

A class cannot:

- Extend an interface class.
- Implement a class or virtual class.

```
interface class ifbase;  
  ...  
endclass
```

```
class cbase;  
  ...  
endclass
```

```
class base3 extends ifbase;  
  ...  
endclass
```



Error

```
class base4 implements cbase;  
  ...  
endclass
```



Error



This page does not contain notes.

Hard Constraints



IEEE 1800-2012 18.5

- All SystemVerilog 2009 constraints are hard:
 - All constraints must be met in randomization.
- Probable ways of resolving conflicts are given below:
 - Disabling default constraints.
 - Using soft constraints.
- Soft constraints can be sacrificed to reach a solution.

```
module base;

class hard_con;
  rand bit [4:0] var1;
  constraint c1 {var1<4;};
endclass

hard_con hc1 = new();

initial
  if(!(hc1.randomize() with {var1 == 5;}))
    $error("Randomization Error");

endmodule
```

Hard constraint

Inline constraint conflicting with declarative constraint

Constraint violation on randomization

```
if(!(hc1.randomize() with {var1 == 5;}))
|
xmsim: *W,SVRNDF (../hardsoft.sv,11|21): The randomize method call failed.
xmsim: *W,RNDOCS: These constraints contribute to the set of conflicting constraints:

constraint c1 {var1<4;} (../hardsoft.sv,5)
if(!(hc1.randomize() with {var1 == 5;})) (../hardsoft.sv,11)...
```

468 © Cadence Design Systems, Inc. All rights reserved.

Constraints which are used till SV2009 are called as hard constraints except **distribution constraints**.

One way to overcome this is by disabling default constraints before overriding them with conflicting constraints.

SystemVerilog 2012 Soft Constraints



IEEE 1800-2012 18.5.14

- Constraints can be made soft using the keyword `soft`
- Order of declaration of soft constraints is important.
 - Prioritized in order of declaration.
 - See the next slide.

```
module base;  
  
    class soft_con;  
        rand bit [4:0] var1;  
        constraint c1 {soft var1<4;}  
    endclass  
  
    soft_con scl = new();  
  
    initial  
        if(!(scl.randomize() with {var1 == 5;}))  
            $error("Randomization Error");  
  
endmodule
```

soft keyword identifies constraint as soft

Inline constraint conflicting with declarative constraint

No constraint violation – soft constraint is sacrificed

RESULT:
Value of var1 is 5

469 © Cadence Design Systems, Inc. All rights reserved.



The priorities of soft constraints are defined such that the last constraint specified prevails.

Consequently, constraints specified in subsequent layers in a verification environment are assigned higher priorities than those in preceding layers.

Soft Constraint Priority Rules

The following rules determine the priorities of soft constraints:

- Soft constraints can only be specified for `rand` variables (not `randc`).
- Constraint expressions that appear later in the construct have higher priority.
- Constraints in derived classes have a higher priority than all constraints in their super classes.

```

class base1;
  rand bit [4:0] var1;
  constraint c1 { soft var1 < 4; }
endclass

class base2 extends base1;
  constraint c2 { soft var1 > 6; }
endclass

base2 b2 = new();
initial begin
  if(!(b2.randomize() with {soft var1 == 5;}))
    $error("Randomization Error");
end

```

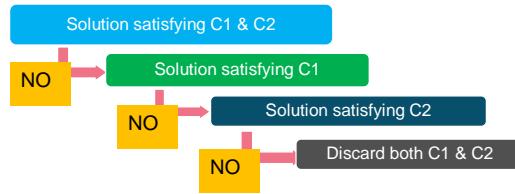
470 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Soft Constraints and the Constraint Solver

- Consider two soft constraints c1 and c2, such that c1 has higher priority than c2. In such a case, the constraint solver provides the following:



- The constraint solver always satisfies the following properties:
 - Randomize call involving soft constraints will never fail.
 - Soft constraints without any contradictions exhibit same result as if all constraints were declared as hard constraints.

This page does not contain notes.

Uniqueness Constraints



IEEE 1800-2012 18.5.5

- A group of variables or an array can be constrained to generate unique values.
- Prefix the `unique` keyword used in the constraint block.
- A group of variables can be either of the following:
 - Scalar variable of integral type.
 - Unpacked array variable with leaf element type as integral.

```
class base1;
    rand bit [1:0] a,b,c,d;
    rand bit [1:0] list[5];

    constraint c1 {unique {b, c, list[1], list[4]};}
endclass
```

Rule 1: Using `unique` as keyword

Rule 2: Having group of variables with same integral type

472 © Cadence Design Systems, Inc. All rights reserved.



The syntax of the `unique` keyword is:

`unique <open_range_list>`

where `open_range_list` is a comma-separated list of the variables to be constrained consisting of one of the following:

- A scalar variable of integral type.
- An unpacked array variable whose leaf element type is integral or a slice of such a variable.

Limitations of Uniqueness Constraints

- The group must all be equivalent types.
- No `randc` variables are allowed in the group.
- If `open_range_list` contains fewer than two members, it is ignored.

```
class base1;
    rand bit [1:0] a,b,c,d;
    rand bit [1:0] list[5];
    randc bit [1:0] var;

    constraint c1 {unique {b, c, list[1], list[4]};}

    constraint c2 {unique {d};}
endclass
```

`randc` variables not used in the constraint group

List having group of variables with equivalent types

This constraint is ignored since group has only a single element

This page does not contain notes.

Uniqueness Constraints: Example 1

```
class base1;
    rand bit [3:0] a,b,c,d;
    rand bit [3:0] list[5];

    constraint c1 {unique {b, c, list[1], list[4]};}
endclass

base1 b1 = new();

initial
repeat(1)
    if(!b1.randomize()) $display("Randomization error");
```

Example showing constraint group members with same equivalent types

RESULT:

Values of b=f, c=8, list[1]=9, list[4]=f

Simulation result showing unique values generated for the group

This page does not contain notes.

Uniqueness Constraints: Example 2

```
class base1;
    rand bit [1:0] a,b,c,d,e;

    constraint c1 {unique {a,b,c,d,e};}
endclass

base1 b1 = new();

initial begin
    if(!b1.randomize()) $error("Randomization error");
    ...

```

unique.sv

Group members where
unique values is not possible
(5 members, only 4 values)

Randomization failure: generation of
unique values not possible for this group

```
if(!b1.randomize()) $error("Randomization error");
    |

```

```
xmsim: *W,SVRNDF (../unique.sv,12|17): The randomize method call failed.
```

```
xmsim: *W,RNDOCS: These constraints contribute to the set of conflicting constraints:
```

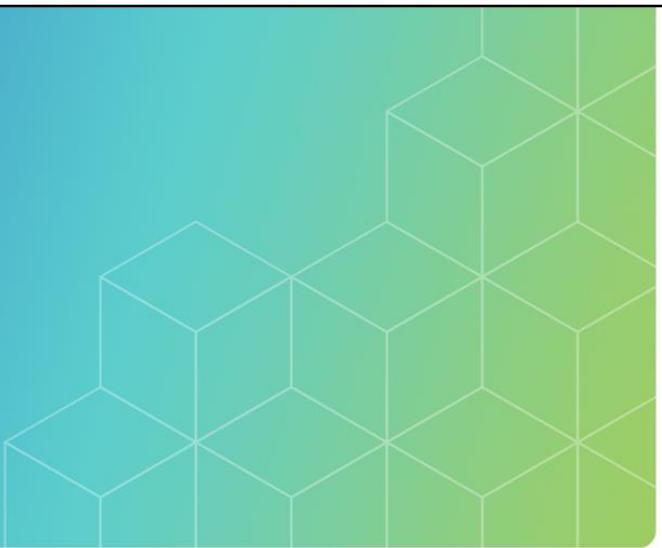
```
constraint c1 {unique {a,b,c,d,e};} (../unique.sv,6)
```



This page does not contain notes.



DPI Reference



Appendix

F

Revision

1.0

Version

21.10

Estimated Time:

- Lecture
- Lab

cadence®

This page does not contain notes.



Quick Reference Guide: Aspects of Task and Function Import

- An import is a declaration – it can be placed anywhere a declaration is allowed.
- For imported functions, the following “small” return types are supported:
 - void, byte, shortint, int, longint, real, shortreal, string, chandle, scalar values of bit or logic.
- In addition to the above, imports support the following formal argument types:
 - Packed arrays/structures/unions of bit or logic.
 - Enumerated types (as their base type only, without value names).
 - Further constructs of supported types:
 - Including unpacked arrays, packed or unpacked structures, packed unions, typedefs of all the above.
 - A C function imported as a task must return the void type.
 - **“Small” type input arguments are passed by value, others by reference.**



You can place import declarations in the compilation unit scope, packages, interfaces, modules or programs. Placing it in a package simplifies sharing it among multiple design and test units.

You can import any C function. It is common to import functions from the C math library.

SystemVerilog returns the function results by value, so restricts the function return types to the “small” types. It passes small arguments by value and large arguments by reference, which on the C side means by pointer.

C functions imported as a SystemVerilog task must return the void type.

You can give a C function imported as a SystemVerilog task or function the context property. You can give a C function imported as a SystemVerilog function the pure property. Following slides describe these properties.

Quick Reference Guide: Aspects of Task and Function Export



- An export is a declaration, and you can place it anywhere a declaration is allowed.
- Export declarations:
 - Can export only those subroutines defined in their own scope.
 - Can export a subroutine no more than once.
 - Cannot export class member methods.
- Exported subroutines:
 - Accept the same argument data types as imported subroutines.
 - Are always context (aware of their scope).
 - Can be called by imported context subroutines.
 - Functions cannot call tasks.
 - Exported tasks can consume time.
 - This is how you create a time delay in the C layer.

478 © Cadence Design Systems, Inc. All rights reserved.



You can place export declarations in the compilation unit scope, packages, interfaces, modules or programs. Placing it in a package simplifies sharing it among multiple design and test units.

An export declaration can export only the subroutines declared in its own scope, and can export a subroutine at most once. An export declaration is illegal within a class type declaration and so cannot export a class method.

An imported subroutine declared as `context` can call an exported subroutine. This is how you create a time layer in C. The time is actually consumed by the exported SystemVerilog task called by an imported task. A SystemVerilog function cannot of course call a SystemVerilog task regardless of what language actually implements the task.



Context Tasks and Functions

```
import "DPI-C" context function int parityf (input int a);
import "DPI-C" context task parityf (input int a, output int b);
```

- Some imported subroutines may need to know the scope (context) of their call:
 - For visibility of variables, etc., in the local scope.
- To reduce overhead, context is not provided by default:
 - Declaring an imported subroutine as `context` provides this information.
 - Specifically the scope of the import *declaration*.
- Imported subroutines must be declared as context if they:
 - Call exported SystemVerilog subroutines.
 - Access any SystemVerilog objects other than through their arguments.
 - For example – through PLI or VPI routines.
- A noncontext imported subroutine cannot access any SystemVerilog data other than its actual arguments.



```
import "DPI-C" context [ c_identifier = ] prototype;
```

An imported subroutine may need to know the scope (or context) of its call. In SystemVerilog code, this is usually to access variables or declarations local to the subroutine declaration. In SystemVerilog, a subroutine is always executed in the scope of its declaration, regardless of where the call is made. For example, calling a task `f (y)` declared in the scope `U1.M1` via a hierarchical pathname (`U1.M1.f (y)`) will execute the task in the context of the declaration `U1.M1`, not in the context of the calling scope.

Declaring an imported subroutine as `context` makes the imported task or function aware of the scope of its declaration. Specifically, the call is instrumented with additional information which identifies the scope of the declaration.

An imported subroutine *must* be declared as context if it contains a call back into the SystemVerilog layer through either:

- A call to an exported SystemVerilog subroutine, or
- A call to a PLI or VPI routine

This means a noncontext subroutine cannot access any SystemVerilog data other than the actual arguments which are passed to the call.

A context subroutine can also access objects in other scopes by changing its context. The `svdpi.h` header defines the `svSetScope ()` routine for setting the context.

Pure Functions



IEEE 1800-2012 35.5.2

```
import "DPI-C" pure function int parityf (input int a);
```

- A pure function call can be optimized or eliminated by the simulator.
 - For example – by reusing a value from a previous call with identical arguments.
- Imported functions can optionally be declared as `pure` only if:
 - They return a nonvoid value.
 - They have at least one input and no output or inout arguments.
 - The result depends solely on the values of the input arguments.
 - They do not directly or indirectly:
 - Perform any file operations.
 - Access any persistent data (e.g., static variables).
 - Read or write anything beyond its arguments.
- Declaring a nonpure function as `pure` can lead to unexpected behavior.
- Imported tasks cannot be declared as `pure`.

480 © Cadence Design Systems, Inc. All rights reserved.



```
import "DPI-C" pure [ c_identifier = ] prototype;
```

A call to a pure function can be optimized by the simulator. For example:

- If the result is not used or is redundant, the call can be eliminated.
- If a call uses the same arguments as a previous call from which the return value has been stored, the previous value could be used and the call again eliminated.

There are some strict conditions for a function to be pure:

- The function cannot be void – i.e., it must return a nonvoid value.
- The function has at least one `input` argument and no `output` or `inout` arguments.
- The return value depends solely on the input argument values; for example, the function does not access any persistent data such as a global variable or static data.
- The function has no side-effects whatsoever; for example, it does not perform any file operations. In fact a pure function is not allowed to access anything outside of its arguments e.g., environmental variables.

If a nonpure function is declared as `pure`, then unexpected behavior may result as the simulator tries to optimize function calls that cannot be optimized.

Imported tasks can never be declared as `pure`.

Imported and Exported Functions Example

```

module calling_the_DPI;
import "DPI-C" context function calculate_pwr (int arr[5:0]);
export "DPI-C" function display_array;

initial begin
    int arr[5:0] = '{1,2,3,4,5,6};
    calculate_pwr(arr);
end

function void display_array(input int arr[6]);
    $display("SV: Contents of Array sent by C are %p",arr);
endfunction

endmodule

```

C: Element received from SV for a[0] is "6"
C: Element received from SV for a[1] is "5"
C: Element received from SV for a[2] is "4"
C: Element received from SV for a[3] is "3"
C: Element received from SV for a[4] is "2"
C: Element received from SV for a[5] is "1"
SV: Contents of Array sent by C are '(36, 25, 16, 9,
4, 1)

481 © Cadence Design Systems, Inc. All rights reserved.

OUTPUT



The SystemVerilog initial block calls the `import_c_func`, which is linked to the C function named `c_func` by the `import` declaration. This function is imported with the `context` property so that it can in turn call exported SystemVerilog subroutines.

The C implementation of the `c_func` function calls the `sv_func` function that it has previously declared to be `extern`. This function is linked to the SystemVerilog function named `export_sv_func` in the `export` declaration.



cadence®

© Cadence Design Systems, Inc. All rights reserved worldwide. Cadence, the Cadence logo, and the other Cadence marks found at <https://www.cadence.com/go/trademarks> are trademarks or registered trademarks of Cadence Design Systems, Inc. Accellera and SystemC are trademarks of Accellera Systems Initiative Inc. All Arm products are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All MIPI specifications are registered trademarks or service marks owned by MIPI Alliance. All PCI-SIG specifications are registered trademarks or trademarks of PCI-SIG. All other trademarks are the property of their respective owners.

This page does not contain notes.