



CUSTOMER EDUCATION SERVICES

# **SystemVerilog Testbench Workshop**

## **Student Guide**

50-I-052-SSG-016      2019.06

**Synopsys Customer Education Services**  
690 E. Middlefield Road  
Mountain View, California 94043

Workshop Registration: <https://training.synopsys.com>

# **Copyright Notice and Proprietary Information**

© 2019 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

## **Destination Control Statement**

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

## **Disclaimer**

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

## **Trademarks**

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <http://www.synopsys.com/Company/Pages/Trademarks.aspx>. All other product or company names may be trademarks of their respective owners.

## **Third-Party Links**

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.  
690 E. Middlefield Road  
Mountain View, CA 94043  
[www.synopsys.com](http://www.synopsys.com)

Document Order Number: 50-I-052-SSG-016  
SystemVerilog Testbench Workshop Student Guide

# Table of Contents

---

## Unit i: Introduction

Facilities .....	i-2
Curriculum Flow .....	i-3
Workshop Target Audience .....	i-4
Workshop Goals .....	i-5
Introductions .....	i-6
Agenda: Day 1 .....	i-7
Agenda: Day 2 .....	i-8
Agenda: Day 3 .....	i-9
Icons Used in this Workshop .....	i-10

---

## Unit 1: The Device Under Test (DUT)

Agenda .....	1-1
Unit Objectives .....	1-2
What Is the Device Under Test? .....	1-3
A Functional Perspective .....	1-4
The Router Description .....	1-5
Input Packet Structure .....	1-6
Output Packet Structure .....	1-7
Reset Signal .....	1-8
The DUT: router.sv .....	1-9
Unit Objectives Review .....	1-10

---

## Unit 2: SystemVerilog Verification Environment

Agenda .....	2-1
Unit Objectives .....	2-2
Verification Goals (1/2) .....	2-3
Verification Goals (2/2) .....	2-4
Process of Reaching Verification Goals .....	2-5
The SystemVerilog Test Environment .....	2-6
SystemVerilog – Key Features .....	2-7
Program Block – Encapsulate Test Code .....	2-8
Interface – Encapsulate Connectivity .....	2-9
Comparing SystemVerilog Containers .....	2-10
Interface – An Example .....	2-11
Synchronous Timing: clocking Blocks .....	2-12
Signal Direction Using modport .....	2-13
A Complete interface .....	2-14
Driving & Sampling DUT Signals .....	2-15
SystemVerilog Testbench Timing .....	2-16

# Table of Contents

Input and Output Skews.....	2-17
SystemVerilog Scheduling.....	2-18
Synchronous Drive Statements.....	2-19
Sampling Synchronous Signals.....	2-20
Signal Synchronization – Level Sensitive .....	2-21
Signal Synchronization – Edge Sensitive .....	2-22
Using Interface in Program.....	2-23
Complete Top-Level Harness .....	2-24
Compile RTL & Simulate with VCS .....	2-25
SystemVerilog Run-Time Options .....	2-26
Getting Help with VCS and SystemVerilog .....	2-27
Lab 1 Introduction.....	2-28
Unit Objectives Review .....	2-29
Appendix.....	2-30
Cycle Delay, Default Clocking, Synchronous Drive .....	2-31
Default clocking and Cycle Delay .....	2-32
Cycle Delay and Synchronous Drive.....	2-33
Cycle Delay Example .....	2-34
Synchronous Drive Example .....	2-35
Useful VCS Compile and Run Time Switches .....	2-36
Compiling and Running with VCS .....	2-37
Compiling with VCS – Legacy Verilog Code (1/2) .....	2-38
Compiling with VCS – Legacy Verilog Code (2/2) .....	2-39
External Binding of Components.....	2-40
External Binding of Components.....	2-41
Interactive Debugging with DVE & Verdi .....	2-42
DVE Testbench Debug: Getting Started.....	2-43
Verdi Testbench Debug: Getting Started.....	2-44

---

## Unit 3: SystemVerilog Language Basics -1

Agenda .....	3-1
Unit Objectives .....	3-2
SystemVerilog Testbench Code Structure .....	3-3
SystemVerilog Lexical Convention .....	3-4
2-State (0 1) Data Types (1/3).....	3-5
2-State (0 1) Data Types (2/3).....	3-6
2-State (0 1) Data Types (3/3).....	3-7
4-State (0 1 X Z) Data Types (1/2).....	3-8
4-State Data Types (2/2).....	3-9
String Data Type .....	3-10
Enumerated Data Types .....	3-11
Data Arrays – Fixed-size Arrays (1/4).....	3-12
Data Arrays – Dynamic Arrays (2/4).....	3-13
Data Arrays – Queues (3/4) .....	3-14

# Table of Contents

Queue Manipulation Examples.....	3-15
Data Arrays – Associative Arrays (4/4).....	3-16
Associative Array Examples.....	3-17
Array Loop Support and Reduction Operators .....	3-18
Array Methods (1/4) .....	3-19
Array Methods (2/4) .....	3-20
Array Methods (3/4) .....	3-21
Array Methods (4/4) .....	3-22
Data Arrays – Out-of-Bounds Access.....	3-23
Array Summary.....	3-24
Quiz Time .....	3-25
Quiz 1.....	3-26
Quiz 2.....	3-27
Unit Objectives Review .....	3-28
Appendix.....	3-29
Unpacked Array Performance.....	3-30
Unpacked Array Performance.....	3-31
Packed Array.....	3-32
Packed Array.....	3-33
Array Querying System Functions.....	3-34
Array Querying System Functions Examples.....	3-35
struct.....	3-36
struct - Data Structure .....	3-37
union .....	3-38
union - Data Union.....	3-39
Streaming Operators (Pack/Unpack) .....	3-40
Streaming Operators: Pack/Unpack (1/3) .....	3-41
Streaming Operators: Pack/Unpack (2/3) .....	3-42
Streaming Operators: Pack/Unpack (3/3) .....	3-43

---

## Unit 4: SystemVerilog Language Basics - 2

Agenda .....	4-1
Unit Objectives .....	4-2
System Functions: Randomization .....	4-3
User Defined Types and Type Cast .....	4-4
Operators.....	4-5
inside Operator.....	4-6
iff Operator.....	4-7
Know Your Operators! .....	4-8
Sequential Flow Control .....	4-9
Subroutines (task and function) .....	4-10
Subroutine Argument Binding and Skipping.....	4-11
Subroutine Arguments .....	4-12
Test for Understanding .....	4-13

# Table of Contents

Code Block Lifetime Controls .....	4-14
Helpful Debugging Features .....	4-15
Lab 2 Introduction.....	4-16
Unit Objectives Review .....	4-17
Appendix.....	4-18
Import and Export Verilog subroutines .....	4-19
Import and Export Verilog Subroutines (1/2).....	4-20
Import and Export Verilog Subroutines (2/2).....	4-21
Import and Export C/C++ subroutines (DPI) .....	4-22
SV Direct Programming Interface .....	4-23
DPI-C: import .....	4-24
DPI-C: export.....	4-25
Declaration of Imported Functions and Tasks.....	4-26
DPI-C: Supported Data Types .....	4-27
DPI-C: Supported Data Types .....	4-28
DPI-C Example: Integer and Strings .....	4-29
DPI-C: 4-State Data Types .....	4-30
DPI-C Example: reg/logic.....	4-31
DPI-C: chandle.....	4-32
DPI-C: Array Access .....	4-33
DPI-C: Array Access .....	4-34
DPI-C: Import Examples .....	4-35
DPI-C: Compile and Debug.....	4-36
DPI-C: Header Files & Examples.....	4-37

---

## Unit 5: Concurrency

Agenda .....	5-1
Unit Objectives .....	5-2
Day 1 Review.....	5-3
Day 1 Review (Building Testbench).....	5-4
Testbench Requires Concurrency .....	5-5
Concurrency in Simulators .....	5-6
Creating Concurrent Threads .....	5-7
How Many Child Threads?.....	5-8
Join Options .....	5-9
Thread Execution .....	5-10
Thread Execution Model.....	5-11
Thread Design (1/2) .....	5-12
Thread Design (2/2) .....	5-13
Sharing Variables Among Threads Forked Using join.....	5-14
Thread v/s Program Completion.....	5-15
Waiting for Child Threads to Finish .....	5-16
Thread Execution Issues .....	5-17
Thread Execution Issues: Unroll the for-loop.....	5-18

# Table of Contents

Thread Execution Issues: Local Variable .....	5-19
Thread Execution Issues: Unroll the for-loop.....	5-20
Implement Watch-Dog Timer with join_any.....	5-21
Avoiding disable fork Problems .....	5-22
Lab 3 Introduction.....	5-23
Unit Objectives Review .....	5-24
Appendix.....	5-25
Alternatives to disable fork .....	5-26
Alternatives to disable fork – kill() .....	5-27
Get and Save Thread Process Handle .....	5-28
Managing Time-Consuming Threads .....	5-29
Managing Non-Time-Consuming Threads.....	5-30

---

## Unit 6: OOP - Encapsulation

Agenda .....	6-1
Unit Objectives .....	6-2
Abstraction Enhances Re-Usability of Code .....	6-3
SystemVerilog OOP Program Constructs.....	6-4
OOP Encapsulation (OOP Class).....	6-5
module vs. class .....	6-6
Constructing OOP Objects.....	6-7
Accessing Object Members .....	6-8
Initialization of Object Properties.....	6-9
Initialization of Object Properties: this .....	6-10
OOP Data Hiding (Integrity of Data) 1/3 .....	6-11
OOP Data Hiding (Integrity of Data) 2/3 .....	6-12
OOP Data Hiding (Integrity of Data) 3/3 .....	6-13
Working with Objects – Handle Assignment .....	6-14
Working with Objects – Garbage Collection.....	6-15
Working with Objects – static Members .....	6-16
Working with Objects – const Properties .....	6-17
Working with Objects – Array Methods.....	6-18
Working with Objects – Concurrency .....	6-19
Parameterized Classes.....	6-20
forward typedef.....	6-21
Best Practices (1/2) .....	6-22
Best Practices (2/2) .....	6-23
Virtual Interfaces .....	6-24
SystemVerilog Packages.....	6-25
Packages: Example .....	6-26
Rules Governing Packages .....	6-27
Using Packages .....	6-28
Using Packages: Example (1/2) .....	6-29
Using Packages: Example (2/2).....	6-30

# Table of Contents

Quiz Time .....	6-31
OOP: Quiz 1.....	6-32
OOP: Quiz 2.....	6-33
OOP: Quiz 3.....	6-34
Unit Objectives Review .....	6-35
Appendix.....	6-36
SystemVerilog Virtual Interface.....	6-37
Virtual Interfaces (1/5) .....	6-38
Virtual Interfaces (2/5) .....	6-39
Virtual Interfaces (3/5) .....	6-40
Virtual Interfaces (4/5) .....	6-41
Virtual Interfaces (5/5) .....	6-42
Singleton Objects.....	6-43
Singleton Objects.....	6-44

---

## Unit 7: OOP - Randomization

Agenda .....	7-1
Unit Objectives .....	7-2
Alternatives to Exhaustive Testing? .....	7-3
Process of Reaching Verification Goals .....	7-4
OOP Based Randomization .....	7-5
Randomization Example.....	7-6
Controlling Random Variables (1/2) .....	7-7
Controlling Random Variables (2/2) .....	7-8
SystemVerilog Constraints .....	7-9
Weighted Constraints .....	7-10
Array Constraint Support.....	7-11
Implication and Order Constraints.....	7-12
Equivalence Constraints.....	7-13
Uniqueness Constraints.....	7-14
System Functions .....	7-15
User-defined Functions in Constraints.....	7-16
Real Numbers.....	7-17
Constraint Solver Order .....	7-18
Inline Constraints .....	7-19
Soft Constraints.....	7-20
Where are Soft Constraints Used? .....	7-21
Mutually Constrained Random Variables.....	7-22
Inconsistent Constraints .....	7-23
Effects of Calling randomize().....	7-24
Controlling Randomization at Runtime .....	7-25
Controlling Constraints at Runtime .....	7-26
Constraint Prototypes.....	7-27
Nested Objects with Random Variables .....	7-28

# Table of Contents

std::randomize() .....	7-29
Changing the Random Seed at Simulation .....	7-30
Quiz Time .....	7-31
Randomization: Quiz 1 .....	7-32
Randomization: Quiz 2 .....	7-33
Randomization: Quiz 3 .....	7-34
Randomization: Quiz 4 .....	7-35
Randomization: Quiz 5 .....	7-36
Lab 4 Introduction.....	7-37
Unit Objectives Review .....	7-38
Appendix.....	7-39
struct Randomization .....	7-40
struct Randomization Example (1/2) .....	7-41
struct Randomization Example (2/2) .....	7-42
Soft Constraints: Rules and Management.....	7-43
How do soft constraints work? .....	7-44
Priority of Soft Constraints (1/5) .....	7-45
Priority of Soft Constraints (2/5) .....	7-46
Priority of Soft Constraints (3/5) .....	7-47
Priority of Soft Constraints (4/5) .....	7-48
Priority of Soft Constraints (5/5) .....	7-49
Disabling Soft Constraints (1/3) .....	7-50
Disabling Soft Constraints (2/3) .....	7-51
Disabling Soft Constraints (3/3) .....	7-52
Soft Constraint Debug using Verdi.....	7-53
Debug Soft Constraints using DVE .....	7-54
Random Stability .....	7-55
Random Stability: Threads and Objects .....	7-56
Constraint Debug and Profiling .....	7-57
Constraint Debug Methods .....	7-58
Interactive Constraint Debug Using Verdi .....	7-59
Constraint Debug & Analysis Flow Using Verdi .....	7-60
Flexible Breakpoint Infrastructure.....	7-61
Debugging Inconsistent Constraints Using Verdi.....	7-62
Cross Probing Using Verdi .....	7-63
Relation Space Analysis Using Verdi.....	7-64
Interactive Constraint Editing Using Verdi .....	7-65
On-the-fly Re-randomization Using Verdi .....	7-66
Constraint Debug and Analysis Using Verdi.....	7-67
Interactive Constraint Debug Using DVE .....	7-68
Debug - Constraint Conflicts .....	7-69
Interactive Constraint Debug Using DVE (1/5).....	7-70
Interactive Constraint Debug Using DVE (2/5).....	7-71
Interactive Constraint Debug Using DVE (3/5).....	7-72
Interactive Constraint Debug Using DVE (4/5).....	7-73
Interactive Constraint Debug Using DVE (5/5).....	7-74

# Table of Contents

Constraint Profiling.....	7-75
Performance: Solver Diagnostics.....	7-76
Constraint Profile Using Simprofile (1/2).....	7-77
Constraint Profile Using Simprofile (2/2).....	7-78
Methodology and Best Practices.....	7-79
Best Practices: Methodology .....	7-80
Best Practices: State Variables.....	7-81
Best Practices: Variable Ranges .....	7-82
Performance: pre-/post-randomize() .....	7-83
Best Practices: Randomizing Scenarios .....	7-84
Performance Example.....	7-85
SV Constraints Best Practices.....	7-86
Watch Out for Verilog Rules (1/4) .....	7-87
Watch Out for Verilog Rules (2/4) .....	7-88
Watch Out for Verilog Rules (3/4) .....	7-89
Watch Out for Verilog Rules (4/4) .....	7-90
Watch Out for State Variables (1/4) .....	7-91
Watch Out for State Variables (2/4) .....	7-92
Watch Out for State Variables (3/4) .....	7-93
Watch Out for State Variables (4/4) .....	7-94
Watch Out for Hierarchical Constraints (1/4).....	7-95
Watch Out for Hierarchical Constraints (2/4).....	7-96
Watch Out for Hierarchical Constraints (3/4).....	7-97
Watch Out for Hierarchical Constraints (4/4).....	7-98
solve..before Changes the Probabilities .....	7-99
Recommendations on solve..before, dist .....	7-100
Implementation Choices .....	7-101
Implementation Choices: Example 1 .....	7-102
Implementation Choices: Example 2 (1/3) .....	7-103
Implementation Choices: Example 2 (2/3) .....	7-104
Implementation Choices: Example 2 (3/3) .....	7-105

---

## Unit 8: OOP - Inheritance

Agenda .....	8-1
Unit Objectives .....	8-2
Day 2 Review - Creating Concurrent Threads.....	8-3
Day 2 Review - OOP Class.....	8-4
Day 2 Review - OOP Based Randomization .....	8-5
Object Oriented Programming: Inheritance.....	8-6
Object Oriented Programming: Inheritance.....	8-7
OOP: Polymorphism.....	8-8
OOP: Polymorphism.....	8-9
OOP: Polymorphism.....	8-10
Modifying Constraints for Test Cases .....	8-11

# Table of Contents

Data Protection: local.....	8-12
Data Protection: protected.....	8-13
Constructing Derived Class Objects .....	8-14
Test for Understanding 1 .....	8-15
Test for Understanding 1: Answers .....	8-16
Test for Understanding 1: Guideline.....	8-17
Test for Understanding 2 .....	8-18
Test for Understanding 2: Answer.....	8-19
Quiz Time .....	8-20
Inheritance: Quiz 1 .....	8-21
Inheritance: Quiz 2.....	8-22
Inheritance: Quiz 3.....	8-23
Inheritance: Quiz 4.....	8-24
Inheritance: Quiz 5.....	8-25
Unit Objectives Review .....	8-26
Appendix.....	8-27
Interface Class.....	8-28
Interface Class.....	8-29
Interface Class – Rules.....	8-30
Interface Class – Multiple Inheritance .....	8-31
Interface Class – Partial Implementation.....	8-32
Interface Class – Multiple Extends and Diamond Relationship .....	8-33

---

## Unit 9: Inter-Thread Communication

Agenda .....	9-1
Unit Objectives .....	9-2
Inter-Thread Communications (ITC).....	9-3
Event Based ITC .....	9-4
Event Based ITC Example.....	9-5
Event Wait Syntax .....	9-6
Trigger Syntax .....	9-7
Controlling Termination of Simulation .....	9-8
Resource Sharing ITC.....	9-9
Semaphores (1/2) .....	9-10
Semaphores (2/2) .....	9-11
Using Semaphores .....	9-12
Using Semaphore Arrays .....	9-13
Arbitration Example Using Semaphore .....	9-14
Mailbox .....	9-15
Mailbox Class .....	9-16
Creating Mailboxes .....	9-17
Putting Messages into Mailboxes .....	9-18
Retrieving Messages from Mailboxes (get).....	9-19
Retrieving Messages from Mailboxes (peek) .....	9-20

# Table of Contents

Lab 5 Introduction.....	9-21
Unit Objectives Review .....	9-22

---

## Unit 10: Functional Coverage

Agenda .....	10-1
Unit Objectives .....	10-2
Phases of Functional Verification.....	10-3
Combinational Logic Example .....	10-4
State Transition Example.....	10-5
Cross Correlation Example .....	10-6
Functional Coverage in SystemVerilog .....	10-7
Functional Coverage Example .....	10-8
State Bin Creation (Automatic) .....	10-9
Measuring Coverage .....	10-10
Automatic State Bin Creation Example .....	10-11
State and Transition Bin Creation (User).....	10-12
Cross Coverage Bin Creation.....	10-13
Wildcard Bins .....	10-14
Bin Coverage – with .....	10-15
Specifying Sample Event Timing .....	10-16
Parameterized Sample Method .....	10-17
Parameterized Sample Method: Example.....	10-18
Determining Coverage Progress .....	10-19
Coverage Measurement Example .....	10-20
Coverage Attributes (1/2) .....	10-21
Coverage Attributes (2/2) .....	10-22
Major Coverage Attributes .....	10-23
Control and Query of Covergroups.....	10-24
Parameterized Coverage Group .....	10-25
Coverage Result Reporting Utilities .....	10-26
Sample URG HTML Report.....	10-27
Lab 6 Introduction.....	10-28
Unit Objectives Review .....	10-29
Appendix.....	10-30
Merging Coverage Results.....	10-31
Merging Coverage Results.....	10-32
Merging Coverage Results – Parallel Merging.....	10-33
Merging Functional Coverage .....	10-34
Union Merge Example (1/3) .....	10-35
Union Merge Example (2/3) .....	10-36
Union Merge Example (3/3) .....	10-37
Reference Merge .....	10-38
Test Records and Merging.....	10-39
Covergroup Exclusion .....	10-40

# Table of Contents

Covergroup Exclusion .....	10-41
Exclusion through DVE .....	10-42
Adaptive Exclusion .....	10-43
Verdi Adaptive Exclusions .....	10-44
Verdi Coverage Exclusion Manager .....	10-45
Exclusion through URG .....	10-46
Test Grading .....	10-47
Fast Test Grading .....	10-48
Cost-Based Grading Example .....	10-49
Test Grading: Additional Options .....	10-50

---

## Unit 11: SystemVerilog UVM Preview

Agenda .....	11-1
Unit Objectives .....	11-2
UVM – Universal Verification Methodology .....	11-3
Origin of UVM .....	11-4
Coverage-Driven Verification .....	11-5
Phases of Verification .....	11-6
The Testbench Environment/Architecture .....	11-7
Goal - Run More Tests, Write Less Code .....	11-8
UVM Guiding Principles .....	11-9
UVM Encourages Encapsulation for Reuse .....	11-10
UVM Structure is Scalable .....	11-11
Standards: Structural Support in UVM .....	11-12
Standards: Component Phasing .....	11-13
Standards: Component Configuration .....	11-14
Standards: Reporting and Handshaking .....	11-15
Standards: Implementing UVM Test .....	11-16
Unit Objectives Review .....	11-17
That's all Folks! .....	11-18

---

## Unit CS: Customer Support

Synopsys Support Resources .....	CS-2
SolvNet Online Support .....	CS-3
SolvNet Registration .....	CS-4
Support Center .....	CS-5
Other Technical Sources .....	CS-6
Summary: Getting Support .....	CS-7

# Table of Contents

This page was intentionally left blank.

# SystemVerilog Testbench

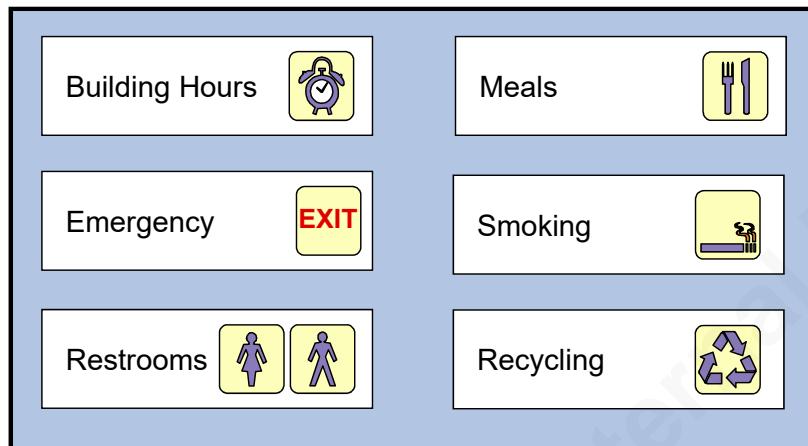
## VCS 2019.06

Synopsys Customer Education Services

© 2019 Synopsys, Inc. All Rights Reserved

Synopsys 50-I-052-SSG-016

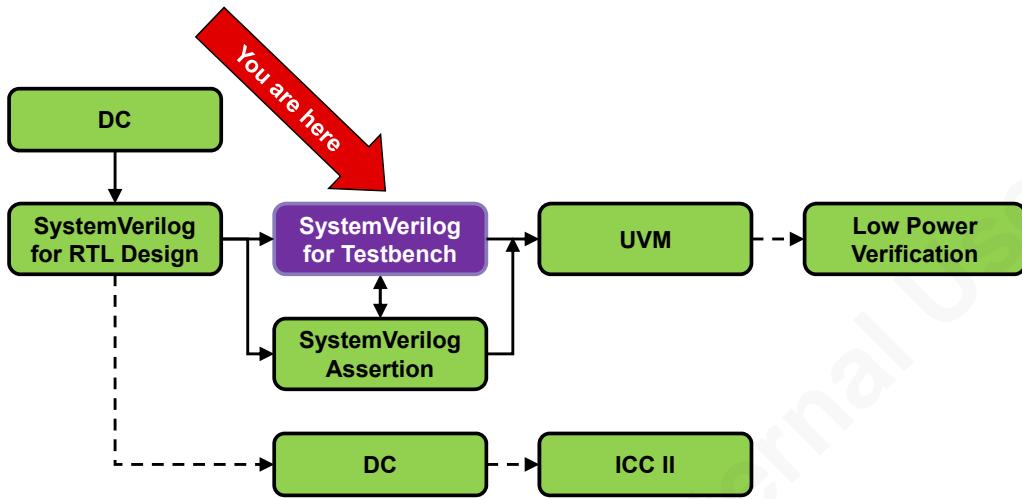
# Facilities



Please turn off or silence your cell phone

i-2

# Curriculum Flow



i-3

The entire Synopsys Customer Education Services course offering can be found at:  
<https://www.synopsys.com/support/training.html>

## Workshop Target Audience

Design or verification engineers  
writing SystemVerilog test-benches  
to verify Verilog or SystemVerilog  
code



i-4

## Workshop Goals



- Acquire the skills to write a SystemVerilog testbench to verify Verilog/SystemVerilog RTL code with coverage-driven random stimulus***

i- 5

# Introductions

- **Name**
- **Company**
- **Job Responsibilities**
- **Relevant Experience**
- **Main Goal(s) and Expectations for this Course**

i- 6

# Agenda: Day 1

DAY  
1

1 The Device Under Test (DUT)

2 SystemVerilog Verification Environment 

3 SystemVerilog Language Basics - 1

4 SystemVerilog Language Basics - 2 

i-7

## Agenda: Day 2

DAY  
2

5 Concurrency



6 Object Oriented Programming (OOP)  
– Encapsulation

7 Object Oriented Programming (OOP)  
– Randomization



i-8

# Agenda: Day 3

DAY  
3

- |    |  |   |
|----|--|---|
| 8  | Object Oriented Programming (OOP)<br>– Inheritance |  |
| 9  | Inter-Thread Communications                        |  |
| 10 | Functional Coverage                                |  |
| 11 | SystemVerilog UVM Preview                          |  |
| 12 | Customer Support                                   |  |

i-9

# Icons Used in this Workshop



**Lab Exercise**



**Caution**



**Recommendation**



**Question**



**For Further Reference**



**Exercise**



**Definition of  
Acronyms**

i-10

**Lab Exercise:** A lab is associated with this unit, module, or concept.

**Recommendation:** Recommendations, tips, performance boost, etc.

**For Further Reference:** Identifies pointer or URL to other references or resources.

**Caution:** Warnings of common mistakes, unexpected behavior, etc.

**Question:** Marks questions asked on the slide.

**Exercise:** Test for Understanding (TFU), which may require you to work in groups.

**Definition of Acronyms:** Defines the acronym used in the slides.

# Agenda

DAY  
1

1 The Device Under Test (DUT)

2 SystemVerilog Verification Environment 

3 SystemVerilog Language Basics - 1

4 SystemVerilog Language Basics - 2 

# Unit Objectives



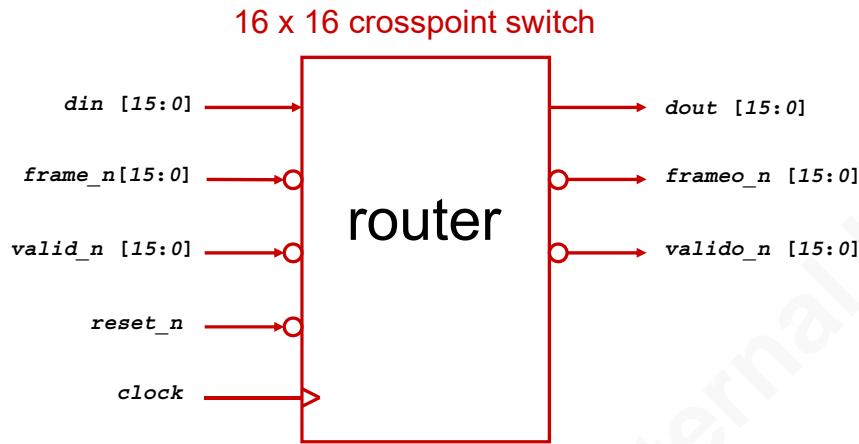
**After completing this unit, you should be able to:**

- **Describe the function of the Device Under Test (DUT)**
- **Identify the control and data signals of the DUT**
- **Draw timing diagram for sending and receiving a packet of data through the DUT**

1-2

# What Is the Device Under Test?

A router:



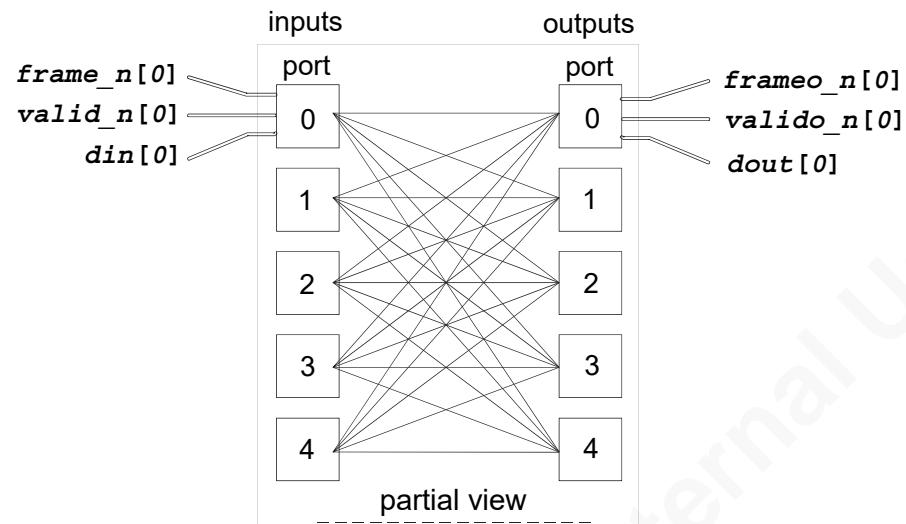
1-3

The router has 16 input and 16 output ports. Each input and output port consists of 3 signals, serial data, frame and valid. These signals are represented in a bit-vector format, din[15:0], frame\_n[15:0], valid\_n[15:0], dout[15:0], frameo\_n[15:0] and valido\_n[15:0].

To drive an individual port, the specific bit position corresponding to the port number must be specified. For example, if input port 3 is to be driven, then the corresponding signals shall be din[3], frame\_n[3] and valid\_n[3].

To sample an individual port, the specific bit position corresponding to the port number must be specified. For example, if output port 7 is to be sampled, then the corresponding signals shall be dout[7], frameo\_n[7] and valido\_n[7].

# A Functional Perspective



1-4

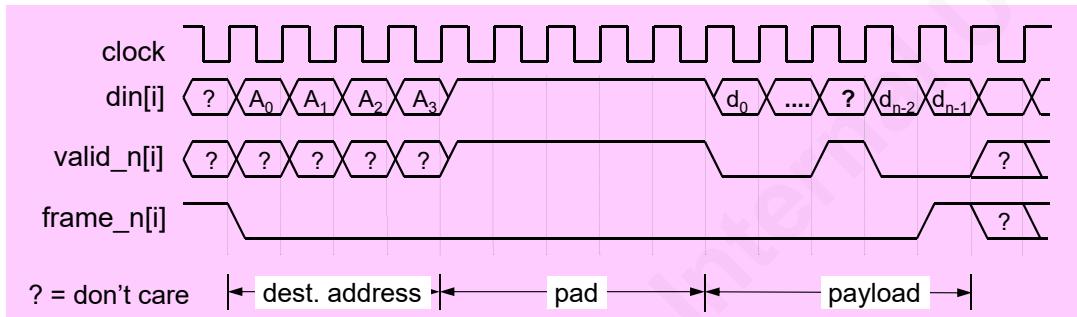
## The Router Description

- Single positive-edge clock
- Input and output data are serial (1 bit / clock)
- Packets are sent through in variable length:
  - Each packet is composed of two parts
    - ◆ Header
    - ◆ Payload
- Packets can be routed from any input port to any output port on a packet-by-packet basis
- No internal buffering or broadcasting (1-to-N)

1-5

# Input Packet Structure

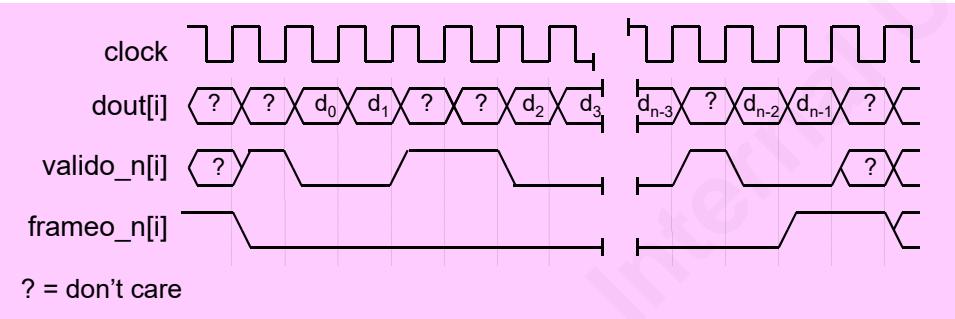
- **frame\_n:**
  - Falling edge indicates first bit of packet
  - Rising edge indicates last bit of packet
- **din:**
  - Header (destination address & padding bits) and payload
- **valid\_n:** low if payload bit is valid, high otherwise



1-6

# Output Packet Structure

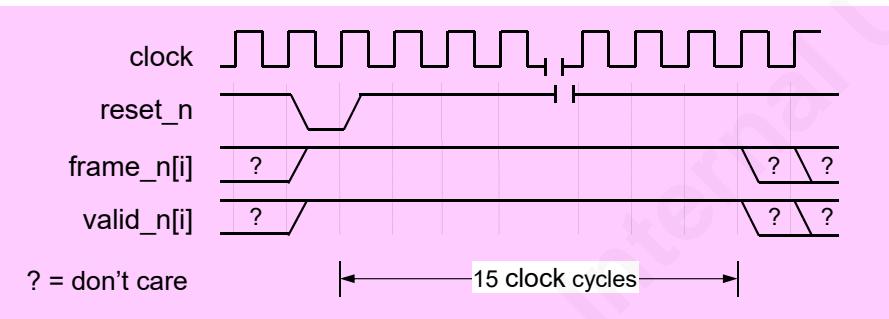
- Output activity is indicated by:  
*frameo\_n*, *valido\_n*, and *dout*
- Data is valid only when:
  - *frameo\_n* output is low (except for last bit)
  - *valido\_n* output is low
- Header field is stripped



1-7

# Reset Signal

- While asserting `reset_n`
  - `frame_n` and `valid_n` must be de-asserted
- `reset_n` is asserted for at least one clock cycle
- After de-asserting `reset_n`, wait for 15 clocks before sending a packet through the router

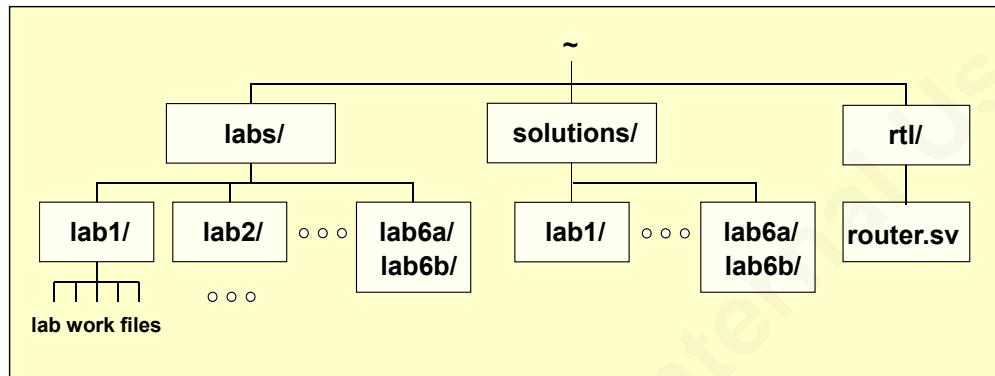


1-8

During these 15 clock cycles, the router is performing self-initialization. If you attempt to drive a packet through the router during this time, the self-initialization will fail and the router will not work correctly afterwards.

## The DUT: `router.sv`

- The Design Under Test, `router.sv`, is a SystemVerilog file
  - Located under `rtl` directory



1-9

## Unit Objectives Review

**Having completed this unit, you should be able to:**

- **Describe the function of the Device Under Test (DUT)**
- **Identify the control and data signals of the DUT**
- **Draw timing diagram for sending and receiving a packet of data through the DUT**



1-10

# Agenda

DAY  
1

1 The Device Under Test (DUT)

2 SystemVerilog Verification Environment 

3 SystemVerilog Language Basics - 1

4 SystemVerilog Language Basics - 2 

# Unit Objectives



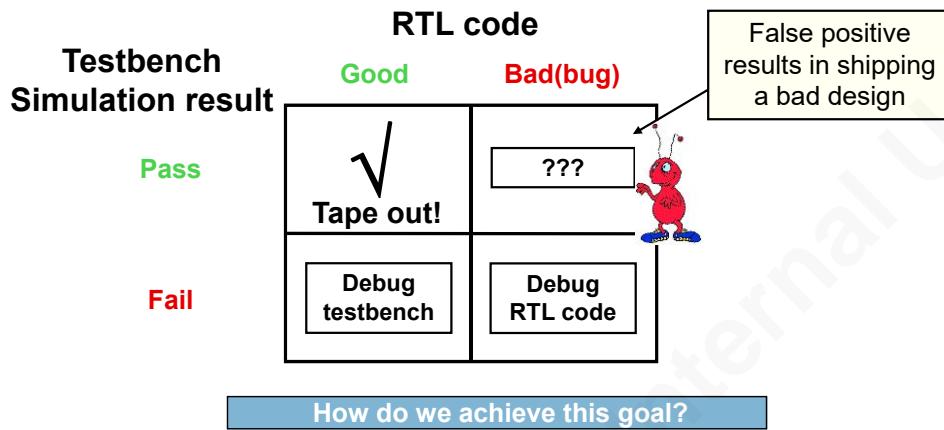
After completing this unit, you should be able to:

- Describe the process of reaching verification goals
- Describe components of a SystemVerilog testbench environment
- Describe program and interface constructs
- Compile and simulate a SV testbench
- Drive and sample DUT signals
- Synchronize to known point in simulation

2-2

## Verification Goals (1/2)

- Verify RTL design code
  - Fully conforms with specifications
- Must avoid false positives



2-3

No one ever wants to ship bad (faulty) RTL. The goal of verification is to find all the bugs in the RTL code.

It is imperative that your verification environment expose as many bugs in the environment as possible.

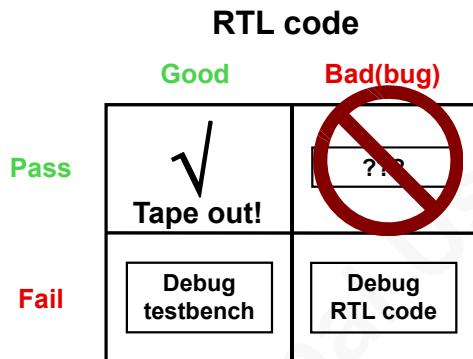
## Verification Goals (2/2)

- **Test Environment must:**

- Be structured for Debug
- Avoid False Positives

- **Tests must:**

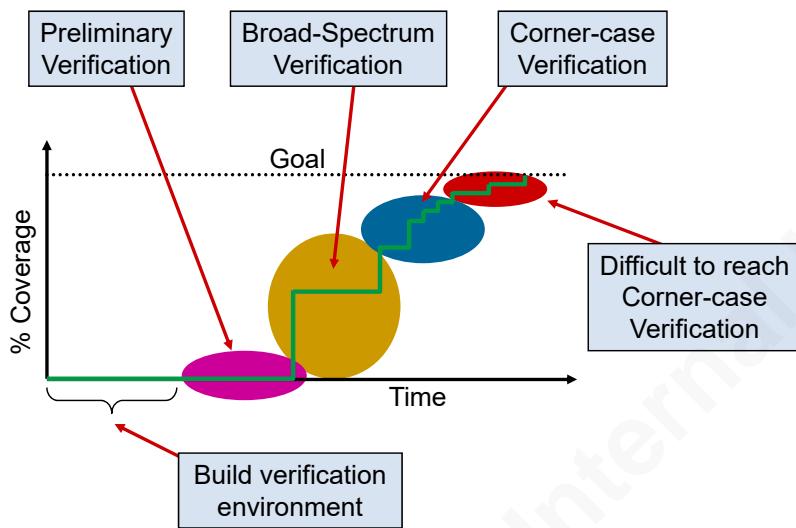
- Achieve Functional Coverage
  - ◆ Prevent untested regions
- Reach Corner Cases
  - ◆ Anticipated Cases
  - ◆ Error Injection
    - Environment Error
    - DUT Error
  - ◆ Unanticipated Cases
    - Random Tests
- Be robust, reusable, scalable



2-4

# Process of Reaching Verification Goals

## Phases of verification



2-5

The process of reaching the verification goal starts with the definition of the verification goal. What does it mean to be done with testing? Typically, the answer lies in the functional coverage spec within a verification plan. The goal is then to reach 100% coverage of the defined functional coverage spec in the verification plan.

Once the goal has been defined, the first step in constructing the testbench is to build the verification environment.

To verify the environment is set up correctly, preliminary verification tests are usually executed to wring out the rudimentary RTL and testbench errors.

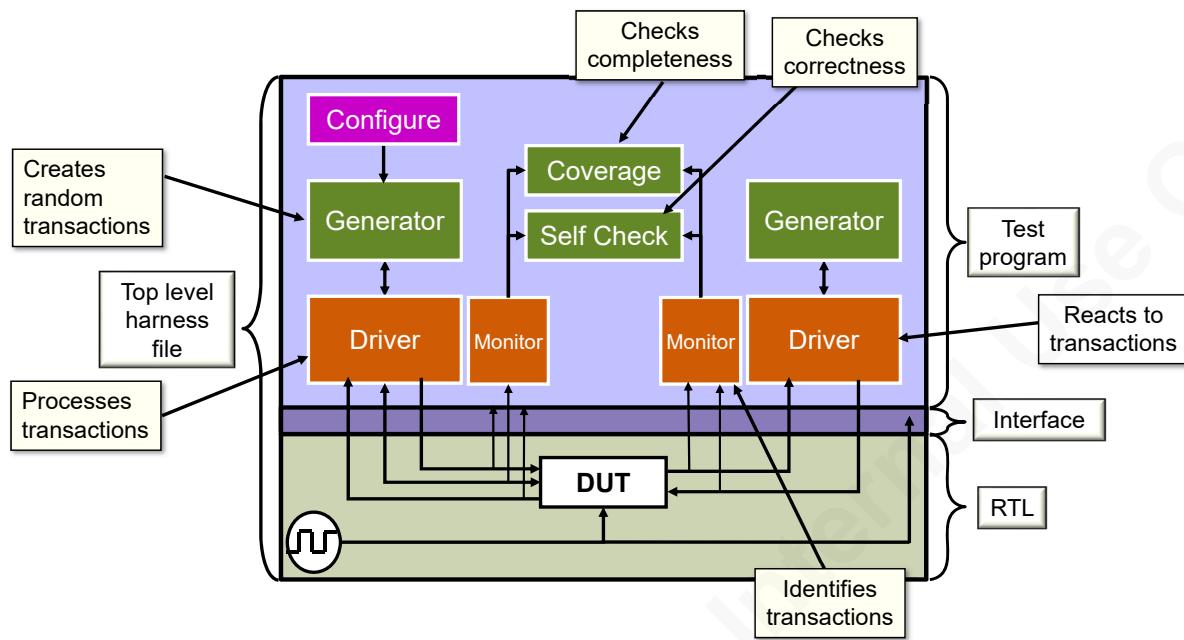
When the testbench environment is deemed to be stable, broad-spectrum verification based on random stimulus generation is utilized to quickly detect and correct the majority of the bugs in both RTL code and testbench code.

Based on functional coverage analysis, the random-based tests are then constrained to focus on corner-cases not yet reached via broad-spectrum testing.

Finally, for the very difficult to reach corner cases, customized directed tests are used to bring the coverage up to 100%.

Verification is complete when you reach 100% coverage as defined in the verification plan.

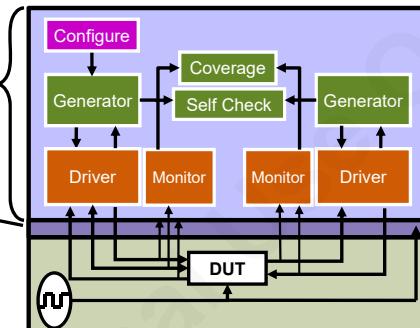
# The SystemVerilog Test Environment



# SystemVerilog – Key Features

- SystemVerilog introduces two new design units

- The **program** block
  - ◆ Is where you develop testbench code
  - ◆ Is entry point for testbench execution
- The **interface**
  - ◆ Is mechanism to connect testbench to DUT
  - ◆ Is a named bundle of wires
  - ◆ Can be passed just like a port in a port list



- SystemVerilog programs use Object Oriented Programming (OOP)

- Uses **class** definitions
  - ◆ discussed later in this workshop

2-7

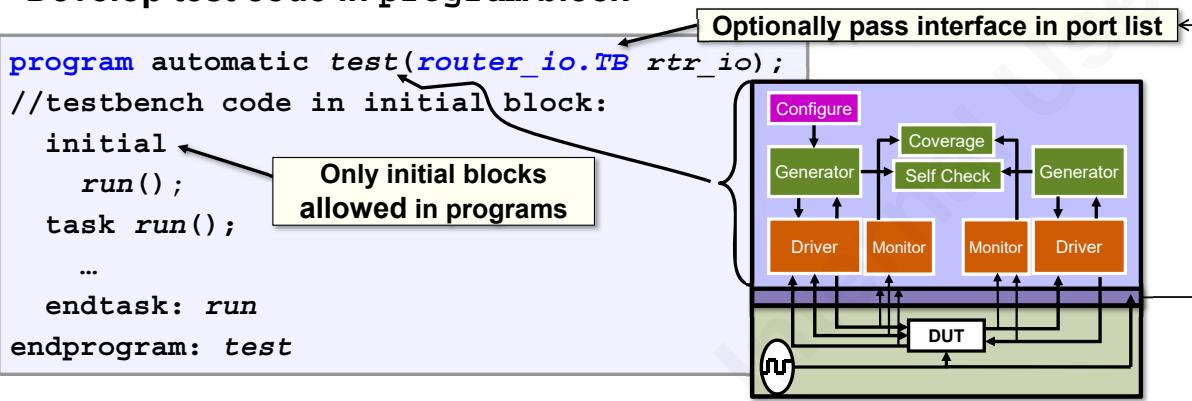
# Program Block – Encapsulate Test Code

## ■ The program block provides

- Entry point to test execution
- Scope for program-wide data and routines
- Race-free interaction between testbench and design

## ■ Develop test code in program block

```
program automatic test(router_io.TB rtr_io);
  //testbench code in initial block:
  initial
    run();
  task run();
    ...
  endtask: run
endprogram: test
```



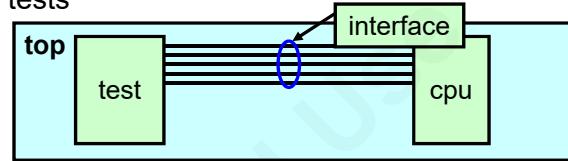
2-8

By default, SystemVerilog programs are static, just like Verilog module instances, tasks and functions. To allow dynamic allocation of memory like Vera and C++, always make programs automatic.

# Interface – Encapsulate Connectivity

- An interface encapsulates the communication between DUT and Testbench including

- Connectivity (signals) – named bundle of wires
  - ◆ One or more bundles to connect modules and tests
  - ◆ Can be reused for different tests and devices
- Directional information (modports)
- Timing (clocking blocks)
- Functionality (routines, assertions, initial/always blocks)



- Solves many problems with traditional connections

- Port lists for the connections are compact
- No missed connections
- Easy to add new connections

2-9

There are only a few (usually one or two) bundles to pass making the connection very compact. New signals in the interface are automatically passed to test program or module preventing connection problems.

Changes are easily made in interface making it easy to integrate in higher levels.

# Comparing SystemVerilog Containers

Hardware (DUT)		Testbench	
module	interface	static	dynamic
<b>module instance</b>			
<b>interface instance</b>	<b>interface instance</b>		
<b>clocking</b>	<b>clocking</b>	<b>clocking</b>	
<b>class</b>	<b>class</b>	<b>class</b>	<b>class</b>
<b>object</b>	<b>object</b>	<b>object</b>	<b>object</b>
<b>reg (logic)</b>	<b>reg (logic)</b>	<b>reg (logic)</b>	<b>reg (logic)</b>
<b>variable</b>	<b>variable</b>	<b>variable</b>	<b>variable</b>
<b>wire</b>	<b>wire</b>	<b>wire</b>	
<b>assign</b>	<b>assign</b>	<b>assign</b>	
<b>initial/always</b>	<b>initial/always</b>	<b>initial</b>	
<b>task</b>	<b>task</b>	<b>task</b>	<b>task</b>
<b>function</b>	<b>function</b>	<b>function</b>	<b>function</b>

2-10

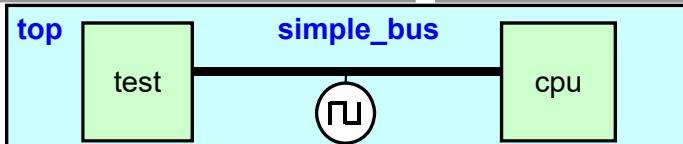
An object is an instance of a class.

## Interface – An Example

- The RTL code is connected with bundled signals

```
program automatic test(simple_bus sb);  
...  
endprogram
```

```
module cpu(simple_bus sb);  
...  
endmodule
```



```
interface simple_bus(input bit clk);  
    logic req, gnt;  
    logic [7:0] addr;  
    wire [7:0] data;  
    logic [1:0] mode;  
    logic start, rdy;  
endinterface
```

```
module top;  
    logic clk = 0;  
    always #10 clk = !clk;  
    simple_bus sb(clk);  
    test t1(sb);  
    cpu c1(sb);  
endmodule
```

2-11

Interfaces are defined just like modules, but unlike a **module**, an **interface** instance may be passed as a port list to a **program** or **module**.

All the signals that are in common between test and DUT have been encapsulated into a single container **simple\_bus**. Notice that test and DUT do not declare these signals, instead these modules simply use the **interface** as the connection between them.

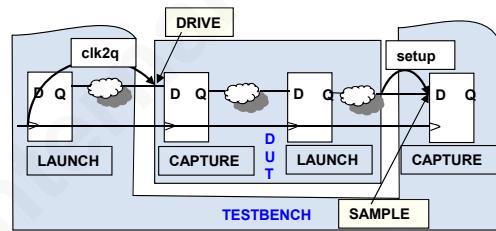
If an **interface** instance is to be used in **program** block, the data type for the signals should be **logic**. The reason is that signals within a **program** block are almost always driven within a procedural block (**initial**). All signals driven within a procedural block must be of type **reg**, the synonym of which is **logic**. When a signal is declared to be **logic**, it can also be driven by a continuous assignment statement. This added flexibility of **logic** is generally desirable.

There is an exception to the above recommendation. If the signal is a bi-directional signal (**inout**), or has multiple drivers, then the data type must be **wire** (or any other form of wire-types).

# Synchronous Timing: clocking Blocks

- Are just for testbench
  - ◆ Emulate the launch and capture flops at IO of DUT
- Create explicit synchronous timing domains
  - ◆ All signals driven or sampled at clocking event
    - by default all interface signals are asynchronous
  - ◆ Interaction between testbench and DUT ideally happens only at clock edges (cycle-based)
- Specify signal direction
  - ◆ Outputs can not be sampled
  - ◆ Input signals cannot be driven
- Typically three per interface
  - ◆ active driver
  - ◆ reactive driver
  - ◆ monitor

```
setup Clock to q
clocking cb @ (posedge clk);
  default input #1ns output #1ns;
  output reset_n;
  output din;
  output frame_n;
  output valid_n;
  input dout;
  input busy_n;
  input valido_n;
  input frameo_n;
endclocking: cb
```



2-12

# Signal Direction Using modport

## ■ Enforce signal access & direction with **modport**

```
interface router_io(input bit clock);
    logic reset_n;
    ...
    clocking cb @(posedge clock);
        default input #1ns output #1ns;
        output reset_n;
        output valid_n;
    ...
endclocking
modport DUT(input reset_n, input din, output dout,...);
modport TB(clocking cb, output reset_n);
endinterface: router_io

program automatic test(router_io.TB rtr_io);
    initial begin
        rtr_io.reset_n = 1'b0 ;
        rtr_io.cb.reset_n <= 1'b1 ;
        rtr_io.cb.valid_n <= ~('b0) ;
    end
endprogram: test
```

```
module router(
    router_io.DUT dut_io,
    input logic clk);
    ...
endmodule: router
```

2-13

A new construct related to interface is also added: **modport**. This provides direction information for module interface ports and controls the use of tasks and functions within certain modules. The directions of ports are those seen from the perspective of the module or the program.

You can define a different view of the interface signals that each module or program sees on its interface ports.

Definition is made within the interface, and it may have any number of modports.

A modport does not contain vector sizes or data types (common error) - only whether the connecting module sees a signal as input, output, inout or ref port.

# A Complete interface

```
interface router_io(input bit clock);
    logic reset_n;
    logic [15:0] din;
    logic [15:0] frame_n;
    logic [15:0] valid_n;
    logic [15:0] dout;
    logic [15:0] busy_n;
    logic [15:0] valido_n;
    logic [15:0] frameo_n;

```

**Named bundle of asynchronous signals**

```
    router.io.sv ...
    clocking cb @(posedge clock);
        default input #1ns output #1ns;
        output reset_n;
        output din;
        output frame_n;
        output valid_n;
        output valido_n;
        input dout;
        input busy_n;
        input valido_n;
        input frameo_n;
    endclocking
    modport TB(clocking cb, output reset_n);
endinterface
```

**Create synchronous behavior by placing into `clocking` block**

**Define access and direction with `modport`**

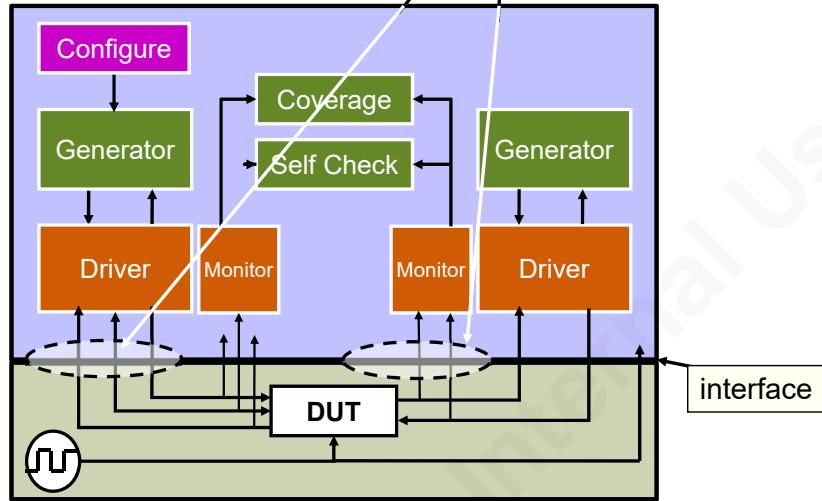
**Sample and drive skews**

**Synchronous** → **Asynchronous**

2-14

# Driving & Sampling DUT Signals

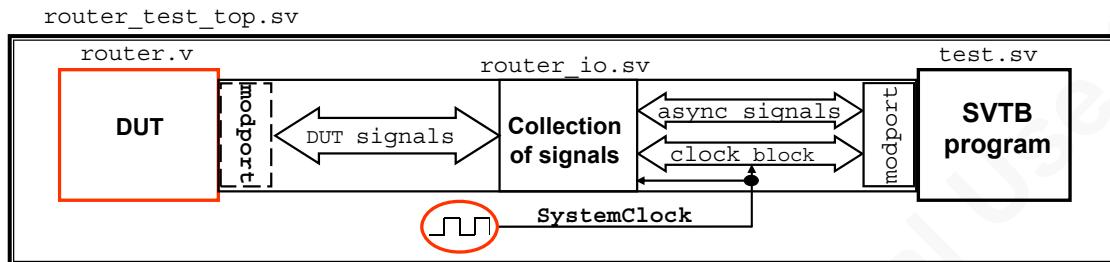
- DUT signals are driven in the device driver
- DUT signals are sampled in the device monitor



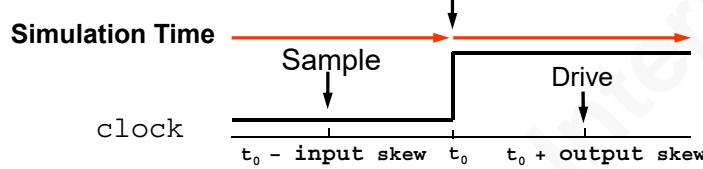
2-15

# SystemVerilog Testbench Timing

- Clocking block emulates synchronous drives and samples
  - Driving and sampling events occur at clocking event

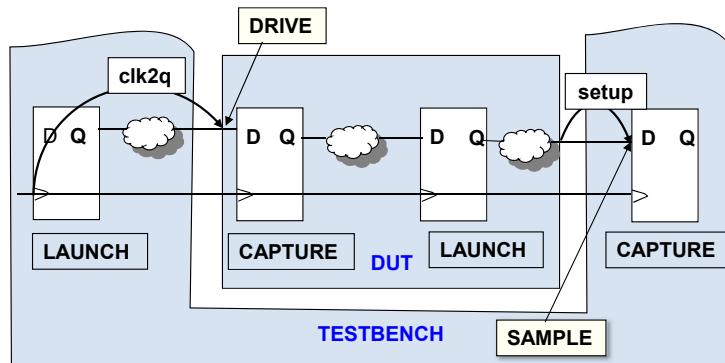


SystemVerilog Testbench (SVTB) synchronous program code execution



2-16

# Input and Output Skews

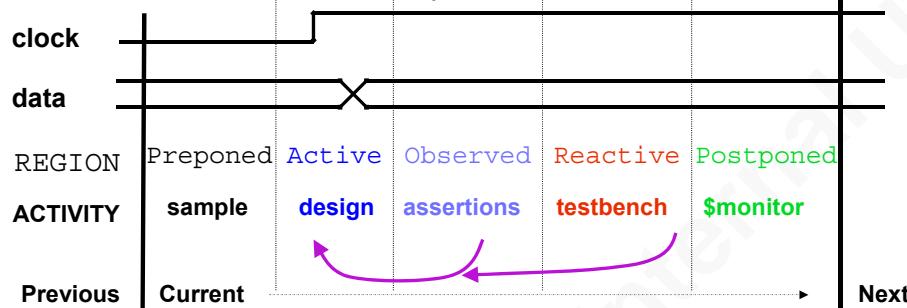


- **Output Skew is the clk2q delay of the launch flop for the DUT input**
  - Defaults to #0
- **Input skew is the setup time of the capture flop for the DUT output**
  - Defaults to #1step – prepended region of simulation step

2-17

# SystemVerilog Scheduling

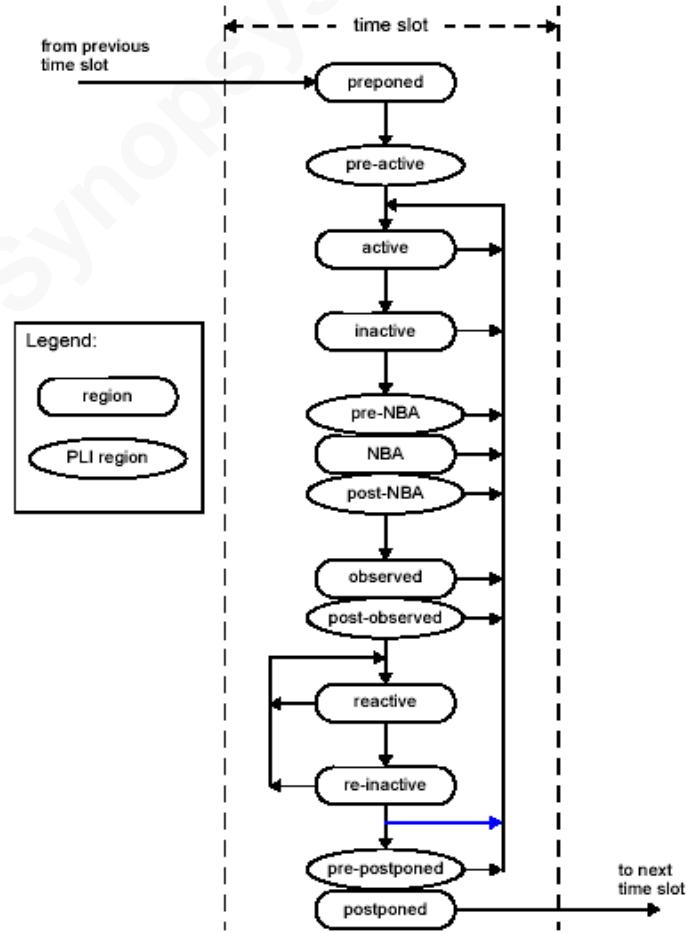
- Each time slot is divided into 5 major regions
  - Preponed Sample signals before any changes (#1step)
  - Active Design simulation (module), including NBA
  - Observed Assertions evaluated after design executes
  - Reactive Testbench activity (program)
  - Postponed Read only phase



Assertion and testbench events can trigger more design evaluations in this time slot

2-18

From IEEE 1800 standard:

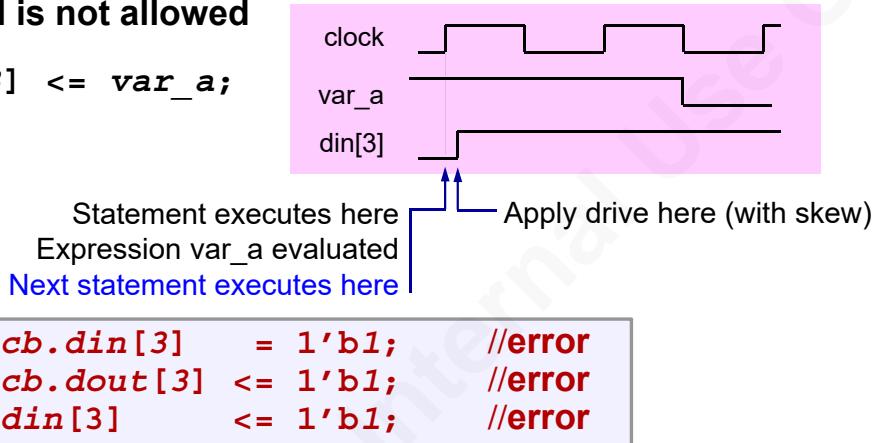


# Synchronous Drive Statements

```
interface.cb.signal <= <value | expression>;
```

- Drive must be non-blocking
- Driving of input signal is not allowed

```
rtr_io.cb.din[3] <= var_a;
```



2-19

Assume the clocking block *cb* in the *router\_io* interface discussed a few slides earlier.

Explanation for the errors:

The first error is due to driving using blocking assignment.

The correction is to make the drive a non-blocking assignment:

```
rtr_io.cb.din[3] <= 1'b1;
```

The second error is an illegal drive of input signal. No correction is possible

The third error is caused by forgetting to drive via the clocking block.

The correction is to reference the clocking block:

```
rtr_io.cb.din[3] <= 1'b1;
```

# Sampling Synchronous Signals

**variable = interface.cb.signal;**

- **Variable is assigned the sampled value**
  - value that the clocking block sampled at the most recent clocking event
- **Avoid non-blocking assignment**
- **Sampling of output signal is not allowed**

```
data[i] = rtr_io.cb.dout[7];
all_data = rtr_io.cb.dout;
frm_out = rtr_io.frameo_n[7];                                //error
$display("din = %b\n", rtr_io.cb.din);                         //error
if(rtr_io.cb.din[3] == 1'b0) begin ... end                  //error
```

2-20

The first error is caused by forgetting to sample via the clocking block.  
The correction is to reference the clocking block:

```
frm_out = rtr_io.cb.frameo_n[7];
```

The second and third errors are illegal sampling of output signal. Typically, this can be corrected by displaying the variable which was used to drive the signal instead of the signal itself:

```
rtr_io.cb.din <= my_data;
$display("din = %b\n", my_data);
if (my_data[3] == 1'b0) begin ... end
```

# Signal Synchronization – Level Sensitive

- Scheduling region of a clocking variable (signal) is not defined by the SystemVerilog LRM 

- Causes inconsistent behavior between programs and modules
- Causes inconsistent behavior between simulators from different vendors
- Solution: Always synchronize to clock first



- E.g. To synchronize to the low level of a clocking variable

`rtr_io.cb.frameo_n[7]`

- ALWAYS USE

```
if (rtr_io.cb.frameo_n[7] !== 1'b0)
@(rtr_io.cb iff(rtr_io.cb.frameo_n[7] === 1'b0));
```

- NEVER USE

```
wait (rtr_io.cb.frameo_n[7] === 1'b0);
```

2-21

LRM = Language Reference Manual

You could define a macro for ease of use

```
`define wait_sig_0(sig, cb) \
if (cb.sig !== 1'b0); \
@(cb iff (cb.sig === 1'b0);
```

For example you could replace the code on the slide with

```
`wait_sig_0(frameo_n[7], rtr_io.cb);
```

# Signal Synchronization – Edge Sensitive

- E.g. To synchronize to the negative edge of a clocking block signal

`rtr_io.cb.frameo_n[7]`

- ALWAYS USE

```
wait (rtr_io.cb.frameo_n[7] !== 1'b0);  
@(rtr_io.cb iff(rtr_io.cb.frameo_n[7] === 1'b0));
```

- NEVER USE

```
@(negedge rtr_io.cb.frameo_n[7]);
```



- ◆ The scheduling of this statement is not clearly defined by the SystemVerilog LRM
- ◆ This may cause inconsistent and unexpected behavior

2-22

You could define a macro for ease of use

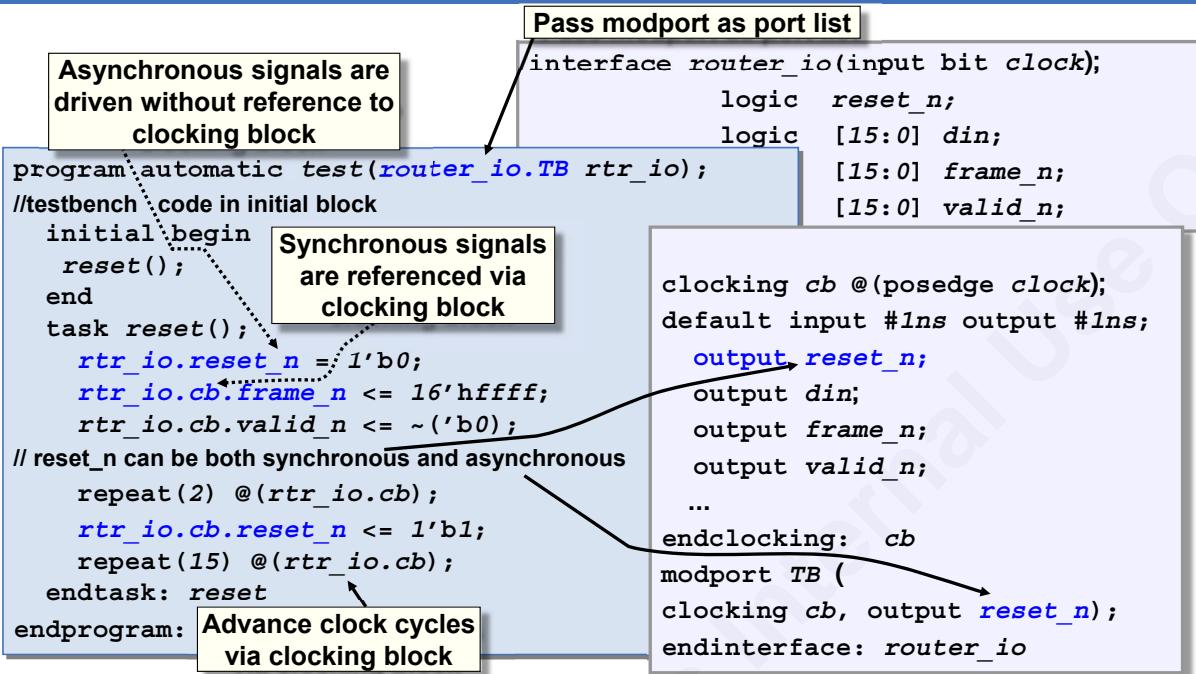
```
`define at_negedge(sig, cb) \  
wait (cb.sig !== 1'b0); \  
@(cb iff (cb.sig === 1'b0);
```

For example you could replace the code on the slide with

```
'at_negedge(frameo_n[7], rtr_io.cb);
```

A similar solution should be used for @ (posedge cb.sig)

# Using Interface in Program



2-23

In the program port list, `rtr_io` is the local reference to the instance of the interface that will be passed to it when it is instantiated. All signals in the interface now need to be referred hierarchically using this local reference.

In practice you likely will not pass any interfaces through a program port list, but instead use virtual interfaces passed to the program through other means. Virtual interfaces are discussed later in the workshop.

Users commonly forget to use the clocking block when driving or sampling synchronous signals

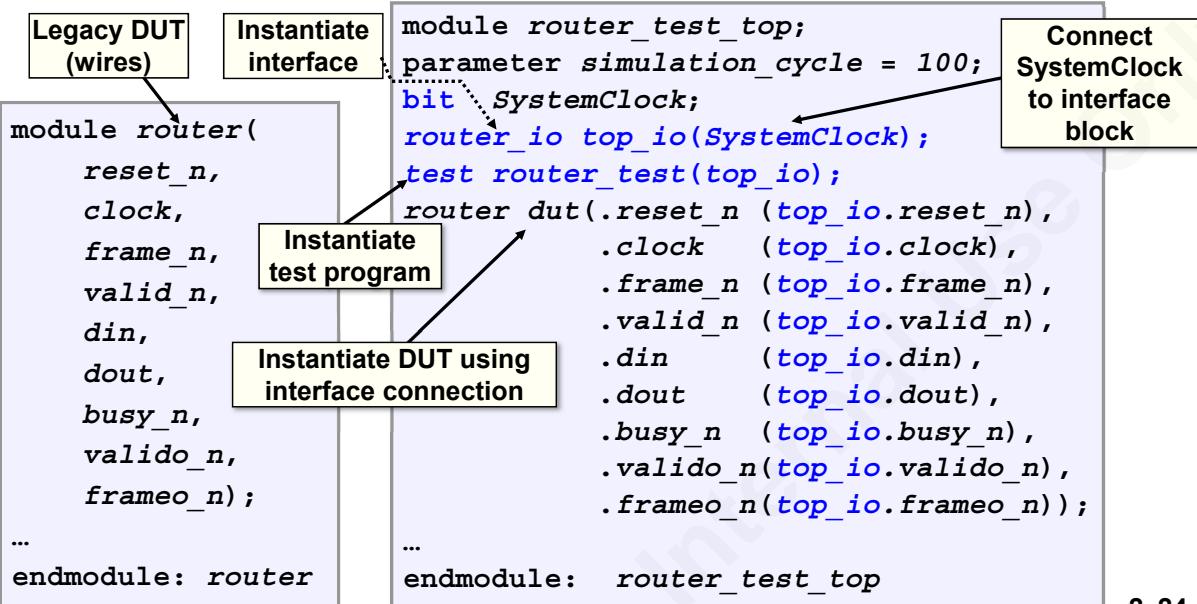
`rtr_io.frame_n` instead of `rtr_io.cb.frame_n`

This will give you an error (in VCS 2011.12 and later) like

```
Error-[SV-SNVVM] Signal not visible via modport
./router.tb.sv, 20
router_test, "router.frame_n"
Variable 'frame_n' of interface 'router_io' cannot be accessed from instance
'rtr_io' of modport 'TB'.
Please check if the signal is declared in the modport of the interface.
```

# Complete Top-Level Harness

Instantiate test program and interface in harness file



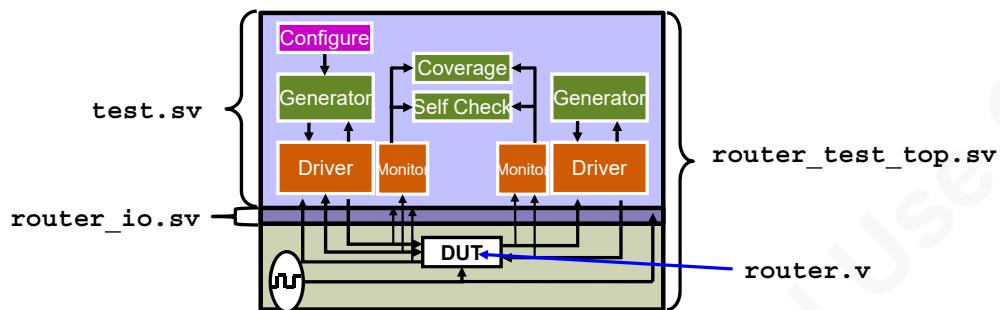
2-24

If the DUT module was already defined with a SystemVerilog interface in it's port list the DUT connection would simplify to

```
router dut (top_io); //instantiate DUT with interface
```

# Compile RTL & Simulate with VCS

- Compile HDL code: (generate simv simulation binary)



```
> vcs -sverilog [-debug] router_test_top.sv test.sv router_io.sv router.v
```

- Simulate DUT with SystemVerilog testbench

```
> ./simv
```

2-25

# SystemVerilog Run-Time Options

- Pass values from simulation command line using `+argument`
- Retrieve `+argument` value with `$value$plusargs()`

```
initial begin: proc_user_args
    int user_seed;
    if ($value$plusargs("ntb_random_seed=%d", user_seed))
        $display("User seed is %d", user_seed);
    else
        $display("Using default seed");
end: proc_user_args
```

user\_seed is 100

```
> ./simv +ntb_random_seed=100
```

- Create your own `+argument` options for simulation control and debug

2-26

In VCS the default seed is 1.

# Getting Help with VCS and SystemVerilog

## ■ VCS compiler and simulator switch summary

> **vcs -help**

## ■ Documents

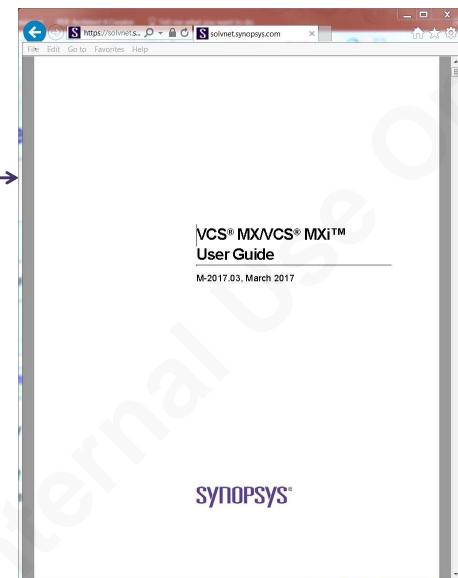
- PDF VCS docs on solvnet
  - ◆ Downloadable
- SystemVerilog LRM on IEEE
  - ◆ <http://ieeexplore.ieee.org/Xplore/guesthome.jsp>

## ■ Code Examples

- \$VCS\_HOME/doc/examples

## ■ Synopsys Support

- Email: [vcs\\_support@synopsys.com](mailto:vcs_support@synopsys.com)
- Online: <http://solvnet.synopsys.com>



2-27

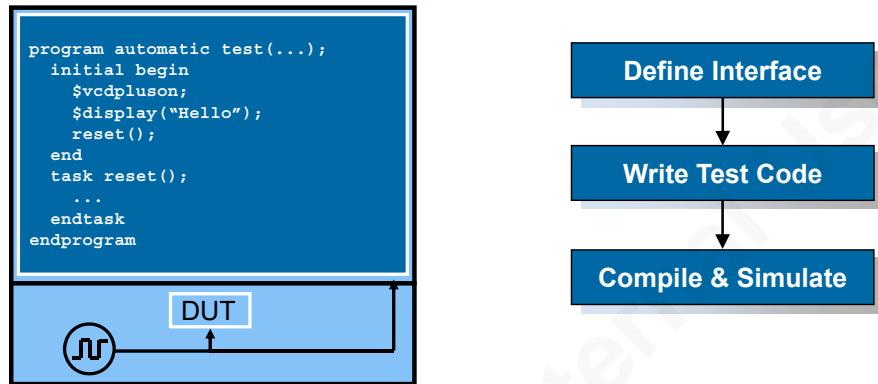
Synopsys Training Calendar and Catalog: <http://training.synopsys.com>

# Lab 1 Introduction



30 minutes

## SystemVerilog Verification Flow



2-28

## Unit Objectives Review

**Having completed this unit, you should be able to:**

- **Describe the process of reaching verification goals**
- **Describe components of a SystemVerilog testbench environment**
- **Describe program and interface constructs**
- **Compile and simulate a SV testbench**
- **Drive and sample DUT signals**
- **Synchronize to known point in simulation**



2-29

## **Appendix**

**Cycle Delay, Default Clocking, Synchronous Drive**

**Useful VCS Compile and Run Time Switches**

**External Binding of Components**

**Interactive Debugging with DVE & Verdi**

**2-30**

## Cycle Delay, Default Clocking, Synchronous Drive

2-31

# Default clocking and Cycle Delay

- One clocking block can be specified as the default within a given module, interface or program

```
default clocking rtr_io.cb;
```

- Not Recommended
  - ◆ Potential for errors when testbench has multiple clocks
  - ◆ Delay not obvious when debugging



- Cycle Delay

- ## operator to indicate number of cycles of default clocking to delay
- Cycle delay operator can be used
  - ◆ Standalone
  - ◆ In a synchronous drive (next slide)

```
// standalone
##4;           //wait 4 cycles using default clocking
##(k + 2);    //wait k+2 cycles of default clocking
```

2-32

`##2 rtr_io.cb.din <= var_a; next_statement;` is similar to:

`#2 rtr_io.cb.din <= var_a; next_statement;`

In that, time advances first, `var_a` is then evaluated, the assignment is scheduled for the non-blocking region (+output skew) and `next_statement` is executed. However a default clocking **must** be defined.

# Cycle Delay and Synchronous Drive

```
##num1 interface.cb.signal <= <value | expression>;
```

- **##num1 specifies default clocking cycle delays**

- A **default clocking** must be defined in the enclosing module, interface or program
- Does not use the *interface.cb* clock
- Execution of statement is **blocked** (delayed)

```
interface.cb.signal <= ##num2 <value | expression>;
```

- **##num2 specifies cycle delays of the *interface.cb* clocking block for the synchronous drive**

- Execution of statement is **not blocked**

2-33

`rtr_io.cb.din <= ##2 var_a; next_statement;` is similar to:

`rtr_io.cb.din <= #2 var_a; next_statement;`

In that, `var_a` is evaluated immediately, the assignment is scheduled for the second valid clock edge from current simulation time (+output skew) and `next_statement` is executed without time advancing.

## Cycle Delay Example

Example:

⚠ These cycle delays use default clocking

Current clock edge

```
default clocking rtr_io.cb;
initial begin
...
##1; rtr_io.cb.din[3] <= 1'b1;
var_a = var_b;
##3 rtr_io.cb.din[3] <= var_b;
...
end
```

clock rtr\_io.cb

assume var\_b is changed by another thread

variables

var\_a = var\_b executed after one clock cycle

var\_b

var\_a

din[3]

1\* cycle      3\* cycles

signal driven after delay

\* default clocking

2-34

It is strongly recommended that you not use default clocking to introduce cycle delays. You can always delay for clocks using the @ construct in Systemverilog.

```
default clocking rtr_io.cb;
##2 rtr_io.cb.din <= var_a;
```

can be rewritten as

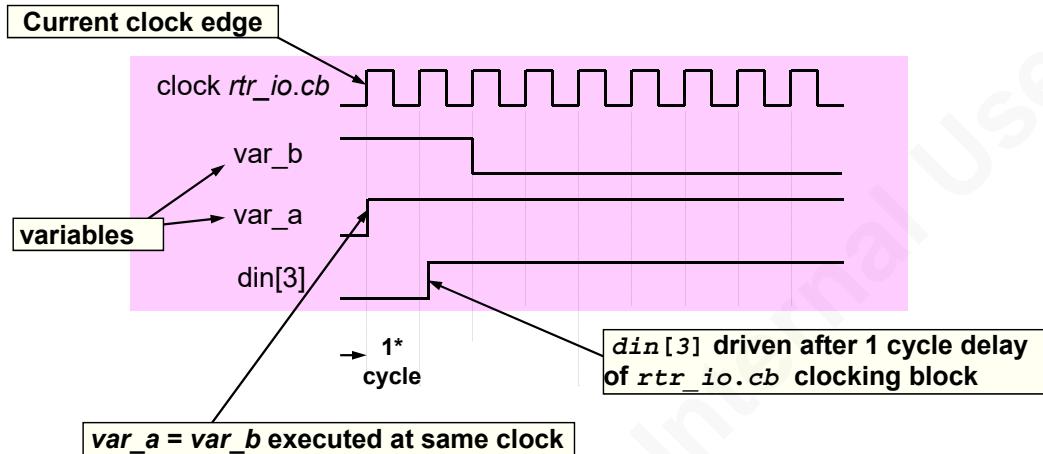
```
repeat(2) @(rtr_io.cb); // the ; is required
rtr_io.cb.din <= var_a;
```

From SystemVerilog 2012 LRM (Section 14-11):

Cycle delays of ##0 are treated specially. If a clocking event has not yet occurred in the current time step, a ##0 cycle delay shall suspend the calling process until the clocking event occurs. When a process executes a ##0 cycle delay and the associated clocking event has already occurred in the current time step, the process shall continue execution without suspension. When used on the right-hand side of a synchronous drive, a ##0 cycle delay shall have no effect, as if it were not present.

## Synchronous Drive Example

**Example:**    `rtr_io.cb.din[3] <= ##1 1'b1;`  
              `var_a = var_b;`



2-35

## **Useful VCS Compile and Run Time Switches**

**2-36**

# Compiling and Running with VCS

## ■ Compile:

```
vcs -sverilog -debug top.sv test.sv dut.sv
  ◆ -sverilog      Enable SystemVerilog constructs
  ◆ -debug        Enable debug except line stepping
  ◆ -debug_all    Enable debug including line stepping
  ◆ -lca          Enable LCA features (see release note)
```

## ■ Run:

See the VCS User Guide for all options

```
simv +user_tb_runtime_options
  ◆ -l logfile      Create log file
  ◆ -gui            Run GUI
  ◆ -ucli           Run with new command line debugger
  ◆ -i cmd.key      Execute UCLI commands
```

## Compiling with VCS – Legacy Verilog Code (1/2)

- SystemVerilog has dozens of new reserved keywords
  - e.g. bit, packed, logic
  - These may conflict with existing Verilog code
- Keep Verilog-2001 code separate from SystemVerilog code compile with:

```
vcs -sverilog new.v +verilog2001ext+.v2k legacy.v2k
```

- or

```
vcs +systemverilogext+.sv legacy.v new.sv
```

```
// Old Verilog-1995/2001 legacy code
integer bit, count;
initial begin
    count = 0;
    for (bit = 0; bit < 8; bit = bit + 1)
        if (adrs[bit] === 1'bx) count = count + 1;
end
```

2-38

## Compiling with VCS – Legacy Verilog Code (2/2)

- New keywords in new LRM revisions cause identifiers in legacy code to be invalid
  - VCS releases 2014.12 and later provide command line control with **-sv** option
    - ◆ **-sv=<version>**, where version is 2005, 2009, 2012 etc.
    - ◆ Enables SystemVerilog mode just like the **-sverilog** option
    - ◆ Specifying both **-sv** and **-sverilog** together will be an error
    - ◆ Controls only keyword set. Does not modify behavior
  - Use model
    - ◆ Two-step flow e.g.
      - **% vcs -sv=2009 <all files>**
    - ◆ Three-step flow e.g.
      - **% vlogan -sv=2005 <some files>**
      - % vlogan -sv=2009 <other files>**
      - % vcs -q <top\_module>**

2-39

## External Binding of Components

2-40

# External Binding of Components

- Components can be instantiated using bind

```
module router_test_top;
    parameter simulation_cycle = 100;
    reg SystemClock;
    router_io top_io(SystemClock);
    test t(top_io);
    router dut( //device under test
        .reset_n (top_io.reset_n),
        ...
        .frameo_n (top_io.frameo_n)
    );
    ...
endmodule
```

instance replaced  
by bind

XMR to top module  
interface instance

```
bind router_test_top test t(router_test_top.top_io);
```

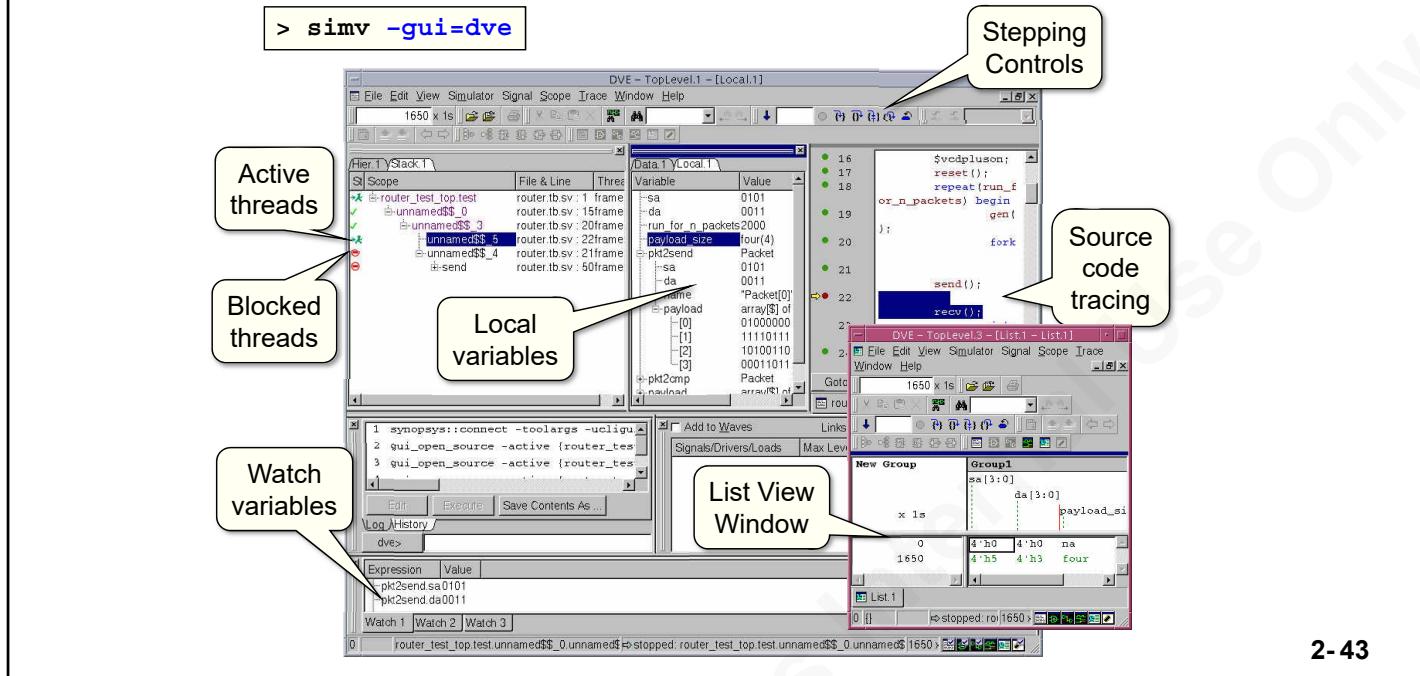
2-41

**bind** statements can be placed inside modules, interfaces or be independent statements in a compilation-unit scope.

## Interactive Debugging with DVE & Verdi

2-42

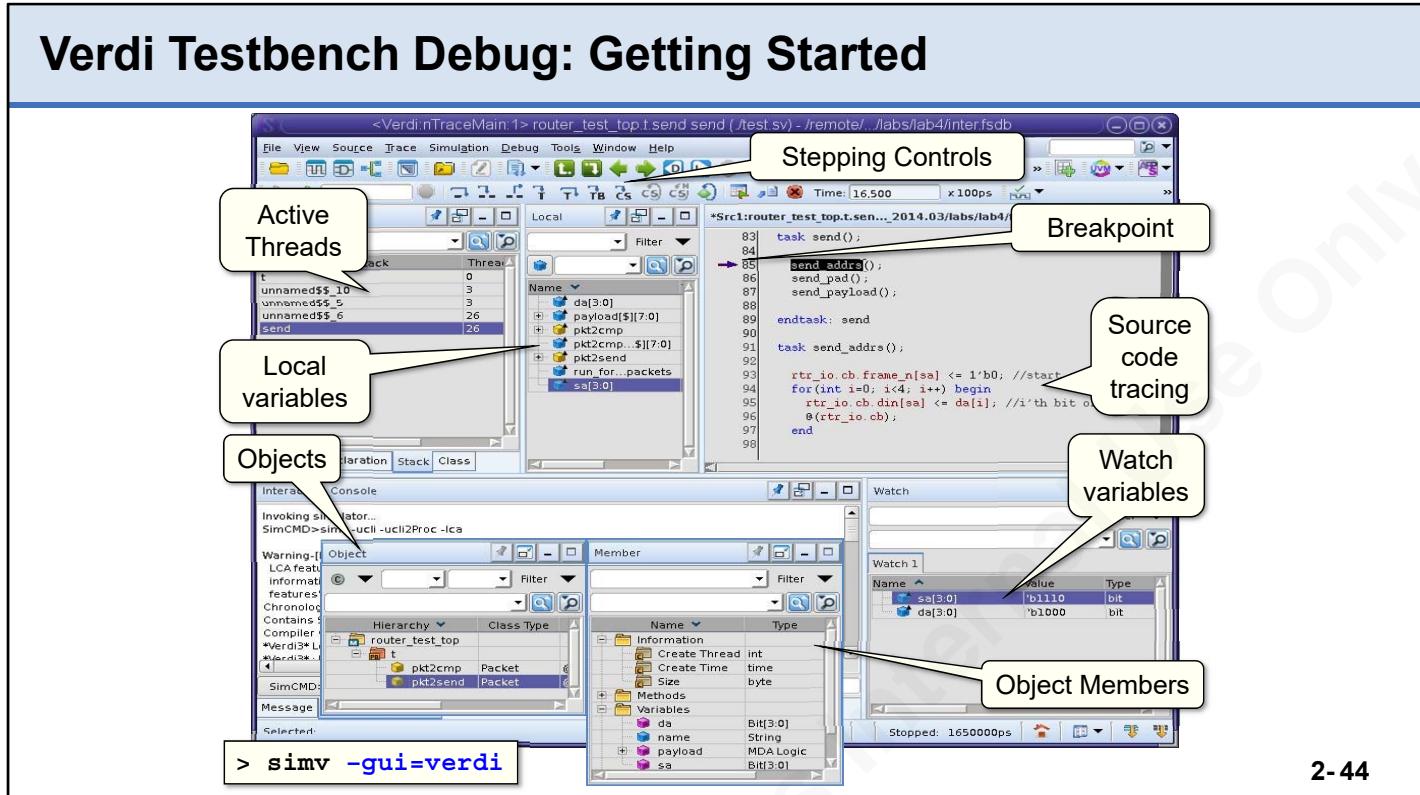
# DVE Testbench Debug: Getting Started



2-43

Not all windows show up by default at simulator invocation.

# Verdi Testbench Debug: Getting Started



Not default view. Some windows undocked for clarity. Verdi3 is not included with VCS by default. It is licensed separately by Synopsys.

# Agenda

DAY  
1

1 The Device Under Test (DUT)

2 SystemVerilog Verification Environment 

3 SystemVerilog Language Basics - 1

4 SystemVerilog Language Basics - 2 

## Unit Objectives



**After completing this unit, you should be able to:**

- Define the structure of a SystemVerilog(SV) program
- Declare variables and understand scope of variables in a SV program
- Define and use arrays in a SV program

3-2

# SystemVerilog Testbench Code Structure

## ■ Test code is embedded inside program block

- **program** is instantiated in the top-level harness file

```
// compile unit scope variables
`include <files>
program [automatic] name(interface);
`include <files>
// program global scope variables
initial begin
// local scope variables
// top-level test code
end
task task_name(...);
// task local scope variables
// task code
endtask
endprogram
```

```
program automatic test ( ... );
initial begin
$vcpluspluson;
reset()
end
task reset();
module router_test_top;
router io top_io(SystemClock);
test t(top_io);
router dut(.reset_n(top_io.reset_n),
.clock (top_io.clock),
.... (top_io...));
...;
endmodule
```

From Lab 1:

3-3

The program block serves three basic purposes:

- It provides an entry point to the execution of testbenches.
- It creates a scope that encapsulates program-wide data, tasks, and functions.
- It provides a syntactic context that specifies scheduling in the Reactive region.

A program block may not contain instances of other modules, programs or **always** blocks. The intent of program blocks is to create a testbench not model or emulate hardware. Program blocks may only use initial blocks for procedural code and use Object Oriented Programming.

A program block should always have an automatic (or dynamic) lifetime, since the intent is to emulate a programming language like C++, for example. By default programs in SystemVerilog programs have a static lifetime, meaning all variables defined are **static** (allocated at compile time). This was done to maintain backward compatibility with Verilog modules which have a static lifetime. However for all practical purposes programs need dynamic variables. This requires that you make the program "automatic" by adding the **automatic** keyword as shown.

Unlike C++, variables can be declared and optionally initialized only in the declarative region of a scope which is before any procedural code within the scope. For example the following code is illegal.

```
initial begin
int i;
i = 10; //procedural code
int j; //cannot declare new variable here
j = 20;
end
```

# SystemVerilog Lexical Convention

## ■ Same as Verilog

- Case sensitive identifiers (names)
- White spaces are ignored except within strings
- Comments:
  - ◆ `// ...`
  - ◆ `/* ... */`  
(Do not nest! As in `/* /* */ */`)

## • Number format:

```
<size>'<base><number>
'b (binary) : [01xXzZ]
'd (decimal) : [0123456789]
'o (octal) : [01234567xXzZ]
'h (hexadecimal)
: [0123456789abcdefABCDEFxXzZ]
◆ Can be padded with '_' (underscore)
for readability:
16'b_1100_1011_1010_0010
32'h_beef_cafe
```

3-4

## 2-State (0|1) Data Types (1/3)

`bit [msb:lsb] var_name [=initial_value];`

- Better compiler optimizations get better performance
- Variable initialized to '0 if initial\_value is not specified
  - '0 is unsized literal (See note)
- Assigned 0 for x or z value assignments
  - Sized as specified
  - Defaults to unsigned

```
bit flag;
bit[15:0] sample, temp = 16'hdeed;
bit[7:0] a = 8'b1;           // 8'b0000_0001
bit[7:0] b = 'b1;           // 8'b0000_0001
bit[7:0] c = '1;            // 8'b1111_1111
bit[31:0] signed ref_data = -155;
```

3-5

Explicit 2-state variables allow more compiler optimizations, giving better performance.

BUT – they will not propagate X or Z, so keep away from DUT.

SystemVerilog adds the ability to specify unsized literal single-bit values with a preceding apostrophe ('), but without the base specifier. All bits of the unsized value are set to the value specified. Supported unsized literals are '0, '1, 'x, 'z

## 2-State (0|1) Data Types (2/3)

`2-state-type variable_name [=initial_value];`

### ■ Sized integral 2-state data types:

- `byte` - 8-bit signed data type
- `shortint` - 16-bit signed data type
- `int` - 32-bit signed data type
- `longint` - 64-bit signed data type

```
shortint temp = 256;
int sample, ref_data = -9876;
longint a, b;
longint unsigned testdata;
```

## 2-State (0|1) Data Types (3/3)

```
2-state-type variable_name = [initial_value];
```

### ■ Real 2-state data types:

- **real** - Equivalent to **double** in C
- **shortreal** - Equivalent to **float** in C
- **realtime**
  - ◆ 64-bit real variable for use with **\$realtime**
  - ◆ Can be used interchangeably with **real** variables

```
real alpha = 100.3, cov_result;
realtime t64;
#100 t64 = $realtime;
cov_result = $get_coverage();
if (cov_result == 100.0) ...;
```

3-7

## 4-State (0|1|X|Z) Data Types (1/2)

```
reg | logic [msb:lsb] variable_name [=initial_value];
```

- **Variables must be 4-state to emulate correct hardware behavior in simulation**
  - `reg` and `logic` are synonyms
  - Used to drive/store DUT interface signals in testbench
  - Initialized to '`x`' if `initial_value` is not specified
    - ◆ '`x`' is unsized literal
  - Can be used in continuous assignment (single driver only), unlike Verilog
  - Can be used as outputs of modules
  - Defaults to `unsigned`

```
logic[15:0] sample = '1, ref_data = 'x;  
assign sample = rtr_io.cb.dout;
```

3-8

A logic/reg should only have a single continuous assignment driver. For multiple drivers, use `wire`.

## 4-State Data Types (2/2)

- Sized 4-state data types:

```
integer variable_name [=initial_value];
```

- 32-bit signed data type

```
time variable_name [=initial_value];
```

- 64-bit unsigned data type

```
integer a = -100, b;  
time current_time;  
b = -a;  
current_time = $time;  
if (current_time >= 100ms) ...;
```

Can use time units in SystemVerilog

## String Data Type

```
string variable_name [=initial_value];
```

- Defaults to empty string ""
- Can be created with \$sformatf() system function
- Built-in operators and methods:
  - ==, !=, compare() and icompare()
  - itoa(), atoi(), atohex(), toupper(), tolower(), etc.
  - len(), getc(), putc(), substr() (See SystemVerilog LRM for more)

```
string name, s = "Now is the time";
for (int i=0; i<4; i++) begin
    name = $sformatf("string%0d", i);
    $display("%s, upper: %s", name, name.toupper());
end
s.putc(s.len()-1, s.getc(5)); // change e to s
$display(s.substr(s.len()-4, s.len()-1));
```

3-10

The resulting print out on terminal is:

```
string0, upper: STRING0
string1, upper: STRING1
string2, upper: STRING2
string3, upper: STRING3
tims
```



LRM: Language Reference Manual

# Enumerated Data Types

## ■ Define enumerated type

```
typedef enum [data_type] {named constants} enumtype;
```

Type declaration

## ■ Declare enum variables

```
enumtype var_name [=initial_value];
```

- Data type defaults to `int`

Variable declaration

- Variable initialized to '`0`' if `initial_value` is not specified

- `enum` can be displayed as ASCII with `name()` function or `%p` format specification

```
typedef enum bit[2:0] {IDLE=1, TEST, START} state_e;  
state_e current, next = IDLE;  
$display("current = %0d, next = %s", current, next.name());  
$display("next = %p", next); //can use .name() or %p for ASCII
```

What will be displayed on screen?

3-11

What's printed to screen:

**current = 0, next = IDLE**  
**next = IDLE**

The reason `current` is printed as `0` is because enum variables of 2-state types default to '`0`' if not initialized or assigned an enum value. 4-state enum variables default to '`X`'.

Recommendation: Always initialize enumerated variables.

Enumerated variables maybe declared directly but this is not very useful.

```
enum bit[2:0] {S0='b001, S1='b010, S2='b100} st;
```

Useful methods available for enumerated variables include:

```
first()  
last()  
prev()  
next()  
num()  
name()
```

## Data Arrays – Fixed-size Arrays (1/4)

```
type array_name[size] [=initial_value];  
  
integer numbers[5]; // array of 5 integers, indexed 0 – 4  
int b[2] = '{3,7}; // ( b[0] = 3, b[1] = 7)  
  
bit[31:0] c[2][3] = '{ {3,7,1}, {5,1,9} }; // multi-dimensional  
byte d[7][2] = '{default:-1}; // all elements set = -1  
  
bit[31:0] a[2][3] = c; // array copy – types and sizes must be same  
for(int i=0; i<$dimensions(a), i++)  
    $display($size(a, i+1)); // 2 3 32  
  
$size returns size of particular dimension  
$dimensions returns number of dimensions  
  
See more array-querying functions in the Appendix
```

3-12

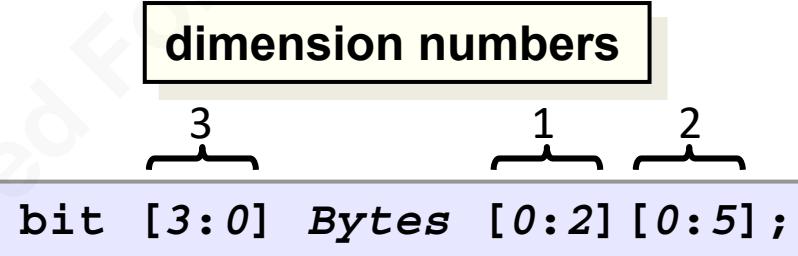
Use a fixed-size array when you know the size of the array at compile time, and the array size never changes.

\$dimensions (array\_name)

Returns the # of dimensions in the array array\_name

\$size (array\_name, dimension)

Returns the total # of elements in specified dimension from (\$high - \$low + 1) of array\_name



packed dimension

same as:

unpacked dimensions

bit [3:0] Bytes [3][6];

## Data Arrays – Dynamic Arrays (2/4)

```
type array_name[] [=initial_value];
```

- Array size allocated at runtime with constructor

```
logic[7:0] ID[], array1[] = new[16];
logic[7:0] data_array[], mdim[] [];

ID = new[100]; // allocate memory
data_array = new[ID.size()] (ID); // copy - types must match
data_array = ID; // copy - types must match

ID = new[ID.size() * 2] (ID); // double the size of ID
ID = data_array; // ID resized to match data_array
data_array.delete(); // de-allocate memory
```

Returns size of array

3-13

Use a dynamic array when you won't know the array size until run time (random stimulus), but the array size does not change. Allocating and copying the array is expensive.

Memory for each dimension of a multidimensional dynamic array has to be allocated separately.

```
reg[7:0] mdim[] []; //two dimensional dynamic array

mdim = new[4]; //allocate first dimension
foreach(mdim[i])
    mdim[i] = new [i+4]; //allocate second dimension
```

Using `size()` you can only access the size of the first dimension for all variable sized arrays.

Dynamic arrays and queues may be copied to each other only if the data types match. Dynamic arrays may be copied to fixed arrays as long as their data types and sizes match. A fixed array can be copied to a dynamic array of the same data type.

## Data Arrays – Queues (3/4)

```
type array_name[$[:bound]] [=initial_value];
```

- Array memory allocated and de-allocated at runtime with
  - `push_back()`, `push_front()`, `insert()`
  - `pop_back()`, `pop_front()`, `delete()`
- Can not be allocated with `new[]`
  - `bit[7:0] ID[$] = new[16];` // Compilation error!
- Index 0 refers to lower (first) index in queue
- Index \$ refers to upper (last) index in queue
- Can be operated on as an array, FIFO or stack
- Optional bound in declaration is last index

3-14

Use a queue when the array size is constantly changing at run time, such as a data queue in a scoreboard. Adding and removing elements at the head and tail is fast, but expensive for elements in the middle.

## Queue Manipulation Examples

```
int j = 2;
int q[$] = {0,1,3,6}; // note no '
int b[$] = {4,5};      // note no '
q.insert(2, j);        // {0,1,2,3,6}
q.insert(4, b);        // {0,1,2,3,4,5,6}
q.delete(1);           // {0,2,3,4,5,6}
q.push_front(7);       // {7,0,2,3,4,5,6}
j = q.pop_back();      // {7,0,2,3,4,5}   j = 6
q.push_back(8);        // {7,0,2,3,4,5,8}
$display(q.size());    // 7
$display("%p", q);    // '{7,0,2,3,4,5,8}
q.delete();            // delete all elements
$display(q.size());    // 0
```

3-15

`%p` operator allows formatted display of an array. The format is a tool default.  
An array literal starts with a '`,`' , while a queue literal does not

## Data Arrays – Associative Arrays (4/4)

```
type array_name[index_type]; // indexed by specified type
```

- Index type can be any numerical, string or class type
- Dynamically allocated and de-allocated

```
integer ID_array[bit[15:0]];
ID_array[71] = 99;           // allocate memory
ID_array.delete(71);        // de-allocate one element
ID_array.delete();          // de-allocate all elements
```

- Array can be traversed with:
  - first(), next(), prev(), last()
- Number of allocated elements can be determined with call to num()
- Existence of a valid index can be determined with call to exists()

3-16

```
function int num();
```

Returns the number of entries in the array. If the array is empty, it returns 0:

```
function void delete( [input index] );
```

Deletes a specified element or the entire array.

```
function int exists( input index );
```

Returns 1 if element exists, otherwise returns 0:

```
function int first( ref index );
```

Returns the index of the first valid index of an array via a pass by reference int variable. If the array contains at least one element, the function return value will be 1 and the argument variable will be modified to reference the first valid index. If the array is empty, the function return value will be 0 and the argument variable is left unmodified:

```
function int last( ref index );
```

Similar to first except the last valid index is returned instead of first:

```
function int next( ref index );
```

Returns the index of the next valid element. If there is a next valid element after the index argument, the argument variable will be modified to contain the index of the next valid element and the return value of the function will be 1. Otherwise, the argument variable is left alone and the return value of the function will be 0:

```
function int prev( ref index );
```

Similar to next except the prev valid index is returned:

## Associative Array Examples

```
byte opcode[string], t[int], a[int]; int index;  
opcode["ADD"] = -8; // create index "ADD" memory  
for (int i=0; i<10; i++)  
    t[1<<i] = i; // create 10 array elements  
a = t; // array copy  
  
$display("num of elements in t is: %0d", t.num());  
  
//process each element  
if (t.first(index)) begin // locate first valid index  
    $display("t[%0d] = %0d", index, t[index]);  
    while(t.next(index)) // locate next valid index  
        $display("t[%0d] = %0d", index, t[index]);  
end  
//better to use foreach shown on next slide
```

3-17

Compile and simulate the above code, you will see something like the following result:

**num of elements in t array is: 10**

t[1] = 0  
t[2] = 1  
t[4] = 2  
t[8] = 3  
t[16] = 4  
t[32] = 5  
t[64] = 6  
t[128] = 7  
t[256] = 8  
t[512] = 9

# Array Loop Support and Reduction Operators

## ■ Loop: foreach

- Supports all array types

```
int data[] = '{1,2,3,4,5,6,7}, qd[$][];  
qd.push_back(data);  
foreach(data[i])  
    $display("data[%0d] = %0d", i, data[i]);  
//foreach(qd[i,j]) // to loop through 2-dimensional array
```

## ■ Reduction operators

```
$display("sum of array content = %0d", data.sum());  
$display("product value is = %0d", data.product());  
$display("and'ed value is = %0d", data.and());  
$display("or'ed value is = %0d", data.or());  
$display("xor'ed value is = %0d", data.xor());
```

3-18

Compile and simulate the above code, you will see something like the following result:

```
data[0] = 1  
data[1] = 2  
data[2] = 3  
data[3] = 4  
data[4] = 5  
data[5] = 6  
data[6] = 7  
sum of array content = 28  
product value is = 5040  
and'ed value is = 0  
or'ed value is = 7  
xor'ed value is = 0
```

## Array Methods (1/4)

```
function array_type[$] array.find() with (expression)
```

- Finds all the elements satisfying the **with** expression
- Matching elements are returned as a queue

```
function int_or_index_type[$] array.find_index()  
    with (expression)
```

- Finds all the indices satisfying the **with** expression
- Matching indices are returned as a queue

■ **item** references the array elements during search

■ Empty queue is returned when match fails

3-19

The `find()` and `find_index()` methods can take a user defined iterator for use in the expression. See example on next slide.

```
find(user_iter) with (expr); //expr uses user_iter to refer to value in the array element.
```

The expressions used by array manipulation methods sometimes need the actual array indices at each iteration. The `index` method of an iterator returns the index value of the specified dimension.

The return type of the `index` method is an `int` for all array iterator items except associative arrays, which return an index of the same type as the associative index type. Associative arrays that specify a wildcard index type are not allowed.

For example:

```
int arr[];  
int q[$];  
...  
// find all items equal to their position (index)  
q = arr.find with ( item == item.index );
```

## Array Methods (2/4)

- Example: `find()` and `find_index()`

```
program automatic test;
    bit[7:0] SQ_array[$] = {2, 1, 8, 3, 5};
    bit[7:0] SQ[$];
    int idx[$];

    initial begin
        SQ = SQ_array.find() with ( item > 3 );
        // SQ[$] contains 8, 5 - item is default iterator variable

        idx = SQ_array.find_index(addr) with ( addr > 3 );
        // idx[$] contains 2, 4 - addr is user defined iterator variable
    end
endprogram: test
```

3-20

The `find()` example is equivalent to

```
foreach (SQ_array[i]) begin
    item = SQ_array[i]; //item is default iterator
    if ( item > 3 )
        SQ.push_back(item);
end
```

The `find_index(addr)` example is equivalent to.

```
foreach (SQ_array[i]) begin
    addr = SQ_array[i]; //addr is user defined iterator
    if ( addr > 3 )
        idx.push_back(i);
end
```

## Array Methods (3/4)

```
function array_type[$] array.find_first() with ([expr] | 1)
```

- First element satisfying the `with` expression is returned in `array_type[0]`

```
function int_or_index_type[$]
```

```
array.find_first_index() with ([expr] | 1)
```

- First index satisfying the `with` expression is returned in `int_or_index_type[0]`

- **Always returns a queue of one or zero elements**

- Empty queue is returned when match fails

- **If `with` expression is 1, first element or index is returned**

- **`with` is mandatory for both methods**

- `item` in expression references array element during search

3-21

## Array Methods (4/4)

- Example: `find_first()` and `find_first_index()`

```
program automatic test;
  int array[] = new[5];
  int idx[$], val[$], dyn_2d[][][], mixed_2d[$][][];
  initial begin
    foreach(array[i])
      array[i] = 4 - i;
    val = array.find_first() with ( item > 3 ); // val[0] == 4
    idx = array.find_first_index() with ( item < 0 ); // idx == {};
  end
endprogram: test
```

More array methods available - check LRM

3-22

Examples of other array methods: (please check release note for supported methods in VCS)

```
find_last()
find_last_index()
unique()
unique_index()
min()
max()
reverse()
sort()
rsort()
shuffle()
```

# Data Arrays – Out-of-Bounds Access

- Multiple dimensions are supported for all unpacked array types

- Can be heterogenous
  - ◆ e.g. int `dada[] []`, bit[3:0] `q_aa[$] [string]`

- For all unpacked array types – in their first (lowest) dimension

- Out-of-bounds write is ignored except for

```
int addr[$:4] = {0,1,2,3,4}; addr.push_back(10); addr[0] = addr[5];
```

- Associative arrays (which cannot be bounded)
- Bounded queues - warning issued for out-of-bounds write

- Out-of-bounds read returns '0 for 2-state, 'x for 4-state arrays

- For all unpacked array types – in their dimensions greater than 1

- Out-of-bounds access results in simulation error



3-23

Out-of-bound reads and writes for the lowest dimension will not be flagged by VCS (except as noted). To flag these errors and stop simulation use

**-ntb\_opts keyword\_argument** at compile time

The keyword arguments are as follows:

**check**

Does a bounds check on dynamic type arrays (dynamic, queues only) and issues an error at runtime.

**check=dynamic**

Same as check. Does a bounds check on dynamic type arrays (dynamic, queues) and issues an error at runtime.

**check=fixed**

Does a bounds check only on fixed size arrays and issues an error at runtime.

**check=all**

Does a bounds check on both fixed size and dynamic type arrays (dynamic, queues) and issues errors at runtime.

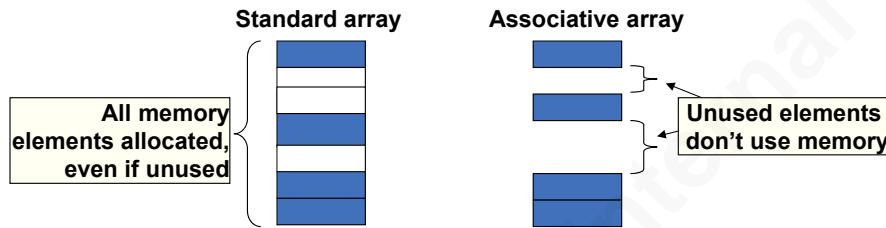
The following error message is displayed during runtime:

**- ERROR-[DT-OBAE] Out-of-bound access**

# Array Summary

Type	Memory	Index	Example (performance)
Fixed Size	Allocated at compile, unchangeable afterwards	Numerical	int <i>addr</i> [5]; (fast)
Dynamic	Allocated at run-time, changeable at run-time	Numerical	logic <i>flags</i> []; (fast)
Queue	Push/pop/copy at run-time to change size	Numerical	int <i>in_use</i> []; (fast)
Associative	Write at run-time to allocate memory	Typed*	state <i>a</i> [string]; (moderate)

\* The index of associative arrays should always be typed



3-24

## Quiz Time ?

3-25

## Quiz 1

- Is the code below legal? Will it compile?

```
program automatic test;
bit [31:0] count; logic [31:0] Count = 'x;
initial begin
    count = Count;
    $display("Count = %0x count = %0d", Count, count);
end
endprogram: test
```

- What type is type logic a synonym of? What does the 'x initialize Count to?
- What will the program display? Why is value of count different from Count?

3-26

count is a 2-state variable. Unknown values assigned to it become 0.

Count = xxxxxxxx count = 0

Type Logic is a synonym for type reg. The x is an unsized assignment. All bits of Count become x.

The program is legal. It will compile.

## Quiz 2

- Define three types of arrays
  - Fixed array of size 1024
  - Dynamic array of size 1024
  - Associative array with an `int` type index
- Write to three locations in each array
  - 0, 500, 1023
- How many elements has each of these allocated after the write operation?
  - Fixed array –
  - Dynamic array –
  - Associative array uses –

3-27

array and `dynamic` allocate 1024 memory locations each.  
`associative` allocates only the three accessed locations.

```
program automatic test;
    int assoc_a[int];
    int fararray[1024];
    int darray[] = new[1024];
    int fararray[1023] = 0;
    int darray[500] = 200;
    int fararray[500] = 100;
    int assoc_a[0] = 0;
    int assoc_a[500] = 500;
    int assoc_a[1023] = 1023;
endprogram
```

## Unit Objectives Review

**Having completed this unit, you should be able to:**

- Define the structure of a SystemVerilog(SV) program
- Declare variables and understand scope of variables in a SV program
- Define and use arrays in a SV program



3-28

# **Appendix**

## **Unpacked Array Performance**

## **Advanced SystemVerilog Constructs**

Packed Array

Struct

Union

## **Streaming Operators: Pack/Unpack**

**3-29**

## Unpacked Array Performance

3-30

## Unpacked Array Performance

Array Type	Application	Performance
Fixed-Size	Use in RTL for FIFO, Memory, Buffer. Use when size of array is known and fixed for duration of simulation.	Best
Dynamic	Use this whenever you need random read/write access to any element of the variable sized array.	Good
Queue	Use for stack, CAM applications, Scoreboard queues.	Good
Associated	Very useful for sparse memory applications. Use when creating hash tables.	Moderate

3-31

The three array types other than associated arrays give similar fast performance for reads and writes.

## Packed Array

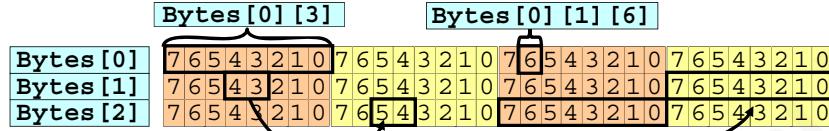
3-32

# Packed Array

- Defines a packed array structure

```
type [msb:lsb] [msb:lsb] name [constant];
```

```
bit [3:0] [7:0] Bytes [3]; // 3 entries of packed 4 bytes
```

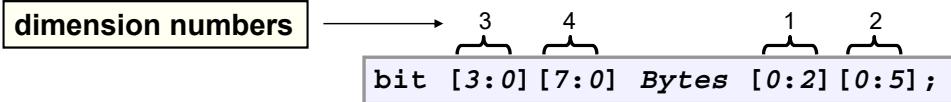


```
Bytes [2] [2] [5:4] = Bytes [1] [3] [4:3]; Bytes [1] [0] = Bytes [2] [1];
```

```
Bytes [2] = 32'hbeef_deed;
```

```
Bytes [2] 101111101110111110101111011101101101
```

# Array Querying System Functions



\$dimensions (array_name)	Returns the # of dimensions in the array
\$left (array_name, dimension)	Returns MSB of specified dimension
\$right (array_name, dimension)	Returns LSB of specified dimension
\$low (array_name, dimension)	Returns the min of \$left and \$right
\$high (array_name, dimension)	Returns the max of \$left and \$right
\$increment (array_name, dimension)	returns 1 if: \$left is >= \$right returns -1 if: \$left is < \$right
\$size (array_name, dimension)	Returns the total # of elements in the specified dimension (\$high - \$low +1)

3-34

## Array Querying System Functions Examples

```
int c[2], a[2][2];
bit[31:0] b[0:2][0:1]={ {3,7}, {5,1}, {0,4} };
$display($dimensions(a));
for (int i=1; i<=$dimensions(a); i++) begin
    $display("a dimension %0d size is %0d", i, $size(a, i));
    $display("a dimension %0d left is %0d", i, $left(a, i));
    $display("a dimension %0d right is %0d", i, $right(a, i));
    $display("a dimension %0d low is %0d", i, $low(a, i));
    $display("a dimension %0d high is %0d", i, $high(a, i));
    $display("a dimension %0d increment is %0d", i,
        signed'($increment(a, i)));
end
$display($dimensions(b));
for (int i=1; i<=$dimensions(b); i++) begin
    $display("b dimension %0d size is %0d", i, $size(b, i));
    $display("b dimension %0d left is %0d", i, $left(b, i));
    $display("b dimension %0d right is %0d", i, $right(b, i));
    $display("b dimension %0d low is %0d", i, $low(b, i));
    $display("b dimension %0d high is %0d", i, $high(b, i));
    $display("b dimension %0d increment is %0d", i,
        signed'($increment(b, i)));
end
```

See Notes for output

3-35

3

a dimension 1 size is 2  
a dimension 1 left is 0  
a dimension 1 right is 1  
a dimension 1 low is 0  
a dimension 1 high is 1  
a dimension 1 increment is -1  
a dimension 2 size is 2  
a dimension 2 left is 0  
a dimension 2 right is 1  
a dimension 2 low is 0  
a dimension 2 high is 1  
a dimension 2 increment is -1  
a dimension 3 size is X  
a dimension 3 left is X  
a dimension 3 right is X  
a dimension 3 low is X  
a dimension 3 high is X  
a dimension 3 increment is X

3

b dimension 1 size is 3  
b dimension 1 left is 0  
b dimension 1 right is 2  
b dimension 1 low is 0  
b dimension 1 high is 2  
b dimension 1 increment is -1  
b dimension 2 size is 2  
b dimension 2 left is 0  
b dimension 2 right is 1  
b dimension 2 low is 0  
b dimension 2 high is 1  
b dimension 2 increment is -1  
b dimension 3 size is 32  
b dimension 3 left is 31  
b dimension 3 right is 0  
b dimension 3 low is 0  
b dimension 3 high is 31  
b dimension 3 increment is 1

**struct**

**3-36**

## struct - Data Structure

- Defines a wrapper for a set of variables

- Similar to C `struct` or VHDL `record`
- Integral variables can be attributed for randomization using `rand` or `randc`

```
typedef struct {  
    data_type variable0;  
    data_type variable1;  
} struct_type;
```

**Example:**

```
typedef struct {  
    rand int my_int;  
    real my_real;  
} my_struct;  
my_struct var0, var1;  
var0 = { 32, 100.2 };  
var1 = { default:0 };  
var1.my_int = var0.my_int;
```

Can not randomize real variables

Both fields set to 0

3-37

`rand` and `randc` are discussed in later units.

**union**

**3-38**

## union - Data Union

### ■ Overloading variable definition similar to C union

- Only **packed** unions supported in VCS (as of 2014.12)
  - ◆ All members must be of same size unless **tagged\*** (See Note)

```
typedef union packed {  
    data_type variable0;  
    data_type variable1;  
} union_type;
```

Example:

```
typedef union packed {  
    int my_int;  
    int my_val;  
} my_union;  
my_union var0, var1;  
var0.my_int = 32;  
var1.my_val = 100;  
var1.my_int = var0.my_int;
```

VCS can not  
randomize  
unions

All  
members  
must  
have  
same size

Can only  
access  
one field  
in unions

```
union packed tagged {  
    data_type0 variable0;  
    data_type1 variable1;  
} union_variable;
```

Example:

```
union packed tagged {  
    int my_i;  
    bit my_r;  
} my_var0, my_var1;  
my_var0.my_i = 32;  
my_var1.my_r = '1;  
my_var1.my_i = my_var0.my_i;
```

tagged  
union  
members  
may have  
different  
size

3-39

In **union**, you can only access one field. As soon as you populate one field, you cannot access the other fields.

\*The qualifier **tagged** in a union declares it as a *tagged union*, which is a type-checked **union**. An ordinary (untagged) **union** can be updated using a value of one member type and read as a value of another member type, which is a potential type loophole. See LRM for more details.

## **Streaming Operators (Pack/Unpack)**

**3-40**

## Streaming Operators: Pack/Unpack (1/3)

- The streaming operators perform packing and unpacking of data into a sequence of bits in a user-specified order.
  - ">>" operator streams data from left to right
  - "<<" operator streams data from right to left
- Data is packed when operators are used on the RHS of an assignment

```
bit[31:0] s; bit[7:0] a,b,c;  
s = {<< {a,b,c}};
```

- Data is unpacked when operators are used on the LHS of an assignment

```
{<< {a,b,c}} = s; // Unpack s into a,b,c
```

3-41

```
bit[3:0] sa = 4'b0111, da = 4'b0011;  
bit[7:0] addr_pair;  
  
// stream concatenation{sa,da}left to right  
addr_pair = {>> {sa,da}};  
$display("%b", addr_pair); //displays 01110011  
  
//stream addr_pair right to left  
addr_pair = {<< {addr_pair}};  
$display("%b", addr_pair); //displays 11001110
```

## Streaming Operators: Pack/Unpack (2/3)

- Stream can be sized

- e.g. `dbl_wrd = { << byte {w1, w2} };`
  - ◆ Stream in byte-sized slices
  - ◆ All braces are required
- Allows any arbitrary order / organization
  - ◆ bit-reversed, little-endian, byte-swapped, nibbles, etc...

```
bit[15:0] l = 16'h_abcd, m = 16'h_cafe;
bit[31:0] lm_pack;
lm_pack = {<< byte{l,m}}; // pack byte-size chunks
$display("%h", lm_pack);    // displays "fecacdab"
{>>{l,m}} = lm_pack;       // unpack
$display("l=%h m=%h", l,m); // displays "l=feca m=cdab"
```

## Streaming Operators: Pack/Unpack (3/3)

- Streaming operators can be used to pack complex data structures into arrays
  - structs and unions
  - objects – discussed in later section

```
typedef struct {  
    int hdr, len;  
    byte data[];  
    byte crc;  
} Packet; // unpacked struct  
  
Packet p;  
byte q[$]; // queue of bytes  
  
q = {<< byte {p.hdr, p.len, p.data, p.crc}}; // pack  
  
// unpack  
{<< byte {p.hdr, p.len, p.data with [0+:p.len], p.crc}} = q;
```

3-43

structs and unions are supported by the SystemVerilog language. They are described elsewhere in the Appendix of this unit.

Object oriented programming is discussed in a later unit.

When unpacking, the with operator limits the data to be streamed into dynamically sized types. When packing, it works like an array slice and allows variable index or complex expressions in the slice. See SystemVerilog LRM for more details.

This page was intentionally left blank

# Agenda

DAY  
1

1 The Device Under Test (DUT)

2 SystemVerilog Verification Environment 

3 SystemVerilog Language Basics - 1

4 SystemVerilog Language Basics - 2 

# Unit Objectives



After completing this unit, you should be able to:

- Use system functions and controls for randomization of variables
- Define aliases using `typedef`
- Use different types of language operators
- Use flow control constructs to build a SV testbench
- Define and use subroutines in a SV program
- Understand lifetime of a code block

# System Functions: Randomization

- **\$urandom**: Return a 32-bit unsigned random number
  - Initial seed can be set with run-time switch: `+ntb_random_seed=seed_value`
  - Object/Thread/Scope level seed can be set with `srandom(seed)`
  - `$random` Verilog function gives poor distribution and repeatability
- **\$urandom\_range(max, [min])**: Return a 32-bit unsigned random number in specified range
  - `min` is 0 if not specified
- **randcase**: Select a weighted executable statement

```
randcase
  10 : f1();
  20 : f2();           // f2() is twice as likely to be executed as f1()
  50 : x = 100;
  30 : randcase ... endcase;    // randcase can be nested
endcase
```

4-3

**\$urandom\_range(max, [min])**

max and min are unsigned integers. min defaults to 0 if not specified. If max< min then the two numbers are reversed.

The default seed in vcs is 1. A seed of 0 is replaced with a seed of 1.

The `srandom` function is also used to set unique seeds (if needed) for threads and objects (both discussed in later units)

# User Defined Types and Type Cast

- Use `typedef` to create an alias for another type

```
typedef bit [31:0] uint;
typedef bit [5:0] bsix_t; // Define new type
bsix_t my_var;           // Create 6-bit variable
```

- Use `<type>'(<value>|<variable>)` to convert data types  
(static cast – checks done at compile-time)

```
bit[7:0] payload[];
//int is a signed type. Use care when randomizing
int temp = $urandom; //temp can be negative
payload = new[(temp % 3) + 2]; //temp % 3 can be -2!!!
payload = new[(uint'(temp) % 3) + 2]; //temp cast to uint
payload = new[(unsigned'(temp)%3)+2]; //can also cast to unsigned
```

**What are the possible sizes of array payload?**

4-4

`payload = new[(tmp % 3) + 2];` will result in possible size of 0, 1, 2, 3 or 4 because `(tmp % 3)` has these possible values: -2, -1, 0, 1, 2.

`payload = new[(uint'(tmp) % 3) + 2];` will result in possible size of 2, 3 or 4 because `uint'(tmp) % 3` has these possible values: 0, 1 and 2.

`payload = new[(unsigned'(tmp) % 3) + 2];` will result in possible size of 2, 3 or 4 because `unsigned'(tmp) % 3` has the following possible values: 0, 1 and 2.

`$cast()` can also be used to convert data types. This is dynamic casting. No compile time checks are performed, but you can use `$cast` as a function that returns a status of success or failure.

A better way to randomize `tmp` in the example above is to use `$urandom_range(2)`. This ensures positive values from 0 to 2.

A better methodology is always to use bit types for testbench variables so that when randomized they will always have positive values, since bit is unsigned.

# Operators

+ - * /	arithmetic	~	bitwise negation
%	modulus division	&	bitwise and
++ --	increment, decrement	&~	bitwise nand
> >= < <=	relational	~	bitwise nor
!	logical negation	^	bitwise inclusive or
&&	logical and	^~	bitwise exclusive or
	logical or	{ }	bitwise exclusive nor
==	logical equality	&	concatenation
!=	logical inequality	~&	unary and
====	case equality		unary nand
!==	case inequality	~	unary or
==?	wildcard case equality	^	unary nor
!=?	wildcard case inequality	~^	unary exclusive
<<	logical shift left	? :	unary exclusive nor
>>	logical shift right	inside	conditional (ternary)
<<<	arithmetic shift left	iff	set membership
>>>	arithmetic shift right		qualifier

## Assignment:

= += -= \*= /= %= <<= >>= <<<= >>>= &= |= = ^= ~&= ~|= = ~^=

4-5

# inside Operator

- Use **inside** operator to find an expression within a set of values

```
bit[31:0] smpl, r1, r2; int golden[$] = {3, 4, 5};  
if (smpl inside {r1, r2})... // (smpl == r1 || smpl == r2)  
if (smpl inside {[r1:r2]})... // (smpl inside range r1 to r2)  
if (result inside {1, 2, golden})... // same as {1, 2, 3, 4, 5}
```

- **inside operator uses**

- == operator on non-integral expressions
- ==? on integral expressions
  - ◆ x and z are ignored in set of values
  - ◆ wildcards (?) preferred instead of x and z

e.g. if(result inside {3'b1?1, 3'b00?})//{3'b101,3'b111,3'b000,3'b001}

4-6

Integral expressions are of type reg, logic, int, integer, bit, time, byte, shortint and longint.

Non-integral expressions are real, shortreal, realtime and string.

# iff Operator

## ■ Use **iff** operator to qualify

- event controls
  - ◆ `@(rtr_io.cb iff(rtr_io.cb.frame_n[prt_id] != 0));`
- property execution – not covered in this workshop
- coverage elements – covered later
  - ◆ cover points
  - ◆ bins of cover points
  - ◆ cross coverage
  - ◆ cross coverage bins

4-7

# Know Your Operators!

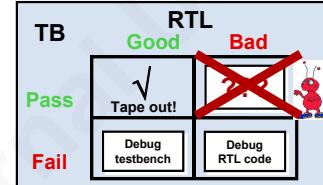
- What is printed to console with following code?

```
logic[3:0] sample, ref_data;  
sample = dut.cb.dout[3:0];  
if (sample != ref_data) $display(" Error! ");  
else $display(" Pass! ");
```

- When sample = 4'b1011 & ref\_data = 4'b1010
- When sample = 4'b101x & ref\_data = 4'b1010
- When sample = 4'b101x & ref\_data = 4'b101x

- Avoid false positives by checking for pass condition!

```
sample = dut.cb.dout[3:0];  
if (sample == ref_data) ;  
else $display(" Error! ");  
  
assert(!$isunknown(sample));  
assert($onehot(sample));  
if(!$onehot0(sample)) ...;
```



4-8

In Verilog, the `==` and `!=` operator will return `x` as the result if any operand contains `x` or `z`. The `if` statement then treats `x` as a failure. Therefore, in the example above, if either `sample` or `ref` contains `x` or `z`, the `else` condition will always execute.

Common SystemVerilog system functions used to query vector state:

`$countones` - Returns the number of 1's in a bit vector - unknown bits ignored

`$onehot0` - Returns true if at most one bit of expression is high – unknown bits ignored

`$onehot` - Returns true if only one bit of an expression is high – unknown bits ignored

`$isunknown` - Returns true if any bit of the expression is ‘x’

# Sequential Flow Control

## ■ Conditionals:

- `if (x==7) a=7; else a=8;`
- `a = (x == y) ? 7 : 8;`
- `assert (true condition);`
- `case(expr) 0: ...; 1: ...; default: ...; endcase`

## ■ Loops:

- `repeat(expr) begin ... end`
- `for(expr; expr; expr) begin ... end`
- `foreach(array[index]) begin ... end`
- `forever begin ... end`
- `while(expr) begin ... end`
- `do begin ... end while (expr);`
- `break` to terminate loop
- `continue` to terminate current loop iteration

4-9

`assert (true condition)` is like an `if (true condition) ; else $error`

`$error([msg])` is a SystemVerilog system function that returns an error message. `msg` is a formatted string with arguments exactly like a `$display()` system function call..

# Subroutines (task and function)

- Tasks can block
- Functions can not block

```
Pass by reference  
task print_sum(ref int a[], input int start=0);  
    automatic int sum = 0;  
    for (int j=start; j<a.size(); j++)  
        sum += a[j];  
    $display("Sum of array is %0d", sum);  
endtask  
...  
print_sum(my_array);
```

Default value  
task does not return value

## Subroutine lifetime

- Defaults to **static** in program, module, package, interface
- Defaults to **automatic** in class
- Can be made **automatic**

## Subroutine variables

- Default to subroutine scope and lifetime
- Can be made **automatic or static**

```
function automatic int factorial(int n);  
    static int shared_value = 0:  
    if (n < 2) return(1);  
    else return(n * factorial(n-1));  
endfunction  
...  
result = factorial(my_val);
```

Pass by value  
function returns value

4-10

A **static** variable is allocated once and exists for the lifetime of the simulation.

An **automatic** variable is automatically allocated when program flow enters, and automatically deallocated when program flow leaves, the context of the variable e.g a subroutine or **begin...end** block.

A **function** can return in two ways – either a **return** is explicitly called or when **endfunction** is reached. Every **function** has a variable matching the function name and return type. A function will return the value of this variable if it ends at **endfunction**, or a **return** is explicitly executed without any argument

Functions can call a **task** as long as the called **task** does not block. This may not work in older releases of simulators.

Classes are discussed in a later unit.

# Subroutine Argument Binding and Skipping

- Arguments can be bound (passed) to the subroutine by

- position
- name

- Arguments can be skipped if they have default values

```
program automatic test;
    task tally(ref byte a[], input logic[15:0] b, c = 0, u, v);
        ...
    endtask
    initial begin
        logic[15:0] B = 100, C = 0, D = 0, E = 0;
        byte A[] = {1,3,5,8,13};
        tally(A, B, , D, E);           arguments passed by position
        tally(.c(C), .b(B), .a(A), .u(D), .v(E) );
    end
endprogram
```

skipped arguments use default value

arguments passed by name

4-11

Arguments passed by name do not have to be in the same position as the formal arguments.

If the arguments have default values, they are treated like parameters to module instances. If the arguments do not have a default, then they must be passed to the subroutine call, or the compiler issues an error.

If both positional and named arguments are specified in a single subroutine call, then all the positional arguments must come before the named arguments.

# Subroutine Arguments

## ■ Type and direction are both sticky

- Any following arguments default to that type and direction



See note

direction	effect
<b>input</b>	copy value in at beginning - default
<b>output</b>	copy value out at end
<b>inout</b>	copy in at beginning and copy out at return
<b>ref</b>	pass by reference, makes argument variable the same as the calling variable. Changes to argument variable will change calling variable immediately
<b>const ref</b>	pass by reference but read only. Saves time and memory for passing arrays to tasks & functions

Default direction is **input**,  
default type is **logic**

**a, b: input logic**  
**u, v: output bit [15:0]**

Read-only pass  
via reference

**task T3(a, b, output bit [15:0] u, v, const ref byte c[]);**

4-12

The SystemVerilog 1800-2012 LRM states:

There is a default direction of **input** if no direction has been specified. Once a direction is given, subsequent formals default to the same direction.

Each formal argument has a data type that can be explicitly declared or inherited from the previous argument. If the data type is not explicitly declared, then the default data type is **logic** if it is the first argument or if the argument direction is explicitly specified. Otherwise, the data type is inherited from the previous argument.

# Test for Understanding

- What's the direction and data type of each argument?

```
task T3(ref byte a[], logic[15:0] b, c, output u, v);
    b = c;
    foreach(a[i])
        a[i] = i;
endtask
```

direction is ?,  
type is ?

direction is ?,  
type is ?

direction is ?,  
type is ? !

```
initial begin
    logic[15:0] B = 100, C = 0, D = 0, E = 0;
    byte A[] = {1,3,5,8,13};
    T3(A, B, C, D, E);
    foreach(A[i])
        $display(A[i]);
    $display(B, C, D, E);
```

What will be displayed?

```
end
```

Recommendation: declare all directions

4-13

Argument	direction	type
a []	ref	byte
b	ref	logic[15:0]
c	ref	logic[15:0]
u	output	logic //Because the direction changed
v	output	logic //sticky from u

The program test results is as follows:

```
0
1
2
3
4
0 0 X X
```

To make b and c pass via value, change the declaration to:

```
task T3(ref byte a[], input logic[15:0] b, c, output u, v);
```

## Code Block Lifetime Controls

- **Simulation ends when all programs end**
  - Execution of a `program` ends when
    - ◆ All `initial` blocks in `program` reach end of code block, or `$finish` is executed
  - Recommend you use only one `program` block to define your testbench
- **Execution of a subroutine ends when one of**
  - `endtask`, `endfunction` is encountered
  - `return` is executed
- **Execution of a loop ends when one of**
  - `end` (of loop `begin`) is encountered
  - `break` is executed
- **Execution of loop immediately advances to next iteration when**
  - `continue` is executed

4-14

Loops are code blocks enclosed in `for`, `foreach`, `forever`, `while`, `do...while` and `repeat`

# Helpful Debugging Features

## ■ What to print for debugging?

- Use %t and %m to print the simulation time and location of call
- Indicate severity of message

```
function void check();  
    static int cnt = 0;  
    string message;  
    if (!compare(message)) begin  
        $display("%m\n[ERROR] %t: %s", $realtime, message);  
        $finish;  
    end  
    $display(" [NOTE] %t: %0d Packets passed\n", $realtime, ++cnt);  
endfunction: check
```

Indicate message severity (ERROR, DEBUG, etc.)

Simulation time

hierarchical path to check()

- Use \$timeformat to set the format for %t

```
// $timeformat [(units, precision, suffix_string , minimum_field_width)];  
// We are using $timeformat(-9, 1, "ns", 10) in the labs  
// $time returns time as a 64-bit integer // $realtime returns time as a real value
```

4-15

\$timeformat default value for arguments

Argument	Default
units	The smallest time precision argument of all the `timescale compiler directives in the source description
precision	0
suffix_string	A null character string
minimum_field_width	20

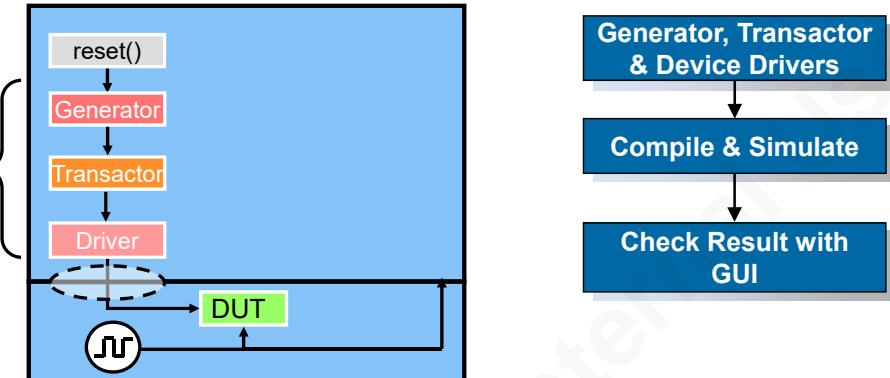
## Lab 2 Introduction



60 minutes

Generator,  
Transactor &  
Device Drivers

Develop Generator, Transactor  
& Device Drivers to Drive One  
Packet Through the Router



4-16

## Unit Objectives Review

**Having completed this unit, you should be able to:**

- **Use system functions and controls for randomization of variables**
- **Define aliases using `typedef`**
- **Use different types of language operators**
- **Use flow control constructs to build a SV testbench**
- **Define and use subroutines in a SV program**
- **Understand lifetime of a code block**



4-17

## **Appendix**

**Import and Export Verilog subroutines**

**Import and Export C/C++ subroutines (DPI)**

**4-18**

## **Import and Export Verilog subroutines**

**4-19**

# Import and Export Verilog Subroutines (1/2)

```
task root_task();                                // Root level subroutine
  $display("I'm Root Task"); endtask
module bfm(top_io.BFM bfm_io);                  // BFM's that implement I/O via SystemVerilog interface
  task bfm_io.bfm_task();                         // Subroutine to be accessed by test program
    $display("I'm BFM task"); endtask
endmodule: bfm
module vip();                                    // BFM's that do not implement I/O via SystemVerilog interface
  task vip_task();                             // Subroutine to be accessed by test program
    $display("I'm VIP task"); endtask
endmodule: vip
interface top_io();                            // SystemVerilog Interface to be used by test program
  task interface_task();                      // Subroutine to be accessed by test program
    $display("I'm Interface task"); endtask
  task vip_task();                           // Wrapper for non-SystemVerilog interface BFM's
    test_top.VIP.vip_task();                  // XMR reference via top-level instance
  endtask
modport TB(import task interface_task(), import task bfm_task(), import task vip_task());
modport BFM(export task bfm_task());
endinterface: top_io
```

...Continued on next slide

4-20

## Import and Export Verilog Subroutines (2/2)

...Continued from previous slide

```
program automatic test(top_io.TB test);
initial begin
    root_task();                                // direct $root access
    test_top.VIP.vip_task();                     // VIP XMR access via top module
    test_top.BFM.bfm_io.bfm_task();              // BFM XMR access via top module
    test.vip_task();                            // interface VIP access
    test.bfm_task();                           // interface BFM access
    test.interface_task();                     // interface access
end
endprogram: test
```

```
module test_top;
    top_io IO();
    test TEST(IO);
    bfm BFM(IO);
    vip VIP();
endmodule
```

4-21

## **Import and Export C/C++ subroutines (DPI)**

**4-22**

# SV Direct Programming Interface

- **Direct Programming Interface (DPI-C)**

- SystemVerilog calls C/C++ functions
- C/C++ calls SystemVerilog functions & blocking tasks

- **Simple interface to C models**

- Allows SystemVerilog to call a C function just like any other native SystemVerilog **function/task**
- Testbench variables passed directly to/from C/C++
- No need to write PLI-like applications/wrappers

- **DPI-C cannot be used to**

- Attach callbacks to a signal
- Traverse hierarchy, get handles to instances or objects
  - ◆ Instead use PLI/VPI for these tasks

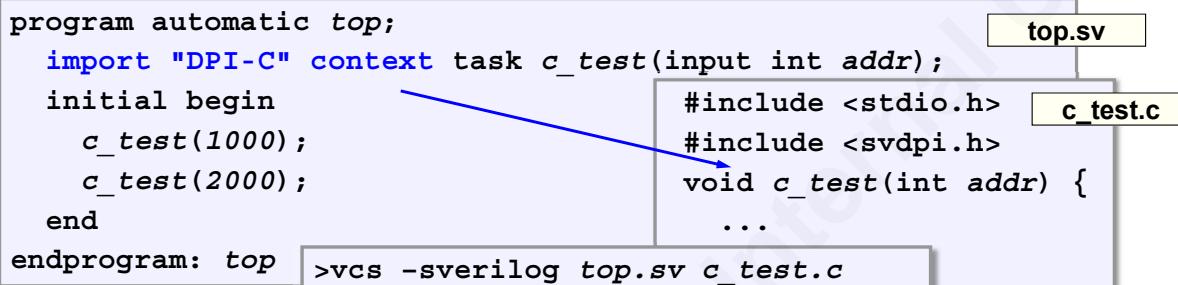
4-23

## DPI-C: import

### ■ SystemVerilog calling C/C++ functions

```
import "DPI-C" [cname =] [pure] function type name (args);  
import "DPI-C" [cname =] [pure | context] task name (args);
```

- **cname**: C function name to map to SystemVerilog prototype name, if different
- **pure**: value returned only via call (no **output/inout** argument)
- **context**: Allows access to SystemVerilog exported routines or data in C code



4-24

Extracted from LRM (see LRM for more information):

Some DPI imported tasks or functions require that the context of their call be known.

An imported task or function not specified as context shall not access any data objects from SystemVerilog other than its actual arguments. Only the actual arguments can be affected (read or written) by its call.

A context imported task or function, however, can access (read or write) any SystemVerilog data objects by calling PLI/VPI or by calling an embedded export task or function. Only the calls of context imported tasks and functions are properly instrumented. Therefore, only those tasks and functions can safely call all functions from other APIs, including PLI and VPI functions or exported SystemVerilog functions.

For imported task or functions not specified as context, the effects of calling PLI, VPI, or SystemVerilog functions can be unpredictable; and such calls can crash if the callee requires a context that has not been properly set.

Special DPI utility functions exist that allow imported task and functions to retrieve and operate on their context. For example, the C implementation of an imported task or function can use **svGetScope()** to retrieve an **svScope** corresponding to the instance scope of its corresponding SystemVerilog import declaration.

A context imported task or function will limit RTL optimization (due to instrumentation). Create context imported task or function only if necessary.

# DPI-C: export

## ■ C calling SystemVerilog functions

- `export "DPI-C" [cname =] function name;`

## ■ C calling SystemVerilog tasks

- task can block
- `export "DPI-C" [cname =] task name;`

```
import "DPI-C" context task c_test(int addr);  
export "DPI-C" task apb_write;  
  
task apb_write(input int addr, data);  
    ... @(posedge ready); ...  
endtask  
  
initial c_test(1000);  
  
>vcs -sverilog top.sv c_test.c
```

```
#include <stdio.h>  
#include <svdpi.h>  
extern void apb_write(int, int);  
  
void c_test(int base_addr) {  
    ...  
    apb_write(addr, data); ...  
}
```

4-25

Export declarations are allowed to occur only in the scope in which the function/task being exported is defined.

1. Only one export per function/task is allowed in a given scope.
2. Class member functions/tasks cannot be exported.
  - All other SystemVerilog functions/tasks can be exported.
3. Export functions/tasks are always context.
4. Multiple export declarations are allowed with the same *c\_identifier*, explicit or implicit, as long as they are in different scopes and have the equivalent type signature .
  - Multiple export declarations with the same *c\_identifier* in the same scope are forbidden.
5. SystemVerilog tasks do not have return value types. The return value of an exported task is an *int* value that indicates if a disable is active or not on the current execution thread.

# Declaration of Imported Functions and Tasks

## ■ Imported task or function

- Can be declared anywhere a native SV **function** or **task** can be declared

```
import "DPI-C" pure function real sin(input real r); // math.h
```

- Can have zero or more **input**, **output** and **inout** arguments. **ref** is not allowed.

```
import "DPI-C" function void myInit();
```

- Can map method name if it conflicts with existing name

```
import "DPI-C" test = function int my_test();
```

## ■ Imported function return restricted to **void**, **byte**, **shortint**, **int**, **longint**, **real**, **shortreal**, **chandle** and **string**

- ◆ Scalar values of type **bit** and **logic**

```
import "DPI-C" function int getStim(input string fname);
```

4-26

## DPI-C: Supported Data Types

SystemVerilog	C (input)	C (out/inout)	SystemVerilog	C (input)	C (out/inout)
<code>byte (1)</code>	<code>char</code>	<code>char*</code>	<code>bit</code>	<code>svBit</code>	<code>svBit*</code>
<code>shortint</code>	<code>short int</code>	<code>short int*</code>	<code>logic, reg</code>	<code>svLogic</code>	<code>svLogic*</code>
<code>int</code>	<code>int</code>	<code>int*</code>	<code>bit[N:0]</code>	<code>const svBitVecVal*</code>	<code>svBitVecVal*</code>
<code>longint (1)</code>	<code>long int</code>	<code>long int*</code>	<code>reg[N:0]</code>	<code>const svLogicVecVal*</code>	<code>svLogicVecVal*</code>
<code>shortreal</code>	<code>float</code>	<code>float*</code>	<code>logic[N:0]</code>		
<code>real</code>	<code>double</code>	<code>double*</code>	<code>array[size]</code>	<code>type[ ]</code>	<code>type[ ]</code>
<code>string</code>	<code>const char*</code>	<code>char**</code>	<code>array[M][N]</code>	<code>type[ ][ ]</code>	<code>type[ ][ ]</code>
<code>string[n]</code>	<code>const char**</code>		<code>array[ ] (import only)</code>	<code>const svOpenArrayHandle</code>	<code>svOpenArrayHandle</code>
			<code>chandle</code>	<code>const void*</code>	<code>void*</code>

(1) `input/output/inout` only, not for function return value

4-27

# DPI-C: Supported Data Types

- Arguments must match types between SystemVerilog and C
  - DPI does not check for type compatibility - user responsibility
- VCS produces vc\_hdrs.h
  - Use it as a guide to see how types are mapped
- Return types (32 bits max)
  - (unsigned) int, char\*
- Protection
  - It's up to the C code to not modify input parameters
  - Use const to double check your C code

Argument directions	
input	Input to C code
output	Output from C code (initial value undefined)
inout	Input and Output from C code
ref	Not supported by LRM. Use inout instead

4-28

## DPI-C Example: Integer and Strings

SystemVerilog	C Data Type	Description
int	int	Integer passed by value
string	char*	String passed by value

```
program automatic top;
    import "DPI-C" function void display_int(int i);
    import "DPI-C" function void display_str(string s);
    initial display_int(10);
    initial display_str("hello");
endprogram: top
```

```
#include <stdio.h>
#include <svdpi.h>
void    display_int(int   i) { io_printf ("C : int = %d\n" , i); }
void    display_str(char* s) { io_printf ("C : str = %s\n" , s); }
```

4-29

Use `io_printf` so output goes to stdout and simulation log file.

## DPI-C: 4-State Data Types

	SystemVerilog	C Data Type	Description
	<code>reg/logic</code>	<code>svLogic</code>	Reg/logic passed by value
■ <code>reg/logic</code>	<code>reg/logic [n:0]</code>	<code>svLogicVecVal*</code>	Reg/logic vector passed by value

- `reg/logic`
  - 4 State values in SystemVerilog represented using `svLogic` in C
- `reg/logic arrays`
  - Represented using array of `svLogicVecVal` struct in C

System Verilog	C: <code>svLogic</code> Data	C: <code>svLogicVecVal</code> Data	Control
0	0	0	0
1	1	1	0
Z	2	0	1
X	3	1	1

```
/* (a chunk of) packed bit array */
typedef uint32_t svBitVecVal;
/* (a chunk of) packed logic array */
typedef struct vpi_vecval {
    uint32_t aval; //Data (Value)
    uint32_t bval; //Control
} s_vpi_vecval, *p_vpi_vecval;
typedef s_vpi_vecval svLogicVecVal;
```

4-30

Use int if you do not need 4-state logic. Integers are easier to handle in C.

## DPI-C Example: reg/logic

```
program automatic top;
    import "DPI-C" function void display_reg(logic r);
    import "DPI-C" function void display_vec(logic [31:0] v);
    initial display_reg(1'bx);
    initial display_vec(32'h12xz);
endprogram: top
#include <stdio.h>
#include <svdpi.h>
void display_reg(svLogic r)
{
    { io_printf ("c=%d, d=%d\n", (r>>1)&1, r&1 );
}
void display_vec(svLogicVecVal* v)
{
    { io_printf ("c=%x, d=%x\n", v->aval, v->bval);
}
```

4-31

The simulation output will be:

c=1, d=1  
c=ff, d=12f0

## DPI-C: chandle

### ■ chandle

- Allows C to allocate memory, pass to SystemVerilog
  - ◆ SystemVerilog can only access memory address
- SystemVerilog can then pass the handle back to C
- Allocate and deallocate in the same language

```
//standard C functions
import "DPI-C" function chandle malloc(int size);
import "DPI-C" function void free(chandle ptr);
// abstract data structure: queue
import "DPI-C" function chandle newQ (input string name_of_queue);
// Note the following import uses the same foreign function for implementation as the
// import, but has different SystemVerilog name and provides a default value for the argument.
import "DPI-C" newQ=function chandle AnonQ(input string s=null);
import "DPI-C" function chandle newElem(bit [15:0]);
```

4- 32

# DPI-C: Array Access

## ■ C Structure contains array details

- struct svOpenArrayHandle

```
logic [31:0] array8[8];
```

```
import "DPI-C" function int my_func(input int data[ ]);  
  
int my_func(const svOpenArrayHandle handle) {  
    int* data;  
    data = (int*) svGetArrayPtr(handle);  
    ...  
}
```

## ■ Data / Array

- Get a handle to the data type using `svGetArrayPtr()`
- Type cast to the correct type to match SystemVerilog
- `char*`, `int*`, `svLogicVecVal*`, etc.

4-33

# DPI-C: Array Access

## ■ Access Functions

using logic [31:0] array8[8]; for the functions below

- **void \*svGetArrayPtr(arg)**
  - ◆ Returns pointer to representation of the whole data array
  - ◆ Type cast this to the correct type, based on SV array type
- **int svDimensions(arb)**
  - ◆ # of dimensions. array8: 1 dimension
- **int svSizeOfArray(ary)**
  - ◆ # of bytes to store array, including 4-state. array8: 64
- **int svLow(ary, dim)**
  - ◆ Low index of array. array8: 0 for dimension 1
- **int svHigh(ary, dim)**
  - ◆ High index of array. array8: 7 for dimension 1

4-34

## DPI-C: Import Examples

SystemVerilog	C Data Type	Description
int array[]	svOpenArrayHandle	Dynamic array

```
program automatic top;
    import "DPI-C" function void display_array_val(int arr[]);
    int arr[] = new[4];
    initial display_array_val(arr);
endprogram: top
```

```
#include <stdio.h>
#include <svdpi.h>
void display_array_val(svOpenArrayHandle handle) {
    int* data = (int*) svGetArrayPtr(handle); int i;
    io_printf("Dim=%d, Size=%d\n", svDimensions(handle),
              svSizeOfArray(handle));
    io_printf("Low=%d, High=%d\n", svLow(handle,1), svHigh(handle,1));
    for (i=svLow(handle,1); i<=svHigh(handle,1); i++)
        io_printf("data[%0d] = %d\n", i, data[i]);
}
```

4-35

logic data type example:

SV:

```
program automatic test(router_io.TB rtr_io);
    import "DPI" function void display_array_val(logic[31:0] arr[]);
    initial begin
        logic[31:0] arr[8];
        foreach(arr[i]) arr[i] = {24'b0, byte'(i)};
        display_array_val(arr);
    end
endprogram
```

C:

```
#include <stdio.h>
#include <svdpi.h>
void display_array_val(svOpenArrayHandle handle) {
    svLogicVecVal* data = (svLogicVecVal*) svGetArrayPtr(handle);
    int i;
    io_printf("Dim=%d, Size=%d\n", svDimensions(handle),
              svSizeOfArray(handle));
    io_printf("Low=%d, High=%d\n", svLow(handle,1), svHigh(handle,1));
    for (i=svLow(handle,1); i<=svHigh(handle,1); i++) {
        io_printf("d[%0d] = %x, c[%0d] = %x\n", i, data->d, i, data->c);
        data++;
    }
}
```

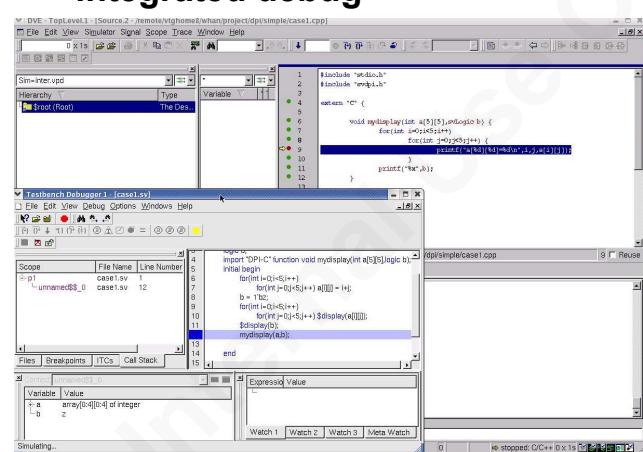
# DPI-C: Compile and Debug

- Enable the C/C++ source code debug with -g options.

```
% vcs -sverilog  
  casel.sv casel.cpp  
  -debug_all -CFLAGS -g
```

- DVE and Verdi support SV design/testbench and C/C++ integrated debug

```
*Src1:/remote/us01/home41/nalina/Verdi_SP1/open_c/test.c  
1 #include <stdio.h>  
2 extern int foo(int i);  
3 int cifunc(int i)  
4 {  
5     int j;  
6     #ifdef mydef  
7         printf("mydef called\n");  
8     #endif  
9     printf("in C: before export, i = %d\n", i);  
10    j = foo(i);  
11    printf("in C: after export, i = %d, j = %d\n", i, j);  
12 }  
13
```



4-36

## DPI-C: Header Files & Examples

- **DPI header files**
  - \$VCS\_HOME/include
- **SystemVerilog examples:**
  - \$VCS\_HOME/doc/examples/sv/dpi
  - Directory contains following two subdirectories:
    - ◆ export\_fun – DPI export function for SV
    - ◆ import\_fun – DPI import function for SV

4-37

This page was intentionally left blank

# Agenda

DAY  
2

5 Concurrency



6 Object Oriented Programming (OOP)  
– Encapsulation



7 Object Oriented Programming (OOP)  
– Randomization



## Unit Objectives

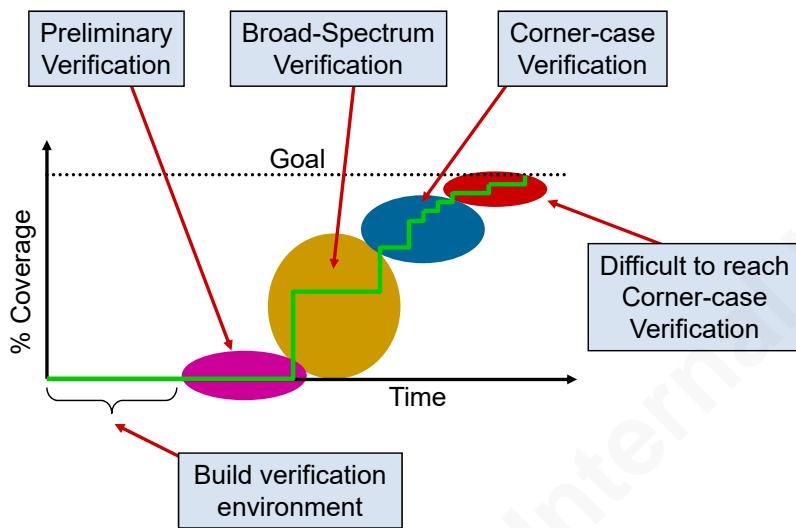


**After completing this unit, you should be able to:**

- **Divide a testbench into multiple current threads to execute parallel tasks**

# Day 1 Review

## Phases of verification



5-3

The process of reaching the verification goal starts with the definition of the verification goal. What does it mean to be done with testing? Typically, the answer lies in the functional coverage spec within a verification plan. The goal is then to reach 100% coverage of the defined functional coverage spec in the verification plan.

Once the goal has been defined, the first step in constructing the testbench is to build the verification environment.

To verify the environment is set up correctly, preliminary verification tests are usually executed to wring out the rudimentary RTL and testbench errors.

When the testbench environment is deemed to be stable, broad-spectrum verification based on random stimulus generation is utilized to quickly detect and correct the majority of the bugs in both RTL code and testbench code.

Based on functional coverage analysis, the random-based tests are then constrained to focus on corner-cases not yet reached via broad-spectrum testing.

Finally, for the very difficult to reach corner cases, customized directed tests are used to bring the coverage up to 100%.

Verification is complete when you reach 100% coverage as defined in the verification plan.

# Day 1 Review (Building Testbench)

```
interface simple_bus(input bit clk);
    logic req, gnt; Define interface
    logic [7:0] addr;
    wire [7:0] data;
    clocking cb @(posedge clk)
        output req;
        input gnt;
        ...
    endclocking: cb
    modport tb(clocking cb);
endinterface: simple bus Develop test program
program automatic test(simple_bus.tb sb);
    initial
        run_test();
    endprogram: test

```

...  
endmodule: cpu

module top;  
 logic clk = 0;  
 always #10ns clk = !clk;  
 simple\_bus sb(clk);  
 test t1(sb);  
 cpu c1(sb);  
endmodule: top

Encapsulate DUT, test and interface in harness module  
Connect DUT and program using interface

% vcs -sverilog cpu.v test.v interface.v top.v  
% simv

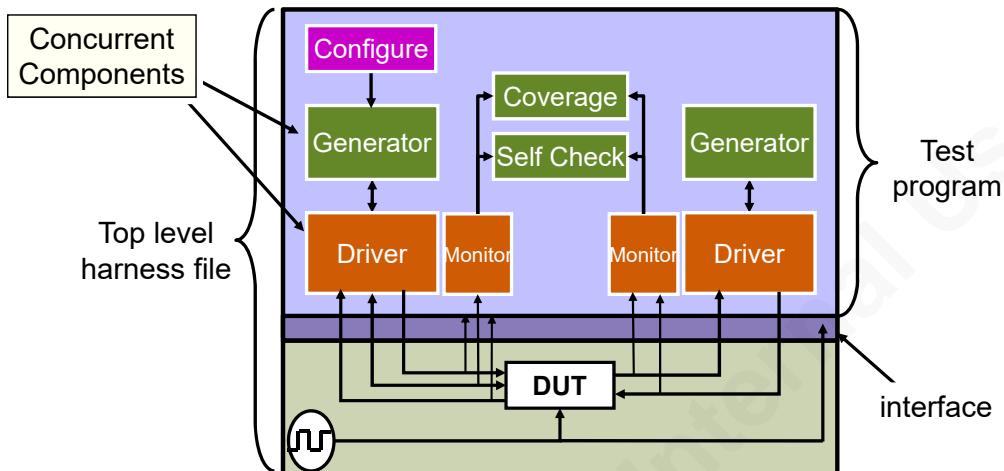
Compile and run with VCS

5-4

# Testbench Requires Concurrency

## ■ Components of the testbench run concurrently

- Concurrent components run as separate threads



5-5

What is a thread? A thread is similar to the sequential program (“Hello World”) i.e. a single point of execution.

A single thread also has a beginning, a sequence, and an end. A thread runs within a program.



The real excitement is not a single thread but rather multiple threads running at the same time performing different tasks in a single program. Multiple threads typically share the state information of a process, and share memory and other resources directly.

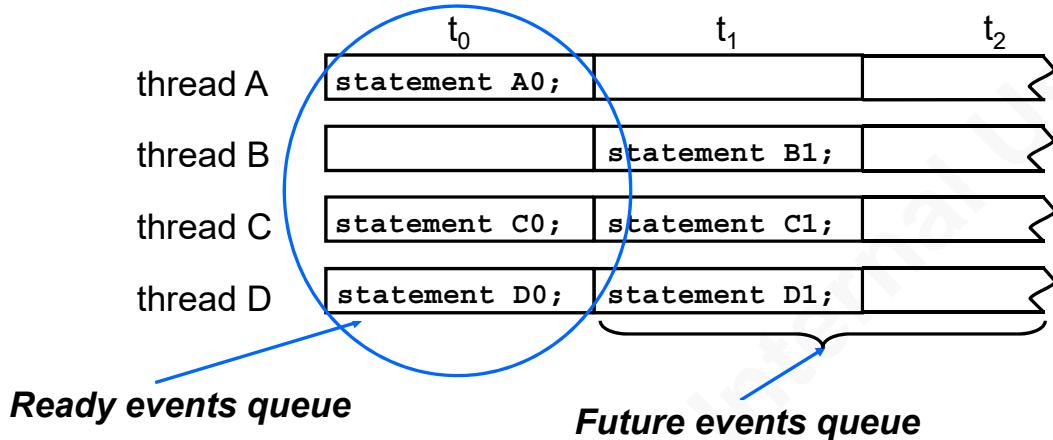
Each and every component may have many concurrent threads that need to be spawned, coordinated and synchronized.

In a module, threads are automatically created by the simulator for each of the `always` and `initial` blocks defined. This unit shows you how you can create threads in a program, where `always` blocks can not be used.

# Concurrency in Simulators

- A simulator can only execute one thread at a time in a single-core CPU

- Multiple threads waiting to execute at one simulation time point have to be scheduled in queues to run one-at-a-time



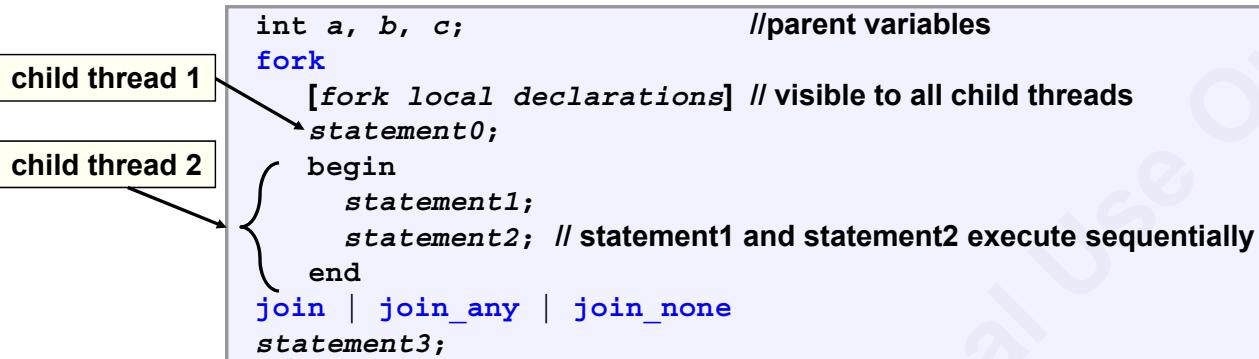
5-6

In a multi-core CPU the simulator may execute multiple threads at once. The number of threads that may execute in parallel depends on the simulator, the options used and other requirements.

# Creating Concurrent Threads

- Concurrent threads are created in a fork-join block

```
int a, b, c; //parent variables
fork
[fork local declarations] // visible to all child threads
statement0;
begin
    statement1;
    statement2; // statement1 and statement2 execute sequentially
    end
join | join_any | join_none
statement3;
```



- Statements enclosed in **begin-end** in a **fork-join** block are executed sequentially as a single concurrent child thread
- No predetermined execution order for concurrent threads
- Parent variables cannot be referred to in **join\_any** or **join\_none** except to initialize variables in fork local declarations

5-7

Make copies of parent variable that are needed by any threads, using local declarations in the **fork** declarative area (**fork local declarations**). These are scoped in the **fork**, only visible to all threads.

# How Many Child Threads?

A:

```
fork
begin
  recv();
end
begin
  send();
end
join
```

B:

```
fork
  recv();
  send();
join
```

C:

```
fork
begin
  recv();
  send();
end
join
```

D:

```
fork
begin
begin
  send();
  recv();
end
  check();
end
join
```

5-8

In the example above, A and B codes are identical because within fork-join blocks, single statements not explicitly inside begin-end blocks each have an implied begin-end. They fork two child threads each.

This is similar to the following:

```
always @(b)
  a = b;
```

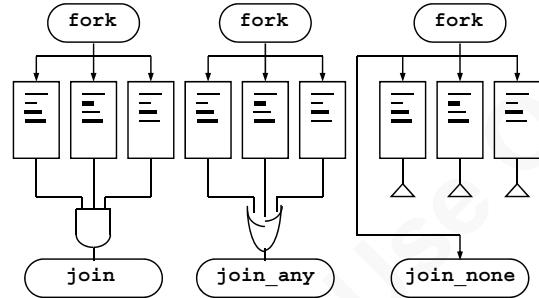
is exactly the same as:

```
always @(b) begin
  a = b;
end
```

Codes C and D fork only one child thread each.

## Join Options

```
fork  
    statement1;  
    statement2;  
    statement3;  
join | join_any | join_none  
statement4;
```



**join:** Child threads execute and all child threads must complete before statement4 is executed

**join\_any:** Child threads execute and one child thread must complete before statement4 is executed. Other child threads continue to run.

**join\_none:** Child threads are queued, statement4 executes. Child threads not executed until parent thread encounters a blocking statement or completes

5-9

Completion of a thread means that it is no longer in existence. If a thread is waiting for some condition to occur it is still in existence.

# Thread Execution

- **Once a thread executes**
  - It continues to execute until it finishes or a blocking statement is encountered
  - Child threads generated by it are queued
- **When executing thread encounters a blocking statement**
  - It is queued and a queued ready thread executes
- **Time advances when all threads are blocked**

**Examples of blocking statements:**

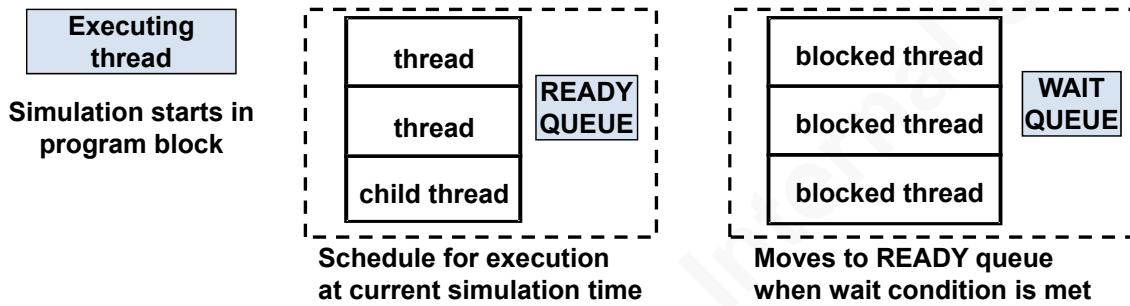
```
@(rtr_io.cb);           join_any *
wait (var_a == 1);      join *
#10;
```

5-10

\*Cannot be standalone. Used with encapsulating fork

# Thread Execution Model

- One executing thread, all other threads reside on queues
  - READY - to be executed at current simulation time
  - WAIT - blocked from execution until wait condition is met
- When the executing thread is blocked, it moves to the WAIT queue
  - The next READY thread then executes
- Simulation time advances when all threads are in WAIT



5-11

For Parallel VCS (using multiple cores) there may be many such executing threads, each with its READY and WAIT queues.

## Thread Design (1/2)

Will this work?

```
a = 0;
fork
begin: thread_1
    while ( a != 5 )
        if ( $time > MAX_TIME )
            $finish;
    end

begin: thread_2
    repeat(5) @rtr_io.cb;
    bus.cb.reg <= 1'b1;
    a = 5;
end
join
```

5-12

Solution on next slide.

## Thread Design (2/2)

- In multi-threaded programs, all threads must be finite or advance the clock!

```
a = 0;
fork
begin
    while ( a != 5 )
        if ( $time > MAX_TIME )
            $finish;
        else
            @(bus.cb);
    end
begin
    repeat(5) @rtr.io.cb;
    bus.cb.reg <= 1'b1;
    a = 5;
end
join
```

5-13

## Sharing Variables Among Threads Forked Using join

```
program automatic fork_join1;
initial begin
    int a = 0, b = 1;
    fork
        begin
            int d = 3;
            a = b + d;
        end
        begin
            int e = 4;
            b = a + e;
        end
    join
    $display("a = %0d", a);
    $display("b = %0d", b);
end
endprogram: fork_join1
```

- Child threads share the same parent variables

Can the child thread access a and b?

What are the final values of a and b?

5-14

The variable a and b are available to the child threads only for a join as shown. The access is illegal for join\_any and join\_none

## Thread v/s Program Completion

```
program automatic test();
    initial begin
        for (int i = 0; i < 16; i++)
            send(i);
    end
    task send(int j);
        fork
            begin
                $display("Driving port %0d", j);
                #1ns;
            end
        join_none
    endtask: send
endprogram: test
```

**Simulation ends at time 0. Why?**

5-15

The typical use of the `fork-join_none` structure is what's shown above. With this type of implementation, one can start multiple components of the testbench to execute concurrently.

With respect to the simulation time:

When an `initial` block inside `program` block reaches the `end` statement, simulation will terminate at that simulation time regardless of whether or not child threads still have future events to be executed.

In the code above, the `end` statement of the `initial` block is reached at time 0. Therefore, simulation will terminate at time 0.

From a practical point of view, objects modeling testbench components will define and use subroutines like `send()` shown on this slide. Objects are discussed in later units.

# Waiting for Child Threads to Finish

- To prevent improper early termination of simulation, use **wait fork**
  - suspends parent thread until all children threads have completed execution

```
program automatic test();
    initial begin
        for (int i = 0; i < 16; i++)
            send(i);
        wait fork;
    end
    task send(int j);
        fork
            begin
                $display("Driving port %0d", j); #1ns ;
            end
            join_none
        endtask: send
    endprogram: test
```

Blocking statement to control proper termination of simulation (more in later units)

5-16

Another mechanism you can use is **wait (...)** or **@(...)**. Example:

```
program automatic router_test(router_io.TB rtr_io);
    int done = 0;
    initial begin
        for (int i=0; i<16; i++) begin
            send(j);
        end
        wait(done >= 16);
    end
    task send(int i);
        $display("Driving port %0d", i);
        fork
            forever begin
                #1ns;
                done++;
            end
        end
    endtask
endprogram
```

# Thread Execution Issues

```
program automatic test;
    initial begin
        for (int i = 0; i < 16; i++)
            fork
                send(i); // illegal – (OK in VCS)
                join_none
            wait fork;
        end
        task send(int j);
            $display("Driving port %0d", j);
            #1ns;
        endtask: send
    endprogram: test
```

Produces output:

```
Driving port 16
Driving port 16
Driving port 16
Driving port 16
...
Driving port 16
Driving port 16
Driving port 16
Driving port 16
```

Why?

5-17

From IEEE SystemVerilog LRM 9.3.2 (Parallel Blocks):

Variables declared in the *block\_item\_declaration* of a fork-join block shall be initialized to their initialization value expression whenever execution enters their scope and before any processes are spawned. Within a **fork-join-any** or **fork-join-none** block, it shall be illegal to refer to formal arguments passed by reference other than in the initialization value expressions of variables declared in a *block\_item\_declaration* of the fork. These variables are useful in processes spawned by looping constructs to store unique, per-iteration data.

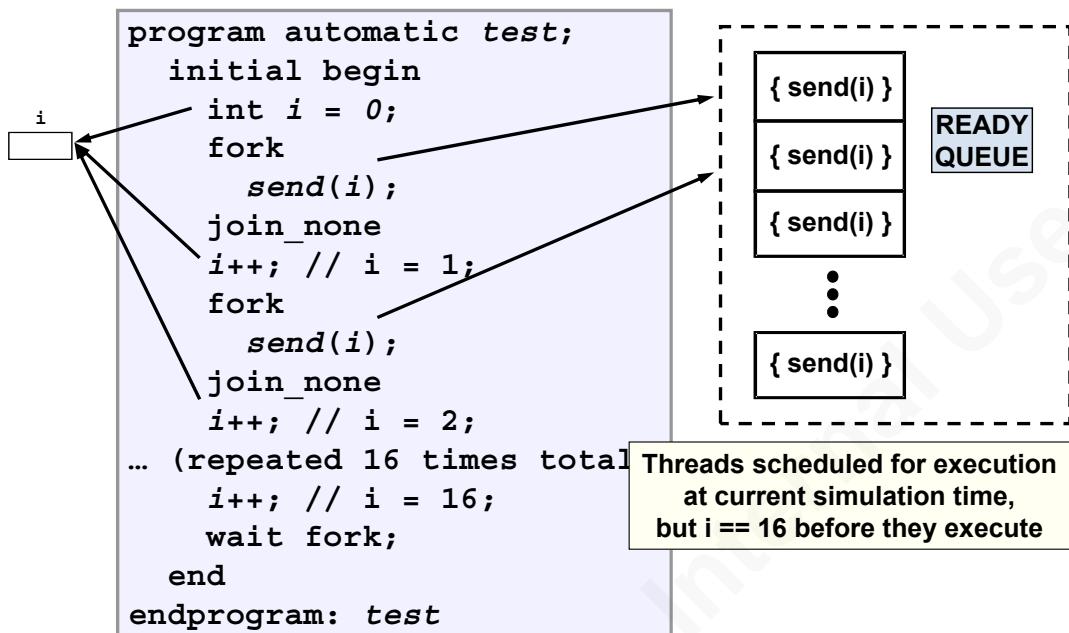
This is not the recommended usage of **fork-join-none** construct. The proper use is as shown a few pages back. For the problems with this type of usage, see the following page for explanation.

As per the IEEE LRM 1800-2009,

“The **wait fork** statement blocks process execution flow until all immediate child subprocesses (processes created by the current process, excluding their descendants) have completed their execution.”

- Prior to 2015.09 release, VCS blocks the process execution flow until all the child processes, not just immediate child processes complete execution. That VCS behavior is not compliant to LRM
- To make VCS behave as prior releases use the following switch in synopsys\_bc.setup file
  - BC ID SV\_WAIT\_FORK

## Thread Execution Issues: Unroll the for-loop



5-18

Since there is only a single allocation of memory for **i**, every time this **i** is modified, the modification overrides the previous value. Unexpected corrupted results occur.

## Thread Execution Issues: Local Variable

- Local variables once created are local to the child context

- Can copy parent variable in `fork` declarative space

```
program automatic test;
    initial begin
        for (int i = 0; i < 16; i++) begin
            fork
                int index = i; // local fork variable
                send(index);
            join_none
        end
        wait fork;
    end
    task send(int j);
        $display("Driving port %0d", j);
    ...
endtask: send
endprogram: test
```

Desired output:

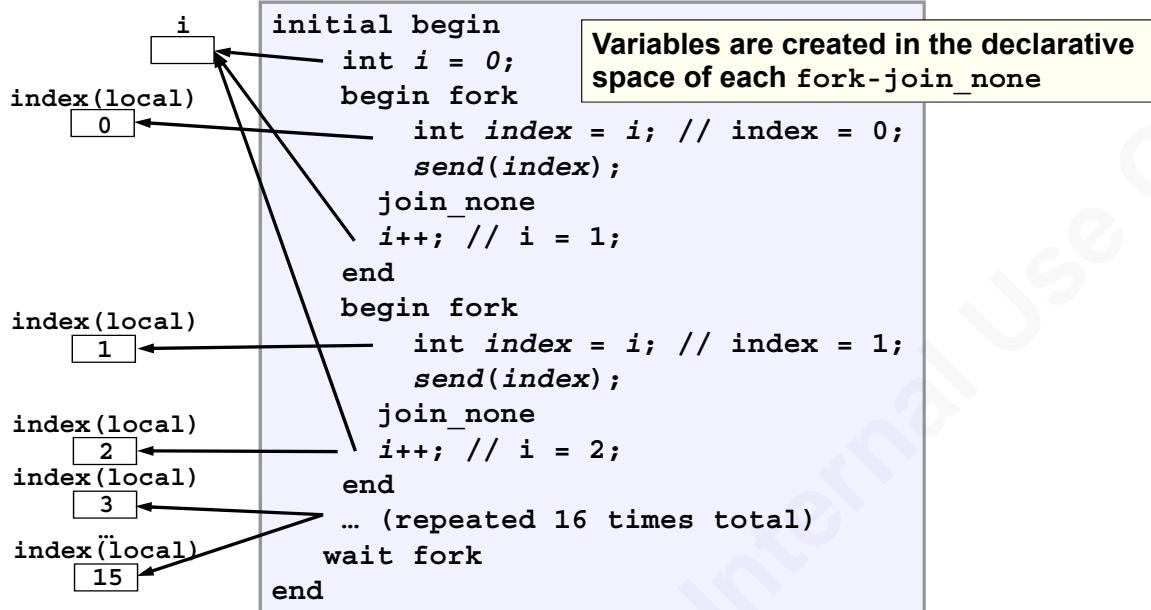
```
Driving port 0
Driving port 1
Driving port 2
Driving port 3
Driving port 4
Driving port 5
Driving port 6
Driving port 7
...
```

5-19

The variable `index` is in the local declarative space of the fork. Variables declared here are available to all threads of the fork. In this case there is only one thread per `fork-join_none`

See following page for illustration.

## Thread Execution Issues: Unroll the for-loop

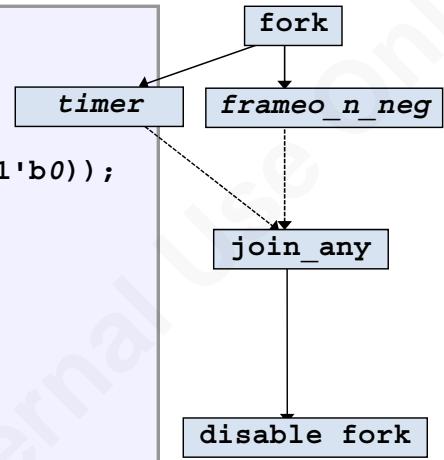


5-20

# Implement Watch-Dog Timer with join\_any

- Typically used in conjunction with `disable fork`

```
task recv();
    fork
        begin: frameo_n_neg
            wait (rtr_io.cb.frameo_n[7] !== 1'b0);
            @(rtr_io.cb iff(rtr_io.cb.frameo_n[7] === 1'b0));
        end
        begin: timer
            repeat(1000) @(rtr_io.cb);
            $display("Timed out!");
            $finish;
        end
        join_any
        disable fork; // kills all child threads!
        get_payload();
    endtask
```



5-21

As per the IEEE LRM 1800-2009,

“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.”

- Prior to 2015.09 release, VCS disabled only the immediate children processes. That VCS behavior is not compliant with LRM.
- To make VCS behave as prior releases use the following switch in synopsys\_bc.setup file
  - BC ID SV\_DISABLE\_FORK

# Avoiding disable fork Problems

## ■ Use enclosing fork join to localize disable fork

```
task recv();
    fork begin // enclosing fork-join
        fork: recv_wd_timer
            begin: frameo_n_neg
                wait (rtr_io.cb.frameo_n[7] !== 1'b0);
                @(rtr_io.cb iff(rtr_io.cb.frameo_n[7] === 1'b0));
            end
            begin: timer
                ...; $finish;
            end
            join_any
            disable fork; // kill all child threads
        // disable recv_wd_timer // LEGAL BUT DO NOT USE!
        end join
        get_payload();
    endtask
```



See Note

5-22

Another way to manage this is to label the **fork-join** construct and disable just the named **fork-join** block. Example:

```
// !!!Warning!!! This method does not work for threads started inside objects because it will kill the
labeled thread of all objects of the same class. Use this method only outside classes.
```

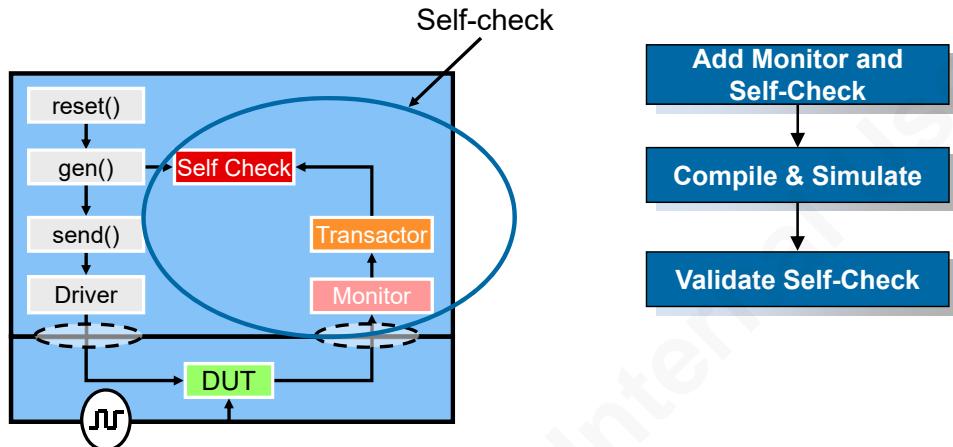
```
program automatic router_test(router_io.TB rtr_io);
    initial begin
        fork
            begin #100 ns $display($time); end
        join_none
        fork: test1
            begin $display("This is a test"); end
            begin #10 ns $display("Did I execute?"); end
        join_any : test1
        disable test1;
    //    disable fork;
    //    wait fork;
    //    $display($time);
    end
endprogram
```

# Lab 3 Introduction



90 minutes

## SystemVerilog Verification Flow



5-23

## Unit Objectives Review

**Having completed this unit, you should be able to:**

- **Divide a testbench into multiple current threads to execute parallel tasks**



**5-24**

## **Appendix**

### **Alternatives to disable fork**

**5-25**

## **Alternatives to disable fork**

**5-26**

## Alternatives to disable fork - kill()

- SystemVerilog allows you to access and store handles to threads created by the fork-join structure
  - The mechanism is a data type called `process`
  - Use `process::self()` to retrieve and store the handle to the thread where the method is called
  - The thread handle can then be used to kill the thread using the `kill()` method

5-27

# Get and Save Thread Process Handle

- Store thread process handles

```
task recv();
  process thread_q[$];
  fork
    begin
      thread_q.push_back(process::self());
      wait (rtr_io.cb.frameo_n[7] !== 1'b0);
      @(rtr_io.cb) iff(rtr_io.cb.frameo_n[7] === 1'b0));
    end
    begin
      thread_q.push_back(process::self());
      repeat(1000) @(rtr_io.cb);
      $display("Timed out!");
      $finish;
    end
  join_any
// see next slide for remaining code
```

5-28

# Managing Time-Consuming Threads

- For threads which consume time
  - Use process `kill()` method to terminate thread for all processes in queue

```
...continued from previous slide
foreach (thread_q[i])
    if (thread_q[i].status != process::FINISHED)
        thread_q[i].kill();
thread_q.delete(); // once killed, remove from queue
endtask
```

Clean up threads

5-29

# Managing Non-Time-Consuming Threads

- For threads which may terminate in 0 simulation time
  - Use dynamic array to store thread process handles
  - Make sure all threads had a chance to start before using `kill()` method to terminate remaining threads

```
task recv();
process threads[] = new[2];
foreach threads[i] begin
    fork
        int thread_index = i;
    begin
        threads[thread] = process::self();
        case thread_index
            0: begin wait (...); ...; end
            1: begin ... end
        endcase
    end
    join_any
end
foreach (threads[i]) wait (threads[i] != null);
// Clean up threads - see previous slide and note
endtask
```

Use dynamic array to store thread process handle

Threads which may terminate in 0 simulation time

Ensure thread process started.  
Otherwise, `kill()` may not work properly.

5-30

Note on `kill()` method.

From IEEE LRM:

"The `kill()` function terminates the given process and all its subprocesses, that is, processes spawned using `fork` statements by the process being killed. If the process to be terminated is not blocked waiting on some other condition, such as an event, `wait` expression, or a delay, then the process shall be terminated at some unspecified time in the current time step."

So, if the thread you are about to kill needs additional processing at that simulation time, you will need to supplement your thread code with additional control.

# Agenda

DAY  
2

5 Concurrency



6 Object Oriented Programming (OOP)  
– Encapsulation

7 Object Oriented Programming (OOP)  
– Randomization



Synopsys 50-I-052-SSG-016

© 2019 Synopsys, Inc. All Rights Reserved

6-1

# Unit Objectives

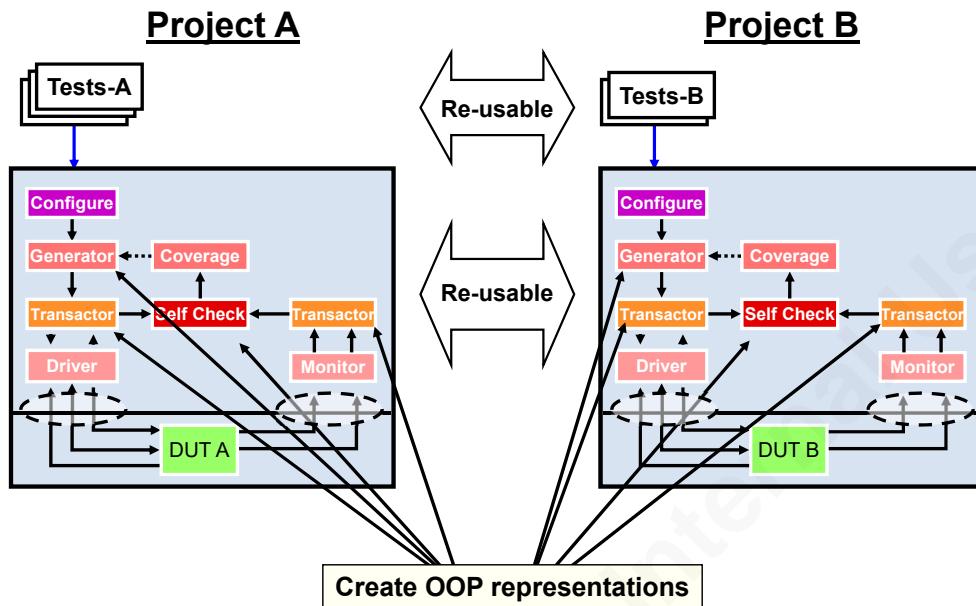


**After completing this unit, you should be able to:**

- **Raise level of abstraction by building data structure with self-contained functionality:**
  - Object-Oriented Programming(OOP) encapsulation
- **Protect integrity of data in OOP data structure:**
  - OOP data hiding
- **Simplify data initialization process:**
  - OOP constructor
- **Define a parameterized class**
- **Define and use SystemVerilog packages**

6-2

# Abstraction Enhances Re-Usability of Code



6-3

The components of a testbench are similar to modules in a design. We can encapsulate certain functions inside a container similar to a module, called a **class**. We can instantiate a **class** creating an object that represents the instance of the class. We can now manipulate the object or “run” it. OOP allows the team to cope with the complexity of a large verification effort consisting of many components and team members.

# SystemVerilog OOP Program Constructs

- Building SystemVerilog OOP structure is similar to building Verilog RTL structure

	RTL	OOP
Block definition	<code>module</code>	<code>class</code>
Block instance	instance	object
Block name	instance name	object handle
Data types	registers and wires	variables
Functionality	tasks, functions behavioral blocks ( <code>always</code> , <code>initial</code> )	subroutines (tasks, functions)



Unlike in a module, nothing executes automatically in an object.  
Some subroutine in the object must be called to perform an action.

6-4

# OOP Encapsulation (OOP Class)

- Similar to a module, an OOP **class** encapsulates:
  - Variables (**properties**) used to model a system
  - Subroutines (**methods**) to manipulate the data
  - Properties & methods are called **members** of class

Class properties and methods are visible inside the class

```
class Packet;
    string     name; //Packet properties
    bit[3:0]   sa, da;
    bit[7:0]   payload[];

task send(); //Packet methods
    send_addrs();
    send_pad();
    send_payload();
endtask: send

task send_addrs(); ... endtask
task send_pad(); ... endtask
task send_payload(); ... endtask
endclass: Packet
```

6-5

Unlike procedural languages, classes allow data and functionality to be grouped together. This helps in creating and maintaining large verification projects.

## module vs. class

### ■ Why use class?

- Objects are dynamic, modules are static
  - ◆ Objects are created and destroyed as needed
- Instances of classes are objects
  - ◆ A handle points to an object (class instance)
  - ◆ Object handles can be passed as arguments
  - ◆ Object memory can be copied or compared
  - ◆ Instances of modules can not be passed, copied or compared
- Classes can be inherited, modules can not
  - ◆ Classes can be modified via inheritance without impacting existing users
  - ◆ Modifications to modules will impact all existing users

6-6

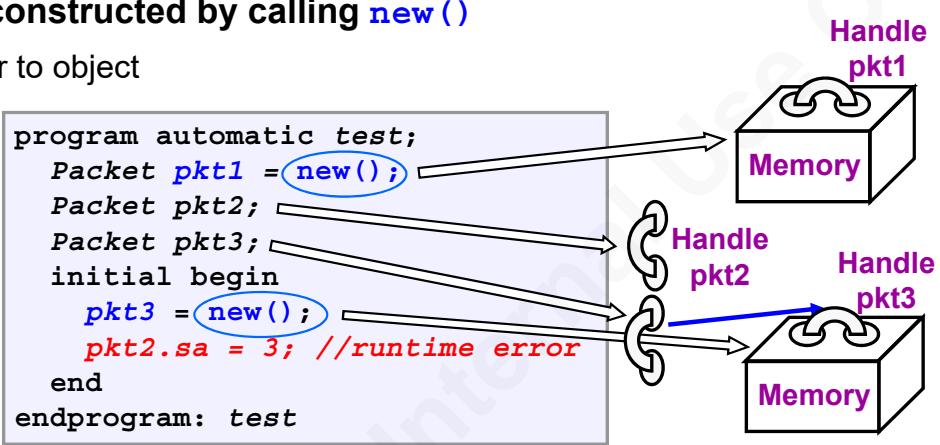
One of the goals of using classes is to improve team productivity. Object oriented code is layered. One layer inherits from the other. Common-currency components and layers form the framework, which other components and layers are plugged into, in a consistent, standardized manner. VMM, UVM, etc. are available open-source frameworks using OOP.

# Constructing OOP Objects

- OOP objects are **constructed** from class definitions
  - Similar to instance creation from module definition
- Object memory is constructed by calling **new()**
  - Handle used to refer to object

```
class Packet;  
  bit[3:0] sa, da;  
  byte payload[];  
  task send();  
endclass: Packet
```

```
program automatic test;  
  Packet pkt1 = new();  
  Packet pkt2;  
  Packet pkt3;  
  initial begin  
    pkt3 = new();  
    pkt2.sa = 3; //runtime error  
  end  
endprogram: test
```



6-7

In the example above, `pkt2` has not been constructed. So no memory is allocated. Thus the reference to `pkt2.sa` will fail at runtime. Remember that a compiler does not execute code. Objects being dynamic, object memory is only allocated at runtime by calling `new()`.

# Accessing Object Members

- Object memory is created by a call to new ()
- Object members are accessed using the object handle
  - Similar to accessing RTL instance signals and subroutines
  - Accessed via dot (.) notation

```
program automatic test;
  class Packet;
    bit[3:0] sa, da;
    byte payload[];
    task send();
    ...
  endclass: Packet
  Packet pkt;
  initial begin
    pkt = new();
    pkt.sa = 3; // access property
    pkt.da = 7; // access property
    pkt.send(); // access method
  end
endprogram: test
```

6-8

# Initialization of Object Properties

- Define constructor `new()` in class to initialize properties
  - No return type in declaration
  - Executes immediately after object memory is allocated
  - Not accessible via dot (.) notation
- If constructor `new()` is not defined
  - A default constructor is implemented as follows:

```
function new();endfunction
```

```
program automatic test;
  class Packet;
    bit[3:0] sa, da;
    bit[7:0] payload[];
    function new(bit[3:0] init_sa, init_da,
                int init_payload_size);
      sa = init_sa;
      da = init_da;
      payload = new[init_payload_size];
    endfunction: new
  endclass: Packet
  initial begin
    Packet pkt1 = new(3, 7, 2);
    pkt1.new(5, 8, 3); // syntax error!
  end
endprogram
```

6-9

# Initialization of Object Properties: **this**

## ■ **this** keyword

- An object's handle to itself
- Unambiguously refers to **class** members of the current instance (object)
  - ◆ More readable – allows method arguments to have same name as **class** variables

```
class Packet;
    bit[3:0] sa, da;
    bit[7:0] payload[];
    function new(bit[3:0]sa, da, int payload_size);
        this.sa = sa;
        this.da = da;
        this.payload = new[payload_size];
    endfunction: new
endclass: Packet
```

6-10

The keyword **this** can be used even when not necessary, to clarify portions of code.

Note: A variable referred to inside a subroutine is searched starting locally in the subroutine. If not found, it is searched for in the local object memory, then in the parent's memory and above, up to the base class, and finally in the **program** global space. If not found the compiler gives an error. When using the keyword **this**, the local scope of the subroutine is ignored as is the **program** global scope. Only the class hierarchy is searched.

**this** can also be used by the class when passing a handle to itself as a subroutine argument.

A subroutine's arguments are variables in the subroutine's local scope.

## OOP Data Hiding (Integrity of Data) 1/3

- Unrestricted access to object properties can cause unintentional data corruption

```
program automatic test;
initial begin
  driver drv = new();
  drv.max_err_cnt = -1; // directly set max_err_cnt
  drv.run();           // Will this work?
end
endprogram: test
```

```
class driver;
  int max_err_cnt = 0, err_cnt = 0;
  task run();
    ...
    if (error_cond()) err_cnt++;
    if ((max_err_cnt != 0) &&
        (err_cnt >= max_err_cnt))
      $finish;
  endtask
  function new(); // details not shown
endclass: driver
```

Are all class data correct?

6-11

## OOP Data Hiding (Integrity of Data) 2/3

- Properties & methods can be protected using `local`

- Object members are public by default
- `local` members of object can be accessed only in class

```
program automatic test;
    class driver;
        local int max_err_cnt = 0, err_cnt = 0;
        task run();... endtask
    endclass: driver
    initial begin
        driver drv = new();
        drv.max_err_cnt = -1; // Compile error!
        drv.run();
    end
endprogram: test
```

6-12

## OOP Data Hiding (Integrity of Data) 3/3

- Create public class method to allow users to access local members
  - Ensure data integrity within the method

```
program automatic test;
initial begin
  driver drv = new();
  drv.set_max_err_cnt(-1); // No Compile error
  drv.run();
end
endprogram: test
```

```
class driver;
  local int max_err_cnt = 0, err_cnt = 0;
  task run();... endtask
  function set_max_err_cnt(int max_err_cnt);
    if (max_err_cnt < 0) begin
      this.max_err_cnt = 0;
      return;
    end else this.max_err_cnt = max_err_cnt;
  endfunction
endclass: driver
```

Ensure integrity  
of object data

6-13

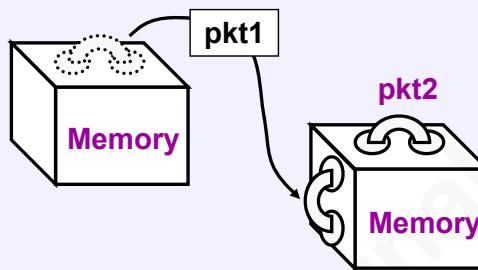
# Working with Objects – Handle Assignment

- What happens when one object handle is assigned to another?
  - Like any variable, the target takes on the value of the source.

```
class Packet;
    int payload_size;
    ...
endclass: Packet

...
Packet pkt1 = new();
Packet pkt2 = new();
...

pkt1 = pkt2; //handle copy, not object copy
pkt1.payload_size = 5; // whose payload_size is set ?
```



What happens to the pkt1 object memory?

6-14

Remember that subroutines can have object handles as arguments. When object handles are passed to subroutines, a copy of the handle is made which is an assignment similar to the one above.

```
task transmit(Packet pkt);
    pkt.send();
endtask: transmit

initial begin
    Packet mypkt = new();
    transmit(mypkt); //does a pkt = mypkt at the call to transmit
end
```

# Working with Objects – Garbage Collection

- VCS **garbage collector** reclaims memory automatically:

- When an object memory is no longer accessible and the object has no scheduled event

- Object can be manually dereferenced using

```
pkt1 = null;
```

This method of copying is not recommended.  
Normally every class that needs it should provide a `copy()` (or similar) method

```
pkt1_copy = pkt1.copy();
```

- Making an exact duplicate of object memory:

```
class Packet;  
    int count;  
    Payload p; // encapsulated object  
endclass: Packet  
...  
Packet pkt1 = new();  
Packet pkt1_copy; // handle only  
pkt1.p = new();  
// construct pkt1_copy and  
// copy contents of pkt1 to pkt1_copy  
pkt1_copy = new pkt1;  
// shallow copy  
// (encapsulated objects not copied)  
// object pkt1 must exist
```

6-15

Every time the constructor `new` is called, new memory is allocated. The handle is removed from the old memory and re-assigned to the new memory. If there are no events scheduled in the old memory, the old object memory is reclaimed by the garbage collector.

The garbage collector is not instantaneous. It is invoked at regular intervals so as to not interrupt the simulator. Generally the garbage collector is invoked when simulation time advances. VCS can even invoke the garbage collector in zero-delay loops. There may be some rare instances when garbage (unreachable memory) can grow in delay-free code with the collector unable to collect it.

A deep copy, where embedded objects are copied to any depth, is not specified in the SystemVerilog language.

# Working with Objects – static Members

- Variables and subroutines can be defined using **static** keyword
  - ◆ Associated with the class, not object
  - ◆ Shared by all objects of that class
  - ◆ Can be accessed using `class_name::`
- Static subroutines
  - ◆ Can only access **static** members
  - ◆ Can not be **virtual**
- Static members are allocated and initialized at compile

```
program automatic test;
initial begin
    Packet pkt0 = new();
    Packet pkt1 = new();
    $display("pkt0 id is: %0d", pkt0.id);
    $display("pkt1 id is: %0d", pkt1.id);
    $display("count: %0d" Packet::count);
end
endprogram: test
```

What values get printed?

```
class Packet;
    static int count = 0;
    int id;
    static function int get_count();
        return count;
    endfunction
    function new();
        this.id = count++;
    endfunction
endclass: Packet
```

6-16

The code above displays:

**pkt0 id is: 0**  
**pkt1 id is: 1**  
**count: 2**

For static subroutines, you can pass arguments to the function which are local automatic function variables. For example

```
static function void print_note(string note);
    $display("NOTE: %s", note);
endfunction
```

IMPORTANT:

```
static function void print_note(string note); //common in classes
```

is NOT the same as

```
function static void print_note(string note); //rarely used inside classes
```

## Working with Objects – const Properties

- Use **const** keyword to define constant properties that can not be modified
  - ◆ Global constant – typically also declared **static**
  - ◆ Instance constant – can not be **static**

```
class Packet;  
    static int count = 0;  
    const int id; // instance constant  
    static const string type_name = "Packet"; // global constant  
    function new();  
        this.id = count++; // instance constant can only be assigned in new()  
    endfunction  
endclass  
program automatic test;  
    initial begin  
        Packet packet0 = new();  
        packet0.id = 0; // Compile error – can not change const property  
        packet1.type_name = "newPacket"; // Compile error  
    end  
endprogram: test
```

6-17

# Working with Objects – Array Methods

```
class Packet;
    rand bit [7:0] payload[]; // Data
    rand bit [2:0] pr;           // user-defined priority 0-7
    rand bit [15:0] addr;       // Address
endclass: Packet

Packet pq[$]; // Queue of packet handles
initial begin
    int len;
    generate_packet_queue(pq); // Some user-defined method
    // Sort objects using user-defined priority property in class
    pq.sort(pkt) with (pkt.pr); // pkt is user-defined, auto-declared iterator
    // Find total length of all payloads – item is default iterator
    len = pq.sum() with (item.payload.size()); ! See Note
end
```



6-18

Examples of other array methods are: (please check release note and LRM for support)

`find_last()`, `find_last_index()`

`unique()`, `unique_index()`

`sum()`, `product()`

`and()`, `or()`, `xor()`

`min()`, `max()`

`reverse()`, `sort()`, `rsort()`, `shuffle()`

Caution: When you use aggregate methods like `sum()`, `product()`, etc. the LRM states

“The array reduction methods can produce a single integral value from an unpacked array of integral values over each element of the array, joined by the relevant operand for each method. The result returns a single value of the same type as the array element type or, if specified, the type of the expression in the `with` clause”.

`len = pq.sum(pkt) with (pkt.payload.size());`

returns an int type containing the sum of all payload sizes because `payload.size()` is of type int, but

`len = pq.sum(pkt) with (pkt.payload.size() < 10 );`

returns the sum of the number of packets whose payload size is less than 10, as a bit type since the expression `pkt.payload.size() < 10` is a bit type!!!

`len = pq.sum(pkt) with (int'(pkt.payload.size() < 10 ) );`//typecast expression  
returns the sum of the number of packets whose payload size is less than 10 as an int.

# Working with Objects – Concurrency

- **Classes can not have `initial` or `always` blocks**
- **Spawn a process similar to an `always` block with `fork-join_none`**
- **Standard methodology**
  - Program calls `run()` method of the various OOP testbench components
    - ◆ Generator, Monitor, Driver, Scoreboard etc.

```
class Driver;  
...  
task run(); //thread start method  
fork //emulate always block  
    forever  
        send();  
join_none  
endtask: run  
...  
endclass: Driver
```

6-19

The concurrent components of a testbench in SystemVerilog are created using objects constructed from classes. Within classes you can not have `initial` blocks or `always` blocks, but you can have methods (tasks and functions) that emulate an `always` block as shown.

# Parameterized Classes

## ■ Written for generic types and/or values

- Parameters passed at instantiation, just like parameterized modules
- Allows reuse of common code

```
program automatic test;
    stack addr_stack; //default type
    stack #(Packet, 128) data_stack;
initial begin
    ...
repeat(addr_stack.size()) begin
    Packet pkt = new();
    if(!pkt.randomize()) $finish;
    pkt.addr = addr_stack.pop();
    data_stack.push(pkt);
end
end
endprogram: test
```

```
class stack #(type T = int,
             bit[11:0] depth = 1024);
protected T items[$:depth];
function void push( T a );
...
function T pop();
function int size(); ...
endclass: stack
```

6-20

Parameterized classes allow you to write many types of generic components. e.g. Generator

```
class Generic_Generator #(type T = Packet); //type T can be left undefined
...
T randomized_obj;
...
while(pkt_cnt < run_for_n_pkts) begin
    T pkt;
    if(!randomized_obj.randomize()) $finish;
    $cast(pkt, randomized_obj.copy());
    out_chan.put(pkt);
end
...
endclass: Generic_Generator
```

One benefit of parameterized classes is that checking is done at compile time for type safety.

Parameterized classes can be extended just like other classes.

## forward typedef

### ■ A forward typedef

- Is needed to use a class before declaration
  - ◆ e.g. two classes need handle to each other

This is a compile error  
if **typedef** is missing

- Can be used for many data types  
(see Note)

```
typedef class Packet;
class Generator;
Packet p1;
function new();
  p1 = new();
  p1.mygen = this;
endfunction
endclass: Generator

class Packet;
  bit[7:0] sa, da, payload[$]
  Generator mygen;
endclass: Packet
```

6-21

A forward typedef can be used for many data types as shown below

```
typedef enum type_identifier;
typedef struct type_identifier;
typedef union type_identifier;
typedef class type_identifier;
typedef interface class type_identifier;
typedef type_identifier;
```

## Best Practices (1/2)

- Methods can be placed outside of the `class` definition

- Inside `class` block, declare an `extern` prototype
- Outside `class` block, associate the method with its class
  - ◆ Use double-colon `::` (a scope/name resolution operator)

```
class node;
  static int count = 0;
  string str;
  node next;
  ...
  task ping();
  ...
endtask: ping
endclass: node
```

```
class node;
  static int count = 0;
  string str;
  node next;
  ...
  extern task ping(); // prototype
endclass: node
task node::ping();
  ...
endtask: ping
```

Place class name and  
double-colon before  
method name

6-22

The advantage is that this allows you to have all the properties and method headers on the first page of code that you see when you open the file. For the same prototype function, it also allows you to define different functionality for different tests or projects.

## Best Practices (2/2)

- **Useful methods for Data classes (user defined)**

- **display()**
  - ◆ Print object variables to console - helpful for debugging
- **compare()**
  - ◆ Returns match, mismatch, other status based by comparing object variables to variables of another object
  - ◆ Simplifies self-check
- **copy()**
  - ◆ Copy selected variables or nested objects
  - ◆ Allows you to do deep copy if required

- **Use `typedef` to create shortcuts**

- `typedef stack#(Packet) pkt_stack;`
  - ◆ Now use `pkt_stack` instead of `stack#(Packet)`

6-23

# Virtual Interfaces

## ■ Classes need to drive/sample signals of interface

- Interfaces can not be created at object construction

Need to create a **virtual** reference to interface

```
class Driver;
    virtual router_io.TB rtr_io_ref;           Create virtual
                                                reference to interface
    ...
    function new(virtual router_io.TB rtr_io_arg);
        this.rtr_io_ref = rtr_io_arg;           Pass virtual connections
                                                via constructor argument*
    endfunction: new
    task send_addrs();
        this.rtr_io_ref.cb.frame_n[sa] <= 1'b0; Drive/Sample signals
        for(int i=0; i<4; i++) begin           using virtual interface
            this.rtr_io_ref.cb.din[sa] <= da[i];
            @(this.rtr_io_ref.cb); program automatic test(router_io.TB rtr_io);
            ...
    endtask: send_addrs
endclass: Driver
```

6- 24

Note: While a class must define a virtual interface for compilation and use, how it is set depends on the methodology used. In this example we are setting it using an argument passed to the constructor. Different methodologies may use different ways to set the virtual interface reference member of a class.

## SystemVerilog Packages

- **Package is a mechanism for sharing among modules, programs and interfaces and other packages the following:**
  - Parameters
  - Data – variables and nets
  - Type definitions
  - Tasks & functions
  - Sequence and property declarations
  - Classes
- **Declarations may be referenced within modules, interfaces, programs, and other packages**

6-25

The purpose of a package is to create multiple namespaces for identifiers that make them either visible or invisible to the user.

## Packages: Example

```
package ComplexPkg;                                ComplexPkg.sv
class Complex;
    float i, r;
    extern virtual task display(); // not shown
endclass: Complex
// standalone functions
function automatic Complex add(Complex a, b);
    add = new();
    add.r = a.r + b.r; add.i = a.i + b.i;
endfunction: add

function automatic Complex mul(Complex a, b);
    mul = new();
    mul.r = (a.r * b.r) - (a.i * b.i);
    mul.i = (a.r * b.i) + (a.i * b.r);
endfunction: mul

endpackage: ComplexPkg
```

6-26

# Rules Governing Packages

- **Packages are explicitly named scopes appearing at the outermost level of the source text (at the same level as top-level modules and primitives)**
- **Packages must not contain any processes**
  - Wire declarations with implicit continuous assignments are not allowed
- **Packages can not have hierarchical references**
- **Variable declaration assignments within the package**
  - Must occur before any `initial`, `always`, `always_comb`, `always_latch`, or `always_ff` blocks are started



Subroutines defined in a package are `static` unless explicitly made `automatic`.  
Classes are always `automatic`.

6-27

Synopsys Design Compiler can use packages. It requires all standalone tasks and functions (outside classes) in packages to be `automatic`.

# Using Packages

- Directly reference package member using class scope resolution operator `::`

```
ComplexPkg::Complex cout = ComplexPkg::mul(a,b);
```

- `import` package into appropriate scope

- Explicit import of specific symbols

```
import ComplexPkg::Complex;  
import ComplexPkg::add;
```

- Implicit import of all symbols in package

```
import ComplexPkg::*;

◆ Now all symbols in ComplexPkg are visible
```

- OK to import same package in multiple locations

```
◆ `include cannot be used in multiple places
```

6-28

Packages must exist in order for the items they define to be recognized by the scopes in which they are imported. This means they must either be separately compiled beforehand or the files must be ordered correctly during compilation.

## Using Packages: Example (1/2)

```
package ComplexPkg;
class Complex;
    float i, r;
    extern virtual task display(); // not shown
endclass: Complex
// standalone functions
function automatic Complex add();
    Complex add = new();
    add.r = 0;
    add.i = 0;
endfunction
function automatic Complex multiply(Complex a, Complex b);
    Complex result;
    result.r = a.r * b.r - a.i * b.i;
    result.i = a.r * b.i + a.i * b.r;
    return result;
endfunction
// implicit import all symbols
module dut(if.dut_port dut_io);
    import ComplexPkg::*;

    Complex l,m,n;
    ...
endmodule: dut
```

**ComplexPkg.sv**

**import whole package**

**import specific symbols**

```
// import of specific symbol
program automatic
    test(if.tb_port tb_io);
        import ComplexPkg::Complex ;
        ...
    endprogram: test
```

**Direct reference using ::**

```
// Direct reference
class harmonix;
    ComplexPkg::Complex i,j ;
    ...
endclass: harmonix
```

6-29

## Using Packages: Example (2/2)

- Packages can be imported by other packages
- To allow a package imported by one package to be imported along with the importing package, **export** it
  - **export** follows same syntax as **import**

```
package signal_analysis;  
import ComplexPkg::*;

// export with signal_analysis  
export ComplexPkg::*;

class harmonix;  
Complex alpha, beta, gamma;  
...  
endclass: harmonix

endpackage: signal_analysis
```

6-30

## Quiz Time ?

6-31

## OOP: Quiz 1

```
program automatic test1 ;  
  
class abc ;  
    int a = 10 ;  
    function new(int a) ;  
        a = a ;  
    endfunction  
endclass  
  
abc o1 ;  
  
initial  
begin  
    o1 = new(5) ;  
    $display("a = %0d",o1.a) ;  
end  
endprogram: test1
```

1. What will the program display?
2. Did it display what you expected?
3. How will you fix this?

6-32

1. a = 10
2. No. The variable in function new is not the class variable. So the new did not change the class variable.
3. Use the handle this, which refers to the object itself.
- ```
class abc;  
int a = 10;  
function new(int a);  
    this.a = a;  
endfunction  
endclass
```

## OOP: Quiz 2

```
program automatic test1 ;
class abc ;
    int a ;
endclass

initial begin
    abc o1 = new() ;
    abc o2 = new() ;

    o1.a = 5 ;
    o2.a = 50 ;

    $display("A: %0d %0d", o1.a, o2.a) ;
    o2 = o1 ;
    o1.a = 500 ;
    $display("B: %0d %0d", o1.a, o2.a) ;
end
endprogram: test1
```

1. What will the program display?
2. Why?
3. How many objects each at first and second \$display lines?
  - Why?
4. If number of objects is less, what happened to missing objects? If it is more, how did more objects get constructed?

6-33

1. A: 5 50  
B: 500 500
2. `o2 = o1` is a handle copy. So both handles represent same object at second display.
3. Two objects at first display. One object at second display, because now handle `o2` points to object `o1`
4. The missing object will be garbage collected i.e. its memory will be reused by simulator for other purposes.

## OOP: Quiz 3

- What is the difference between a public and local member of a class?
- Can local members be static?
- What is the :: operator?
  - Give some examples where it can be used
- List two uses of `typedef`

6-34

1. Public members are accessible to all users of the class. Local members are only visible to other class members.
2. Yes.
3. The :: operator is the class scope resolution operator. It can be used to define external method prototypes outside of the class. It can also be used to refer to static members of the class without having to construct an object of the class. It can also be used to access typedefs in the class.
4. `typedef` can be used to
  - to provide compilers with forward declarations for classes and other data types
  - create aliases for complex expressions and types

## Unit Objectives Review

**Having completed this unit, you should be able to:**

- **Raise level of abstraction by building data structure with self-contained functionality:**
  - Object-Oriented Programming(OOP) encapsulation
- **Protect integrity of data in OOP data structure:**
  - OOP data hiding
- **Simplify data initialization process:**
  - OOP constructor
- **Define a parameterized class**
- **Define and use packages**



6-35

## **Appendix**

### **SystemVerilog Virtual Interface Singleton Objects**

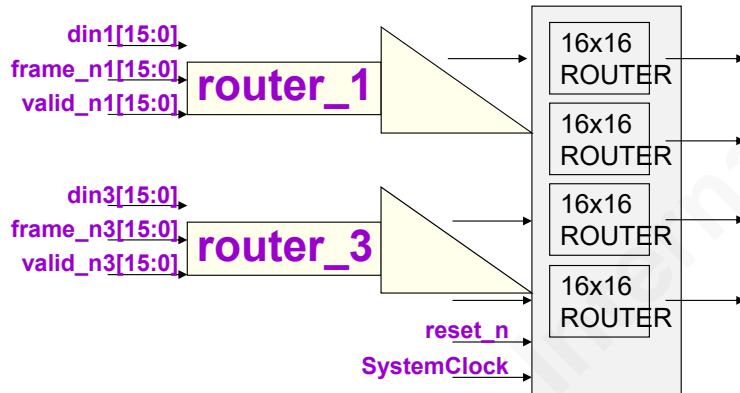
**6-36**

## **SystemVerilog Virtual Interface**

**6-37**

## Virtual Interfaces (1/5)

- Allow grouping of signals by function
- Create a handle to an interface
  - Virtual interfaces can be passed via routine argument
- Promotes reuse by separating testbench from implementation names



6-38

## Virtual Interfaces (2/5)

### ■ STEP 1: Define a physical interface

- Similar to creating interface for just the single instance
- The difference is that common connections are removed

```
interface router_io(input bit clock);
    // logic reset_n;
    logic [15:0] din, frame_n, valid_n;
    logic [15:0] dout, valido_n, frameo_n;

    clocking cb @(posedge clock);
        output din, frame_n, valid_n;
        input dout, valido_n, frameo_n;
    endclocking: cb

    modport TB(clocking cb);
    // instead of modport TB(clocking cb, output reset_n);
endinterface: router_io
```

6-39

## Virtual Interfaces (3/5)

### ■ STEP 2: Connect the interface

```
...
router dut(
    .clock      (SystemClock),
    .reset_n    (reset_n),
    .din0       (io_0.din),
    .frame_n0   (io_0.frame_n),
    .valid_n0   (io_0.valid_n),
    ...
    .din1       (io_1.din),
    .frame_n1   (io_1.frame_n),
    .valid_n1   (io_1.valid_n),
    ...
);
endmodule: router_test_top
```

```
module router_test_top;
    bit SystemClock;
    logic reset_n;
    router_io io_0(SystemClock);
    router_io io_1(SystemClock);
    router_io io_2(SystemClock);
    router_io io_3(SystemClock);
    test t(io_0, io_1, io_2, io_3, reset_n);
    ...

```

One interface per instance

Connect common signals separately

Connect unique signals with a specific interface instance

## Virtual Interfaces (4/5)

- STEP 3: Pass virtual interface in via constructor
- STEP 4: Drive/Sample signals with virtual interface

- This class is now re-useable for any router instance

```
class Driver;
    string name;
    virtual router_io.TB rtr_io;
    ...
function new(string name = "Driver", virtual router_io.TB
router);
    this.name = name;
    this.rtr_io = router;
endfunction: new
...
...
virtual task send_addrs();
    rtr_io.cb.frame_n[sa] <= 1'b0;
    for(int i=0; i<4; i++) begin
        rtr_io.cb.din[sa] <= da[i];
        @(rtr_io.cb);
    end
endtask: send_addrs
endclass: Driver
```

Create reference  
to virtual interface

Pass virtual  
connections via  
constructor  
argument

Drive/Sample  
signals using  
virtual interface

6-41

## Virtual Interfaces (5/5)

### ■ STEP 5: Connect Virtual to physical

```
program automatic test(router_io.TB r0, r1, r2, r3, output logic reset_n);
    class BFM_environment;
        DriverClass driver[16];
        function new(virtual router_io.TB rtr_io);
            ...
        endfunction: new
    endclass: BFM_environment
    BFM_environment bfm[4];
    initial begin
        bfm[0] = new(r0);
        bfm[1] = new(r1);
        bfm[2] = new(r2);
        bfm[3] = new(r3);
        ...
    end
endprogram: test
```

Connect Virtual to physical

```
module router_test_top;
    logic SystemClock, reset_n;
    router_io io_0(...),io_1(...),io_2(...),io_3(...);
    test t(io_0,io_1,io_2,io_3,reset_n);
    router dut(...);
    ...
endmodule: router_test_top
```

6-42

## Singleton Objects

6-43

# Singleton Objects

- A singleton object is a globally accessible static object which provides customized service methods

- Created at compile-time
- Globally accessible at run-time
- Can have static and non-static members
- For convenience only

```
class service_class;  
    static service_class me = get();  
    static function service_class get();  
        if (me == null) me = new(); return me;  
    endfunction  
    protected function new();  
    endfunction  
    extern function void error (string msg);  
endclass
```

```
service_class service_object;  
service_object = service_class::get();  
service_object.error("A different error");
```

Singleton object **me**  
created at compile-time

**get** is globally  
accessible at run-  
time

non-static function **error**

6-44

A singleton object is a static object that is automatically constructed within the class. The user need not create any instance of the class outside of the scope of the class. This singleton object's handle can be retrieved with a user-defined method like the *get()* method. All infrastructure for the singleton object must be defined within the class.

The execution of the non-static methods will require the user to first retrieve the single object's handle then execute the method through the retrieved handle as shown above. You can also refer to the handle directly, e.g. *service\_class::get().error("A different error")*

# Agenda

DAY  
2

5 Concurrency



6 Object Oriented Programming (OOP)  
– Encapsulation

7 Object Oriented Programming (OOP)  
– Randomization



## Unit Objectives

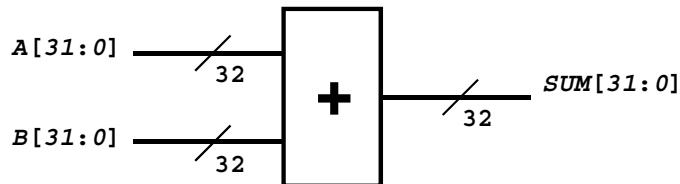


**After completing this unit, you should be able to:**

- Explain why randomization is needed in verification
- Randomize variables
- Constrain randomization of variables

7-2

# Alternatives to Exhaustive Testing?



32-bit adder example: Assume one set of input and output can be verified every 1ns.  
How long will exhaustive testing take?

- ◆ A day? – A week? – A year?\*

- **What if exhaustive testing is unachievable?**

- Answer: Verify design with a sufficient set of vectors to gain a level of confidence that product will ship with a tolerable field-failure rate

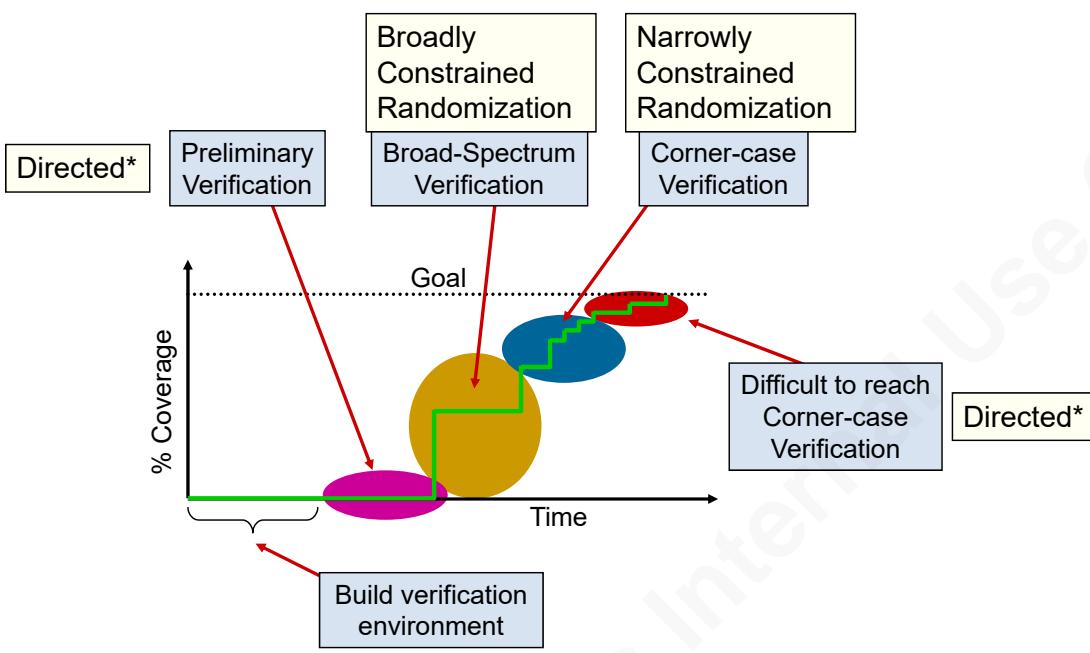
- **Best known mechanism is randomization of data**

7-3

\*It will take over 584 years to verify this simple adder.

The point is that exhaustive simulation testing is impossible!

# Process of Reaching Verification Goals



7-4

\* Randomization is always in use. Directed tests are essentially very highly constrained random tests.

## OOP Based Randomization

- In SystemVerilog, randomization is achieved via classes
  - `randomize()` function built into every class
- Two types of random properties are supported:
  - `rand` - Values can repeat without exhausting all possible values
    - ◆ Think “rolling dice”
  - `randc` - Exhaust all values before repeating any value
    - ◆ Think “picking a card from a deck of cards”
    - ◆ Can be as large as 32-bits in VCS
- When the class function `randomize()` is called:
  - Randomizes each `rand` and `randc` property value
    - ◆ To full range of its data type if no constraints specified

7-5

The SystemVerilog LRM limits `randomize()` to only generates integral types and packed structs. It does not specify randomization for non-integral types like reals, strings, class objects, unpacked structs and unions. VCS has limited support for randomizing real numbers.

# Randomization Example

```
program automatic test;
  int run_for_n_pkts = 100;
  Packet pkt = new();
  initial begin
    ...
    repeat (run_for_n_pkts)
      begin
        if(!pkt.randomize()) ...;
        fork
          send();
          recv();
        join
        check();
      end
    end
  endprogram: test
```

```
class Packet;
  randc bit[3:0] sa, da;
  rand  bit[7:0] payload[];

  function Packet copy(...);
    ...
  endfunction: copy
endclass: Packet
```

Declare random properties in class 1

Construct an object to be randomized 2

Randomize content of object 3

7-6

The `randomize()` function returns success or failure  
Always check the return value!

Do not do the following:

```
data_handle.randomize();
```

Do:

```
if (!data_handle.randomize())
begin
  $display("Error: data_handle randomization failure.
           Exiting");
  $finish;
end
```

Note: If you do the following,

```
assert(data_handle.randomize())
       else ...
```

the `assert` statement will be disabled if assertions are turned off using a run-time switch. This will disable object randomization.

Note: For `randc` properties to exhibit proper cyclic randomization, the same object must be randomized repeatedly.

## Controlling Random Variables (1/2)

- How do you control the value range for **sa** and **da**?
- How do you control the size of **payload[]**?

```
class Packet;
    randc bit[3:0] sa, da;
    rand bit[7:0] payload[];
    function void display();
        $display("sa = %0d, da = %0d", sa, da);
        $display("size of payload array = %0d", payload.size());
        $display("payload[] = %p", payload);
    endfunction: display
endclass: Packet
```

What does **pkt.display()** show  
for **sa**, **da** and **payload.size()**?

```
program automatic test;
    Packet pkt = new();
    if(!pkt.randomize()) $finish;
    pkt.display();
end
endprogram: test
```

What if **sa**, **da** are **int** type?  
rand int sa, da;

7-7

Since **sa** and **da** are unsigned four-bit type, the **sa** and **da** values will be in { 0:15 }.

Because memory for **payload** array is not allocated at object construction time, the size of **payload** array will always be 0.

If **sa** and **da** were to be changed to **int** type, then the full range of integer values will be possible. This means that there is a 50% chance for **sa** and **da** values to be negative numbers.

## Controlling Random Variables (2/2)

- Randomization can be controlled using constraint block

```
class Packet;
    rand bit[3:0] sa, da;  rand bit[7:0] payload[];
    constraint corner_test {
        sa == 12;           //equality operator, not assignment
        da inside {2,4,[6:10]}; //set membership
        payload.size() >= 2;   //array aggregate
        payload.size() <= 4;
    }
    constraint valid {...}
endclass: Packet
```

- Constraints support only 2-state values
- Multiple constraint blocks may be defined in same class
- Constraint expression must return true or false

```
constraint single_sa { sa = 12; } // Syntax error
```

7-8

# SystemVerilog Constraints

## ■ Relational Operators

```
constraint single_sa {  
    sa == 12;  
    da < sa ;  
}
```

## ■ Set Membership

- Select from a list or set with keyword inside

```
constraint Limit1 {  
    sa inside {[5:7], 10, 15};  
    // 5,6,7,10,15 equally weighted probability  
}
```

- Exclude from a specified set with !

```
constraint Limit2 {  
    !( sa inside {[1:10], 15} );  
    // 0,11,12,13,14 equally weighted probability  
}
```

7-9

# Weighted Constraints

- Constraint values can also be weighted over a specified range using keyword `dist` and:

- `:=` (apply the same weight to all values in range)

```
constraint Limit {  
    sa dist {[5:7]:=30, 9:=20};  
}  
// 5,6,7 have weight of 30 each, 9 has weight of 20
```

equal weights

- `:/` (divide the weight among all values in range)

```
constraint Limit {  
    da dist {[5:7]:/30, 9:=20};  
}  
// 5,6,7 have weight of 10 each, 9 has weight of 20
```

divided weights

7-10

Only `rands` variables can be constrained using weights. constraints of `randsC` variable cannot be weighted.

# Array Constraint Support

- Members can be constrained within `foreach` loop
- Aggregates can be used to constrain arrays
  - `size()`, `sum()` and more (see release notes)
- Set membership can be used to reference content

```
class Config;
    rand bit[7:0] addrs[10];  rand bit drivers_in_use[16];
    rand int num_of_drivers, one_addr;
    constraint limit {
        num_of_drivers inside { [1:16] };
        drivers_in_use.sum() with (int'(item)) == num_of_drivers;
        foreach(addr[ idx]) (idx > 0) -> addrs[idx] > addrs[idx-1];
        one_addr inside addrs;
    }
endclass: Config
```

See Note

7-11

Examples of other array methods are: (please check release notes and LRM for support. Some functions do not make sense in constraints. Not all are supported by VCS)

- .find\_last(), .find\_last\_index()
- .unique(), .unique\_index()
- .sum(), .product()
- .and(), .or(), .xor()
- .min(), .max()
- .reverse(), .sort(), .rsort(), .shuffle()
- .exists(), .num() (associative arrays)
- .first(), .last(), .prev(), .next() (associative arrays)

`drivers_in_use.sum()` returns the sum of all elements of the `drivers_in_use` array typed to the data type of the array `driver_in_use`. Since `drivers_in_use` is of type `bit` `drivers_in_use.sum()` returns only one bit!!

How about if we use `drivers_in_use.sum() with (item)`? This will also return a single bit sum since item is of type `bit`!

Hence we used `(int'(item))` to cast the `with` expression to an `int` type.

Randomization of multi-dimensional arrays can also be constrained similarly. See VCS documentation and the SystemVerilog LRM for more details

# Implication and Order Constraints

## ■ -> (Implication Operator)

- if ( ... ) ... [ else ... ] also available
- Caution: does not imply solving order 

```
typedef enum { low, mid, high, any } AddrTyp_e;
class MyBus;
    rand bit[7:0] addr;
    rand AddrTyp_e atype;
    constraint addr_range {
        (atype == low) -> addr inside { [0:15] };
        (atype == mid) -> addr inside { [16:127] };
        (atype == high) -> addr inside { [128:255] };
    }
endclass: MyBus
//if (atype == low) addr inside { [0:15] };
//else if (atype == mid) addr inside { [16:127] };
//else if (atype == high) addr inside { [128:255] };
```

7-12

There is NO implied order in the constraint blocks of how the variable values will be solved. In other words the solver is free to choose the variables to randomize first.

Note that if atype == low then addr must be inside the range [0:15]. However this does not imply that if addr is in the range [0:15] then atype must be low. For the constraints in this example if atype == any then addr may be any value in its range.

# Equivalence Constraints

## ■ Use `<->` (Equivalence operator) to define a true bidirectional constraint

- `A <-> B` means if A is true B must be true and if B is true A must be true
- Caution: does not imply solving order 

```
typedef enum { low, mid, high, any } AddrTyp_e;
class MyBus;
    rand bit[7:0] addr;
    rand AddrTyp_e atype;
    constraint addr_range {
        (atype == low ) <-> addr inside { [0:15] };
        (atype == mid ) <-> addr inside { [16:127] };
        (atype == high) <-> addr inside { [128:255] };
    }
endclass: MyBus
```

7-13

Note that if `atype == low` then `addr` must be inside the range `[0:15]`. Conversely, if `addr` is in the range `[0:15]` then `atype` must be `low`. For the constraints in this example, `atype` can never be randomized to `any`.

# Uniqueness Constraints

- Constrain each variable in a group to be unique after randomization

```
class C;
    rand bit [2:0] a[7];
    rand bit [2:0] b;
    constraint cst1 {
        unique { a[0:2], a[6], b };
    }
endclass: C  
  
C c_obj = new;
if (!c_obj.randomize()) $finish;
$display ("a = ", c_obj.a);
$display ("b = ", c_obj.b);
```

Array slices allowed

a = {'h5, 'h0, 'h3, 'h1, 'h7, 'h2, 'h2};  
b = 6

7-14

Only rand variables can be constrained using unique. randc variables cannot be unique.

Starting with vcs2017.12, unique also works on multi-dimensional arrays.

# System Functions

## ■ Bit-vector system functions can be used in constraints (VCS only)

- Treated as an operator/expression instead of a function
  - ◆ \$countbits
  - ◆ \$countones
  - ◆ \$onehot
  - ◆ \$onehot0
  - ◆ \$bits

```
rand bit [3:0] vector;
constraint cst { $countones (vector) == 2; }

//same as
constraint cst {
( vector[0] + vector[1] + vector[2] + vector[3] ) == 2; }
```

Semantic restrictions on function calls  
in constraints do NOT apply here

7-15

The function \$countbits counts the number of bits that have a specific set of values (e.g., 0, 1, X, Z) in a bit vector.

- \$countbits ( expression , control\_bit { , control\_bit } )
- \$countones ( expression ) is equivalent to \$countbits(expression, '1').
- \$onehot ( expression ) returns true if \$countbits(expression, '1) == 1, otherwise it returns false.
- \$onehot0 ( expression ) returns true if \$countbits(expression, '1) <= 1, otherwise it returns false.

The \$bits system function returns the number of bits required to hold an expression as a bit stream. The return type is **integer**

- \$bits ( expression ) | \$bits ( data\_type )

# User-defined Functions in Constraints

- User-defined functions can be used to constrain variables
  - See LRM for rules and limitations on functions and randomization order
  - Can also use C functions using DPI 

```
class D;
    rand bit [6:0] a,b;
    rand bit [7:0] c;
    constraint c0 { c == add(a,b); }

    function bit[7:0] add(input bit[6:0] i1, i2);
        return (i1+i2);
    endfunction
endclass
```

7-16

DPI = Direct Programming Interface. See LRM for details.

Only **rand** variables can be constrained using functions. **randc** variables can not be constrained using functions.

# Real Numbers

## ■ Limited support for randomization of real variables (VCS only)

- Starting with the vcs 2017.12 release you can randomize variables of type
  - ◆ `real`, `shortreal`, `realtime`
- Requires the `-xlrn floating_pnt_constraint` compiler switch
- Only one constraint per real variable allowed
- Only one real variable per constraint allowed
- Can only be constrained by constants or non-random state variables



See Note

```
class cls;
    rand int ix; rand real rx; rand shortreal sx; rand realtime tx;
    constraint t { tx dist {[0.02:0.20]::/30 , [0.1:1.0]::/70}; }
    constraint s { sx dist {[0.04:0.40]::/50 , [0.1:1.0]::/50}; }
    constraint r { rx dist {[0.01:0.10]::/10 , [0.1:1.0]::/90};
                    ix inside {[5:10]}; }
endclass
```

7-17

This feature has limited support. Use with caution.

# Constraint Solver Order

## ■ solve-before construct sets solving order for rand properties

- randc properties are always solved before rand properties
  - ◆ Can not force rand to be randomized before randc properties
- \$void(rand\_property) solves *rand\_property* first (VCS only)

```
class MyBus;
    rand bit flag;
    rand bit[11:0] addr;
    constraint addr_range {
        if ( flag == 0 ) addr == 0;
        else addr inside { [1:1024] };
        solve flag before addr; // guidance only
    }
endclass: MyBus
// solve addr before flag;           // what's the difference?
// solve flag before addr hard;    // force order – VCS only
//     if ( $void(flag) == 0 ) addr == 0; // alternative
```

7-18

VCS only:

```
$void(user_var); //solve user_var before everything else (with a minimum set of constraints)
solve addr before flag hard; //fail if order can not be followed
```

Most simulators may have some similar constructs.

# Inline Constraints

- Individual invocations of `randomize()` can be customized using

```
obj.randomize( ) with { <additional constraints> };

program automatic test;
    class demo;
        rand int x, y, z;
        constraint Limit1 { x > 0; x <= 5; }
        ...
    endclass: demo
    initial begin
        demo obj_a = new();
        //ADD another constraint. Does NOT override Limit1
        if(!obj_a.randomize() with { x > 3 && x < 10; })...
        ...
    end
endprogram: test
```

constraint has scope of class

7-19

Sometimes, the constraint defined within the object being randomized is insufficient to get the desired result. The `with { ... }` construct allows one to solve this problem without the need to go back and modify the original definition of the class. The constraints specified with the `with { ... }` construct are added to the constraints already defined in the original class definition.

For this construct to work effectively, the constraints defined within the original class definition should be as broad as possible within the limits of the DUT spec. And, the properties to be constrained can not be declared as local.

If you want to use variables in the calling scope within the constraint use the keyword `local`. Do not confuse this with a `local` variable inside a `class`.

```
initial begin
    int y = 3;
    demo obj_a = new();
    if(!obj_a.randomize() with { x > local::y //this y is from scope of initial block
                                && x < y ;}) ...; //this y is from class scope
    ...
end
```

# Soft Constraints

## ■ Use keyword **soft** when defining soft constraints

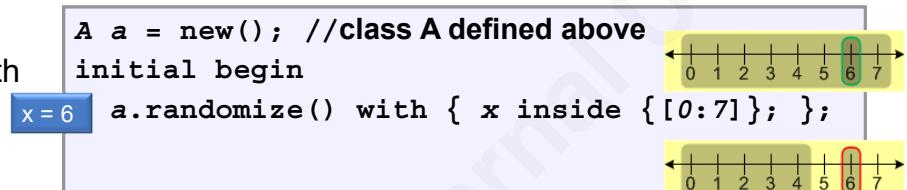
- Only for **rand** variables
- Not for **randc** variables

```
class A;  
  rand bit [7:0] x;  
  constraint A1 { soft x == 6; }  
endclass
```

## ■ Soft constraints are satisfied unless contradicted

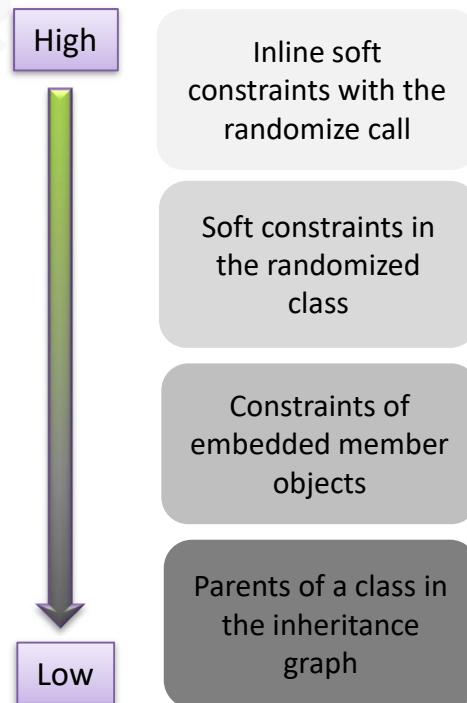
- By a hard constraint
- By a soft constraint with higher priority

```
A a = new(); //class A defined above  
initial begin  
  x = 6; a.randomize() with { x inside {[0:7]}; };  
  
  0 ≤ x ≤ 4 a.randomize() with { x inside {[0:4]}; };  
end
```



7-20

Soft Constraints have the priority shown here.



## Where are Soft Constraints Used?

- In environment classes: to specify default ranges of random variables
- In test program: to bias ranges in test

```
class Packet;
    rand bit [11:0] len; rand int min, max;
    constraint len_c { soft len inside {[min:max]}; }
    constraint range { soft min == 0; soft max == 10; }
endclass
Packet p,q; int tmin, tmax;
// override class constraints with higher priority constraints
stat = p.randomize() with {
    soft len inside {[tmin:tmax]};
}
// or change the object's min and max with hard constraints
stat = q.randomize() with { min==tmin; max==tmax; }
```

7-21

`p.len` gets the values inside the range `tmin:tmax` if the range is completely outside the `min:max` since soft constraints inline have higher priority over soft class constraints. If there is an intersection in the two ranges, only the values within the intersection are valid. For more details on soft constraints refer to the Appendix of this section.

# Mutually Constrained Random Variables

- Constraint limits can be random variables

```
class demo;
  rand bit[7:0] high;
  rand int unsigned x;
  constraint Limit
  {
    x > 1;
    x < high;
  }
endclass
```

- What random values are generated for variable **high**?

- `randomize()` will eliminate values {0,1,2} from possible values for `high`
- If there is no legal value for `high`, then `randomize()` function prints warning and returns a 0. The properties are left unchanged
- Caution: does not imply solving order

7-22

# Inconsistent Constraints

## ■ What if the constraints cannot be solved by `randomize()`?

- It leaves the `object unchanged` and returns a status value of 0
  - ◆ Simulation does not stop
- It produces this simulation error:

Solver failed when solving following set of constraints

```
rand bit[31:0] x; // rand_mode = ON
rand bit[7:0] high; // rand_mode = ON
constraint Limit // (from this) (constraint_mode = ON)
(demo.sv:4)
{
    (x > 1000);
    (x <= high);
}
```

```
class demo;
    rand bit[7:0] high;
    rand int unsigned x;
    constraint Limit {
        x > 1000;
        x <= high;
    }
endclass: demo
```

7-23

When there are conflicting constraints, no variables are changed – they keep their original values.

The simulation switch `+ntb_stop_on_constraint_solver_error=1` will stop VCS simulation when there is an error so you don't have to use the `$finish` system function.

## Effects of Calling randomize()

- When randomize() executes, three events occur:
  - pre\_randomize() is called
  - Variables are randomized
  - post\_randomize() is called
- pre\_randomize() (Optional)
  - Set/Correct constraints
    - ◆ Example: rand\_mode(0|1)
- post\_randomize() (Optional)
  - Make changes after randomization
    - ◆ Example: CRC

```
class Packet;
    int test_mode;
    rand bit[3:0] sa, da;
    rand bit[7:0] payload[];
    bit[15:0] crc;
constraint LimitA {
    sa inside { [0:7] };
    da inside { [0:7] };
    payload.size() inside {[2:4]};
}
function void pre_randomize();
    if(test_mode) sa.rand_mode(0)
endfunction
function void post_randomize();
    gen_crc(); //user method
endfunction
endclass: Packet
```

7-24

rand\_mode() is discussed in a later slide.

# Controlling Randomization at Runtime

- Turn randomization for properties on or off with:

```
task/function int object_name.property.rand_mode ( 0 | 1 );
```

- 1 - enable randomization (default )
- 0 - disable randomization
- If called as function, returns rand\_mode state of property (0 or 1)

```
class Node;
  rand int x, y, z;
  constraint Limit1 {
    x inside {[0:16]};
    y inside {[23:41]};
    z < y; z > x;
  }
endclass: Node
```

```
program automatic test;
initial begin
  Node obj1 = new();
  obj1.x = 0;
  obj1.x.rand_mode(0);
  if(!obj1.randomize()) ...;
end
endprogram: test
```

!! Solver still checks x  
satisfies its constraints !!

7-25

# Controlling Constraints at Runtime

## ■ Turn constraint blocks on and off with:

```
task/function int object_name.constraint_block_name.constraint_mode(0|1);
```

- 1 - enable constraint (default )
- 0 - disable constraint
- If called as function, return state of constraint (0 or 1)

```
program automatic test;
    initial begin
        demo obj_a = new();
        obj_a.no_error.constraint_mode(0); //test with errors
        if(!obj_a.randomize()) ...
    end
endprogram: test
```

```
class demo;
    rand int x, y, z;
    constraint no_error { x > 0; x <= 5; }
    static constraint with_err { x > 0; x <= 32; }
endclass: demo
```

7-26

If a constraint block is declared as **static**, then calls to **constraint\_mode()** shall affect all instances of the specified constraint in all objects. Thus, if a static constraint is set to off, it is off for all instances (objects) of that particular class.

# Constraint Prototypes

- Can define constraint **prototypes** in class using **extern**

- Define the constraint in same scope

```
class demo;
    rand int x, y, z;
    extern constraint valid ; //must define
endclass: demo
//extern constraint must be defined later in same scope as class
constraint demo::valid { x > 0; y >= 0; z % x == 0; }
```

demo.sv

```
program automatic test_corner_case;
`include "demo.sv"
initial begin
    demo obj_a = new();
    if(!obj_a.randomize()) ...;
    end
endprogram: test_corner_case
```

test.sv

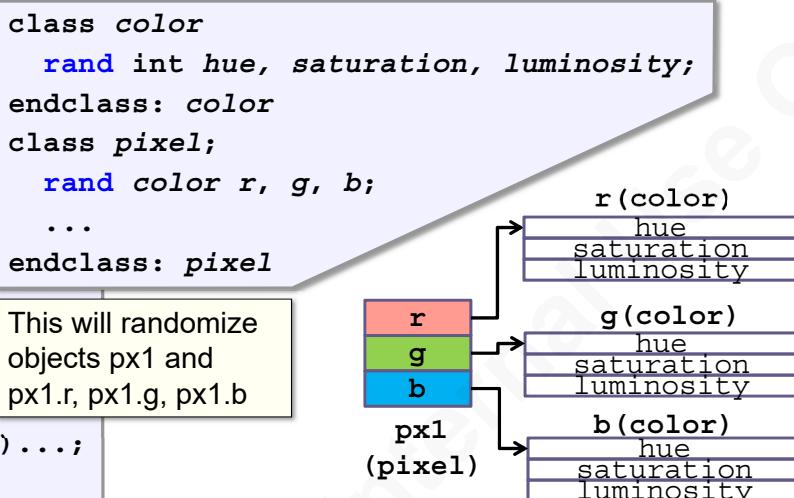
7-27

If an **extern constraint** is undefined, VCS gives a compile-time fatal error.  
**extern constraint** enhances readability just like an **extern method**.

## Nested Objects with Random Variables

- `randomize()` follows a linked list of object handles, randomizing each linked object to the end of the list

```
program automatic test;
initial begin
    pixel px1 = new();
    px1.r = new();
    px1.g = new();
    px1.b = new();
    ...
    if (!px1.randomize()) ...;
end
endprogram: test
```



7-28

If any of the handles `r`, `g`, `b` are null, vcs will produce a CNST-PPRW warning.

This warning can be turned off with compile option:

```
+ntb_disable_cnst_null_object_warning=1
```

## std::randomize()

- **std::randomize() for variables outside classes**
  - Very fast performance in VCS
  - std:: is optional
- **Available in program, module, function, task, and class**
  - Randomization using *obj.randomize()* is still preferred

```
program automatic test;
    bit [11:0] addr;
    bit [5:0] offset;
    function bit genConstrainedAddrOffset();
        return std::randomize(addr, offset) with
            {addr > 1000; addr + offset < 2000;};
    endfunction
endprogram: test
```

Constraints  
using with

7-29

Variables in an object are randomized using the `randomize()` class method. Every class has a built-in `randomize()` virtual function.

However, some less-demanding problems that do not require the full flexibility of classes, can use a simpler mechanism to randomize data that does not belong to a class. The scope randomize method function, `std::randomize()`, enables users to randomize data in the current scope, without the need to define a class or instantiate a class object. The scope randomize method function behaves exactly the same as a class randomize method, except that it operates on the variables of the current scope instead of class member variables. Arguments to this function specify the variables that are to be assigned random values, i.e., the random variables.

You can also break complex random problems into smaller randomizations by doing some randomization using `std::randomize()`. For example you may call it in `pre_randomize()` or `post_randomize()` of a class. This may allow better performance by the constraint solver.

Syntactically outside a class you can drop the `std::` library reference. But it is good practice to use it. If using it inside a class method, you must refer to `std::` library

# Changing the Random Seed at Simulation

- Provide an initial seed for simulator with the following options (VCS)

- `+ntb_random_seed = <initial_seed>`  
`% simv <other_opts> +ntb_random_seed=123`
- `+ntb_random_seed_automatic`
  - ◆ unique initial seed, combining the time of day, hostname and process id
  - `% simv <other_opts> +ntb_random_seed_automatic`

- Seed appears in simulation log and coverage report

- To query for the initial simulation seed use

- `$get_initial_random_seed();`

- To save simulation log messages to a file use

```
% simv <other_opts> -l simv.log
```

7-30

```
program automatic test_corner_case;
    class test_demo extends demo;
        constraint corner_case {
            x == 5; y == 10; z == 20
        }
    endclass: test_demo

    initial begin
        int init_seed;
        init_seed = $get_initial_random_seed();
        $display("Simulation started with seed = %0d", init_seed)
        test_demo obj_a = new();
        if(!obj_a.randomize())
            ...
    end
endprogram: test_corner_case
```

## Quiz Time ?

7-31

# Randomization: Quiz 1

```
program automatic test1 ;
class abc ;
    rand int a ;
    constraint c1 {a inside {[1:4]} ;}
    constraint c2 {a inside {[5:8]} ;}
endclass
initial begin
    abc o1 = new() ;
    o1.randomize() ;
    $display("test: o1 = %p", o1) ;
end
endprogram: test1
```

1. Will this code compile without errors?
2. Will it throw any runtime errors? Why?
3. What will the program display?

7-32

3. test: o1 = {a:0}

```
=====
constraint c1 // (from this) (constraint_mode = ON) (test1.sv:5)
{
    (a inside {[1:4]}) ;
}
constraint c2 // (from this) (constraint_mode = ON) (test1.sv:6)
{
    (a inside {[5:8]}) ;
}
```

Solver failed when solving following set of constraints

1. Yes
2. Yes. The constraints c1 and c2 are not solvable together. All constraints must be satisfied.

## Randomization: Quiz 2

```
program automatic test2 ;
  class abc ;
    rand int a ;
    constraint c1 {a inside {[1:4]} ;}
  endclass

  initial begin
    abc o2 = new() ;
    o2.randomize() with {a inside {[5:8]} ;} ;
    $display("test1: o2 = %p", o2) ;
  end
endprogram: test2
```

1. Will this code compile without errors?
2. Will it throw any runtime errors? Why?
3. What will the program display?

7-33

3. test: o1 = {a:0}

```
=====
constraint c1 // (from this) (constraint_mode = ON) (test1.sv:5)
{
  (a inside {[1:4]}) ;
}

constraint c2 // (from this) (constraint_mode = ON) (test1.sv:6)
{
  (a inside {[5:8]}) ;
}
```

Solver failed when solving following set of constraints

1. Yes
2. Yes. The constraints c1 and in the with clause are not solvable together. All constraints must be satisfied.

## Randomization: Quiz 3

```
program automatic test3 ;
  class abc ;
    rand int a ;
    constraint c1 {soft a inside {[1:4]} ;}
  endclass

  initial begin
    abc o3 = new() ;
    o3.randomize() with {a inside {[5:8]} ;}
    $display("test3: o3 = %p", o3) ;
  end
endprogram: test3
```

1. Will this code compile without errors?
2. Will it throw any runtime errors?
3. What will the program display?

7- 34

3. test3: o3 = {a:5} (a will have value between 5 and 8 as picked by constraint solver
1. Yes  
2. No

## Randomization: Quiz 4

```
program automatic test4 ;  
  
class abc ;  
    int a ;  
    constraint c {a inside {[0:4]} ;}  
endclass: abc  
  
initial begin  
  
    abc o4 = new() ;  
    o4.a = 6 ;  
    o4.randomize() ;  
    $display("test4: o4 = %p", o4) ;  
  
end  
  
endprogram: test4
```

1. Will this code compile without errors?
2. Will it throw any runtime errors? Why?
3. What will the program display?

7-35

3. test4: o4 = {a:6}

```
=====  
constraint c // (from this) (constraint_mode = ON) (test4.sv:5)  
{  
    a inside {[0:4]};  
}  
integer a = 6;
```

Solver failed when solving following set of constraints

- ```
=====  
1. Yes  
2. Yes. Property a is not declared as a rand or randc property, but the solver still checks that the constraints are satisfied.
```

## Randomization: Quiz 5

```
program automatic test5 ;  
  
class abc ;  
    rand int a ;  
    constraint c {a inside {[0:4]} ;}  
endclass: abc  
  
initial begin  
  
    abc o5 = new() ;  
    o5.a = 6 ;  
    o5.randomize() ;  
    $display("test5: o5 = %p", o5) ;  
  
end  
  
endprogram: test5
```

- 1. Will this code compile without errors?
- 2. Will it throw any runtime errors?
- 3. What will the program display?

7- 36

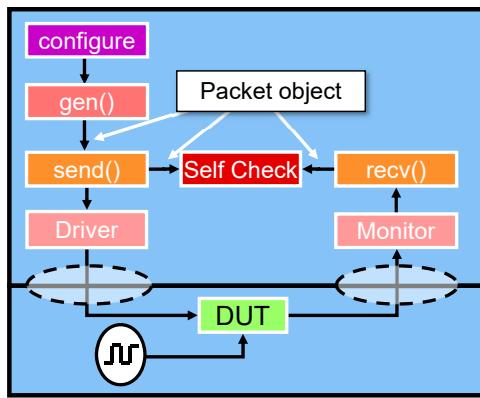
3. test5: o5 = {a:3} (a will have value between 0 and 4 as picked by constraint solver)
2. No
1. Yes

# Lab 4 Introduction



60 minutes

## Encapsulate Data in Packet Class



Encapsulate Data  
in Packet Class

Compile & Simulate

7-37

## Unit Objectives Review

**Having completed this unit, you should be able to:**

- Explain why randomization is needed in verification
- Randomize variables
- Constrain randomization of variables



7-38

## **Appendix**

**struct Randomization**

**Soft Constraints: Rules and Management**

**Random Stability**

**Constraint Debug and Profiling**

**Methodology and Best Practices**

**7-39**

## **struct Randomization**

**7-40**

## struct Randomization Example (1/2)

### ■ struct inside class

```
typedef struct {
    rand int x;
    int y;
} ST0;
typedef struct packed {
    int x;
    int y;
} ST1;
class C;
    rand ST0 st0;
    rand ST1 st1;
constraint cst0 { st0.x == 10; }
constraint cst1 { st0.x == st1.x; }
endclass
```

Not rand

x and y  
randomized  
because packed

Output:  
st0: x:10, y:0  
st1: x:10, y:8

```
program automatic test;
    C obj = new;
    initial begin
        obj.randomize;
        $display("%p", obj);
    end
endprogram
```

7-41

## struct Randomization Example (2/2)

### ■ Object inside struct

```
class C;  
    rand int x;  
    constraint cst { x == 123; }  
endclass  
  
typedef struct {  
    rand int y;  
    rand C c0;  
} ST1;
```

Output:  
y:123, c0:{ ref to class C}  
x:123

```
program automatic test;  
initial begin  
    ST1 s;  
    s.c0 = new;  
    std::randomize(s) with {  
        s.c0.x == s.y;  
    };  
    $display("%p, %p", s, s.c0);  
end  
endprogram
```

Auto-allocates *s.y*  
*c0* must be  
constructed

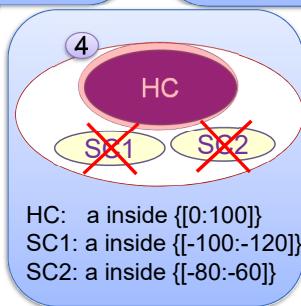
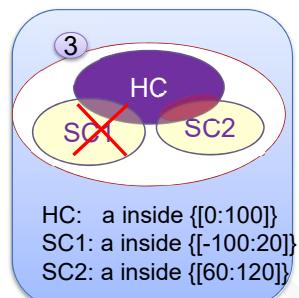
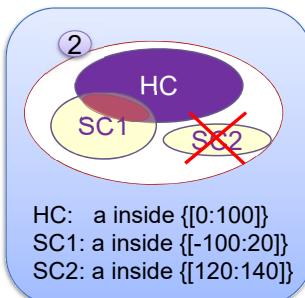
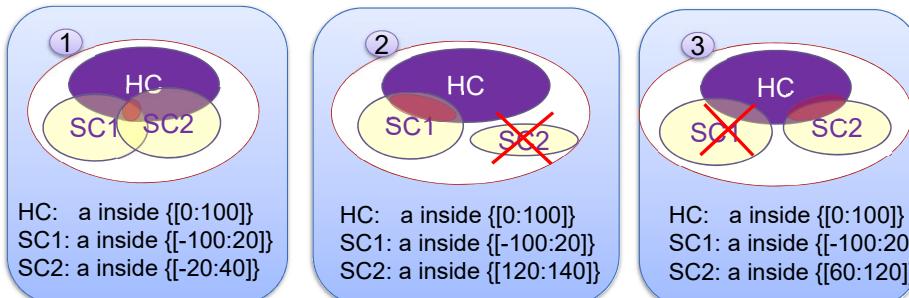
7-42

## **Soft Constraints: Rules and Management**

**7-43**

# How do soft constraints work?

Only **Contradictions** can lower solving priorities



HC Hard constraint  
SC1 Soft constraint #1  
SC2 Soft constraint #2

Higher priority

7-44

## Priority of Soft Constraints (1/5)

High  
↓  
Low

- 1 Inline soft constraints with the randomize call
- 2 Soft constraints in the randomized class
- Constraints of embedded member objects
- Parents of a class in the inheritance graph

```
class A;  
    rand bit [2:0] n;  
    constraint A1 {  
        soft n inside {[1:3]}; ②  
    }  
endclass  
program automatic test;  
    A a = new();  
    initial  
        a.randomize() with {  
            soft n > 4; ①  
        };  
    endprogram
```



Note: The last declared constraint within the same scope has higher priority

7-45

The inline constraints added to the `randomize()` call have a higher priority than the constraints inside the class definition. So `n` will be constrained to the range `[5:7]`.

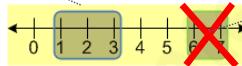
## Priority of Soft Constraints (2/5)

High  
↓  
Low

- Inline soft constraints with the randomize call
- Soft constraints in the randomized class
- Constraints of embedded member objects
- Parents of a class in the inheritance graph

① ②

```
class A;  
  rand bit [2:0] x;  
  constraint A1 {  
    ② soft x > 5;  
    ① soft x inside {[1:3]};  
  }  
endclass  
program automatic test;  
  A a = new();  
  initial  
    a.randomize();  
endprogram
```



Note: The last declared constraint within the same scope has higher priority

7-46

## Priority of Soft Constraints (3/5)

High  
↓  
Low

- 1 Inline soft constraints with the randomize call
- 2 Soft constraints in the randomized class
- 3 Constraints of embedded member objects
- 4 Parents of a class in the inheritance graph

```

class A;
    rand bit [7:0] n;
    ③ constraint A1 { soft n inside {[5:9]}; }
endclass

class B;
    rand bit [7:0] m;
    ② constraint B1 { soft m inside {[0:4]}; }
endclass

class C;
    rand A objA = new(); rand B objB = new();
    ① constraint C1 { soft objA.n == objB.m; }
    ...
endclass

program automatic test;
    C objC = new();
    initial objC.randomize();
endprogram

```

Note: The last declared constraint within the same scope has higher priority

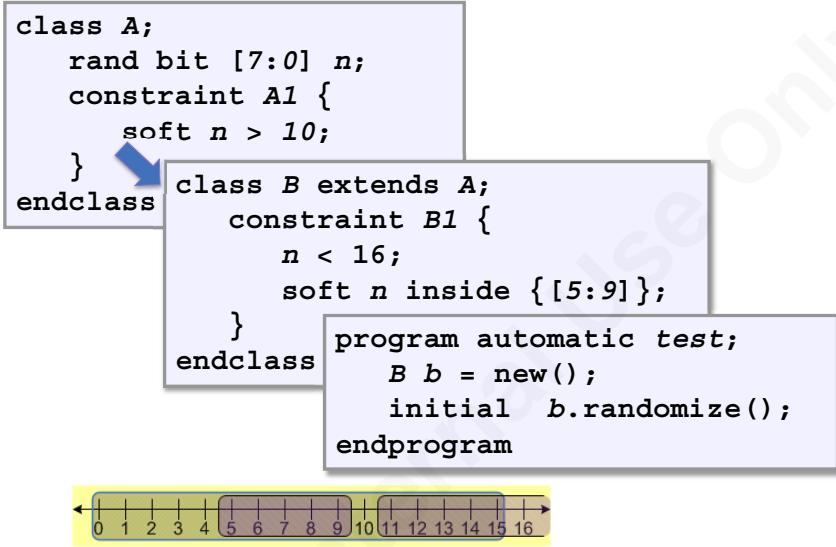
7-47

The constraint **C1** in class **C** sets **objA.n == objB.m**. Since this constraint is inside the class **C** it takes priority over the soft constraints inside the member objects **objA** and **objB**. There being no other constraints on **objB.m**, the values of **objB.m** are unconstrained within **C**, but constrained to [0:4] when **objB** is randomized due to the constraint **B1**. This limits **objA.n** to the same range as **objB.m** due to the constraint **C1**. This overrides the soft constraint **A1** in class **A** for **objA.n**.

## Priority of Soft Constraints (4/5)

High  
↓  
Low

- Inline soft constraints with the randomize call
- Soft constraints in the randomized class
- Constraints of embedded member objects
- Parents of a class in the inheritance graph



Note: The last declared constraint within the same scope has higher priority

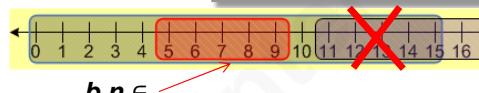
7-48

## Priority of Soft Constraints (5/5)

High  
↓  
Low

- 1 Inline soft constraints with the randomize call
- 2 Soft constraints in the randomized class
- 3 Constraints of embedded member objects
- 4 Parents of a class in the inheritance graph

```
class A;  
    rand bit [7:0] n;  
    constraint A1 {  
        soft n > 10;  
    }  
endclass  
  
class B extends A;  
    constraint B1 {  
        1 n < 16;  
        soft n inside {[5:9]};  
    }  
endclass  
  
program automatic test;  
    B objB = new();  
    initial objB.randomize();  
endprogram
```



b.n ∈

Note: The last declared constraint within the same scope has higher priority

7-49

Since class **B** inherits from **A** it's soft constraints override conflicting soft constraints of parent class **A**. The constraint **soft n inside {[5:9]}** in class **B** does not conflict with the constraint **n < 16**, so it will be the final constraint for **n**.

## Disabling Soft Constraints (1/3)

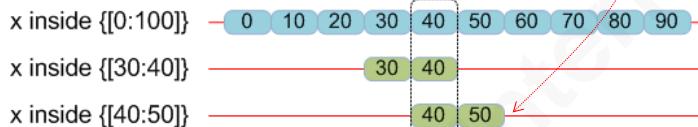
- Soft constraints can sometimes narrow down the range of a random variable even more than needed

```
class my_class;  
  rand bit [7:0] x;  
  constraint x_constr {  
    x inside {[0:100]};  
    soft x inside {[30:40]};  
  }  
endclass: my_class
```

But I do NOT want to keep  
this soft constraint!!!

```
my_class obj = new();  
initial begin  
  obj.randomize with {  
    soft x inside {[40:50]};  
  };  
end
```

Test



7-50

## Disabling Soft Constraints (2/3)

- Use `disable soft` to disable soft constraints

```
class my_class;
  rand bit [7:0] x;
  constraint x_constr {
    x inside {[0:100]};
    soft x inside {[30:40]};
  }
endclass: my_class
```

my\_class obj = new();  
initial begin  
 obj.randomize with {  
 disable soft x;  
 soft x inside {[40:50]};  
 };  
end

But I do NOT want to keep  
this soft constraint!!!



7-51

## Disabling Soft Constraints (3/3)

```
class sc_class;
    rand bit [7:0] x, y;
    constraint c1 {
        soft x == 10;
    }
    extern constraint c2;
endclass

constraint sc_class::c2 {
    soft x + y == 15;
}
```

```
program automatic t;
    sc_class o = new();
    initial begin
        o.randomize() with {
            disable soft x;
        };
        ...
    end
endprogram
```



x and y become unconstrained

- A **disable soft** constraint on a random variable specifies that all lower priority soft constraints that reference that random variable shall be dropped.

7-52

# Soft Constraint Debug using Verdi

The screenshot shows three code snippets on the left and a constraint debugger interface on the right.

**Code Snippets:**

- Class A:**

```
class A;
  rand int x;
  constraint c1 { soft x == 2; }
endclass
```
- Class B:**

```
class B;
  rand int x;
  constraint c2 {
    soft x == 5;
    soft x == 3;
  }
endclass
```
- Class C:**

```
class C;
  rand int x;
  rand A objA;
  rand B objB;
  constraint c3 {
    soft x == objA.x;
    soft objA.x < objB.x;
    soft x == 4;
  }
endclass
```
- Program automatic test:**

```
program automatic test;
  initial begin
    C objC = new;
    objC.objA = new;
    objC.objB = new;
    objC.randomize;
  end
endprogram
```

**Constraint Debugger Interface:**

The interface has tabs for "Split" and "Origin". The "Solver" tab is selected. The "Relation" tab is also visible.

Name	Value
Randomize Call:1	01
Partition:1	
Variables	
x	4
objA.x	4
objB.x	5
c3	
soft (x == objA.x)	
soft (objA.x < objB.x)	
soft (x == 4)	
objA.c1	
soft (objA.x == 2)	
objB.c2	
soft (objB.x == 5)	
soft (objB.x == 3)	

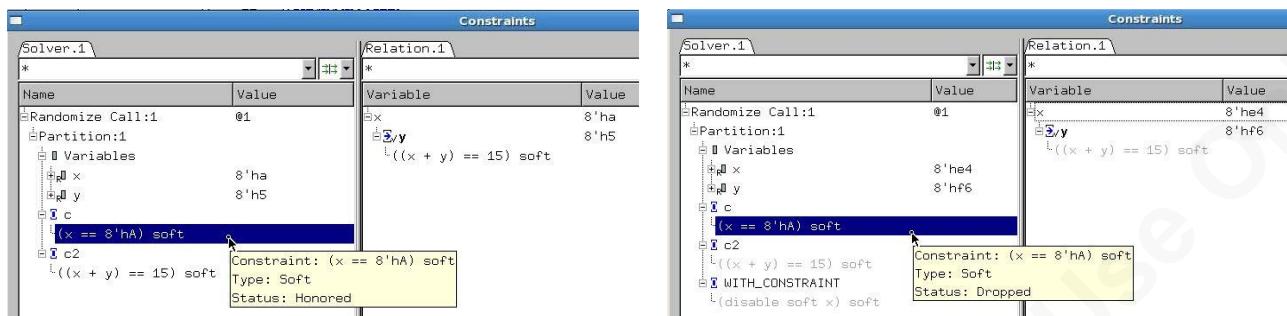
A tooltip is shown for the constraint "soft (objA.x == 2)":  
Constraint: soft (objA.x == 2)  
Type: Soft  
Status: Dropped

## ■ Use Constraint debugger to debug constraints

- Was a soft constraint dropped or honored?
- Explore relations between the variables
- List all constraints related to some variable
- Find insufficiencies in the constraints
- View initial (all possible values) list of variables

7-53

# Debug Soft Constraints using DVE



## ■ Constraints debugger allows you to debug soft and hard constraints

- Was a soft constraint dropped or honored?
- Explore relations between the variables
- List all constraints related to some variable
- Find insufficiencies in the constraints
- View initial (all possible values) list of variables

7 - 54

## **Random Stability**

**7-55**

# Random Stability: Threads and Objects

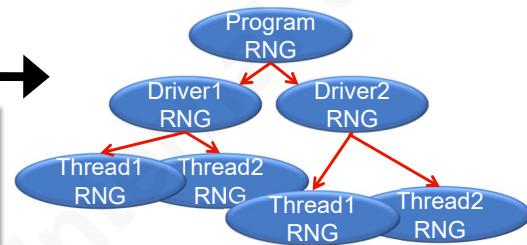
## ■ SystemVerilog

- All modules start with the same seed
  - ◆ Modules not recommended for tests – use program
- Random Number Generator (RNG) per object, thread
- RNG are hierarchically seeded from program root
  - ◆ Most small code changes give same runtime results
  - ◆ Use srandom(seed) to seed each object or thread if needed

```
task driver::main();
  fork
    thread1();
    thread2();
  join_none
endtask: main
```

```
program automatic test;
  driver drv1=new();
  driver drv2=new();
  ...

```



7-56

Random stability refers to a program, such as a testbench, producing the same randomization results for executions on the same software (simulator). As much as possible, a program should remain stable when small changes are made.

SystemVerilog offers a feature called random stability. This feature ensures that the results from one simulation to another are consistent. Testbenches with this feature exhibit more stable RNG behavior in the face of small changes to the user code.

First, each thread and object in SystemVerilog TB contains a unique random number generator, ie. each object maintains its own internal RNG, which is used exclusively by its randomize() method. This allows objects to be randomized independent of each other and of calls to other system randomization functions.

When an object is created, its RNG is seeded using the next value from the RNG of the thread that creates the object. This process is called hierarchical object seeding.

Random stability encompasses the following properties:

- Initialization RNG. Each module, interface and program instance, has an initialization RNG.
- Thread stability. Each thread has an independent RNG for all randomization.
- Object stability. Each class instance (object) has an independent RNG for all randomization methods in the class. When an object is created using new, its RNG is seeded with the next random value from the thread that creates the object.
- Each thread or object has a srandom() function to allow controlled seeding of the thread or object.

## **Constraint Debug and Profiling**

**Interactive Constraint Debug Using Verdi**

**Interactive Constraint Debug Using DVE**

**Constraint Profiling**

**7-57**

# Constraint Debug Methods

- VCS automatically reports/extracts the minimum sets of constraints that are causing inconsistencies
- Constraints satisfied ... but wrong result?
  - % simv +ntb\_solver\_debug=serial
  - % simv +ntb\_solver\_debug=extract+trace  
    +ntb\_solver\_debug\_filter=<serial#>
- Constraints satisfied ... but too slow!
  - % simv +ntb\_solver\_debug=profile
  - % firefox simv.cst/html/profile.xml
  - % simv +ntb\_solver\_debug=extract  
    +ntb\_solver\_debug\_filter=<serial#>. <partition#>
  - simv.cst/testcases contains extracted testcases

7-58

The runtime option `+ntb_solver_debug` provides you with many constraint debug features  
`+ntb_solver_debug=serial`

The serial number assignment to the randomizations in a simulation provides a method to identify the `randomize()` calls to be debugged next. Once identified, you can use this runtime option with appropriate arguments to report the trace and extract test cases. The constraint profiler also uses the same identification method to provide feedback to you.

`+ntb_solver_debug=trace`

This enables solver trace reporting for the specified `randomize()` calls. This helps the user to understand how VCS solves the random variables for given `randomize()` calls. The `+ntb_solver_debug_filter` option is required to specify a list of `randomize()` calls for which to enable the solver trace.

`+ntb_solver_debug=profile`

This enables constraint profiling for the simulation at runtime. The profile report provides important information to you which `randomize()` calls should be targeted for improving constraint performance to bring down the total simulation run time or memory.

`+ntb_solver_debug=extract`

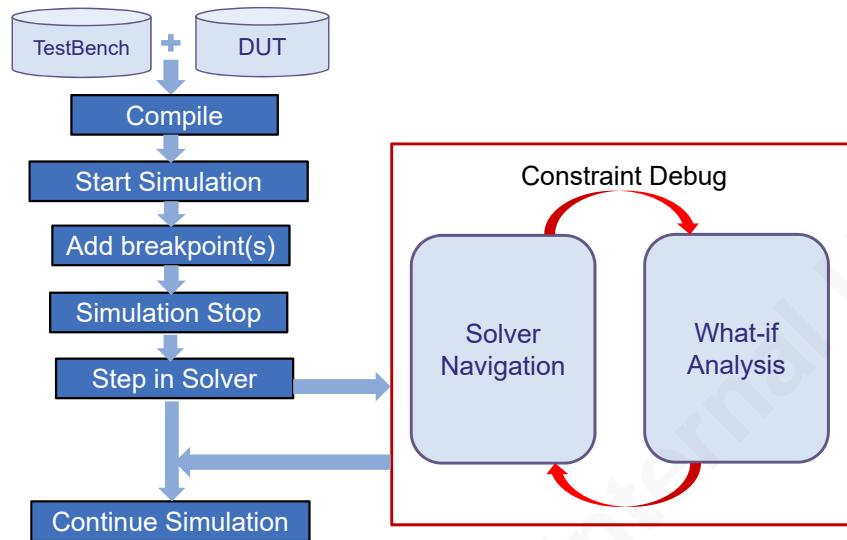
This enables test case extraction for the specified `randomize()` calls. This creates standalone test cases for you to compile and run outside of the original design. This should help quicker turnaround time to experiment possible fixes as it is faster to compile and run a smaller test case.

The `+ntb_solver_debug_filter` option is required to specify a list of `randomize()` calls for which to enable test case extraction.

## Interactive Constraint Debug Using Verdi

7 - 59

# Constraint Debug & Analysis Flow Using Verdi

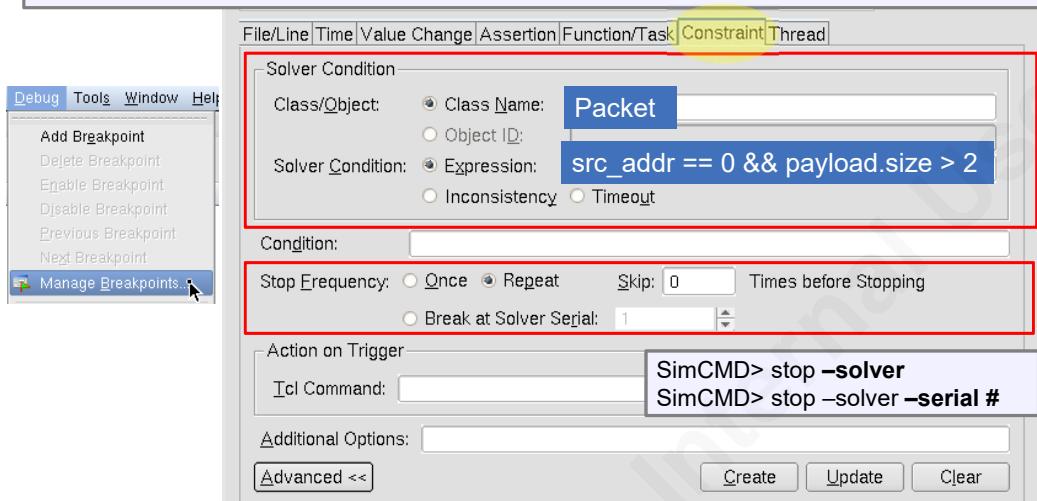


7-60

# Flexible Breakpoint Infrastructure

## ■ One-pass debug starts here!

```
SimCMD> stop -solver -class Packet -solver_cond { src_addr == 0 && payload.size > 2 }
SimCMD> stop -solver -class * -inconsistency
```



7-61

Do we always know where the randomize statements are in the testbench? Once upon a time, perhaps we did as we wrote those generators and we owned the entire block level verification. Now, with the adoption of standardized methodology base classes like UVM, with a lot of reuse of other testbench components and increasing verification IP usage, we do not always know.

What we do know is that when we are debugging something, we usually see something suspicious and we want to be able to describe that symptom to the debugger so that we can catch that situation as it gets generated. For example, we see a packet that's received by the DUT as it has a src address of 0 and payload size greater than 2, and we want to be able to describe that to the constraint debugger.

Guess what? We can.

This breakpoint dialog has a constraints tab and that allows you to specify symptoms similar to what was just described.

We can enter the name of the packet class, and write out the solver condition (`src_addr==0` and `payload.size > 2`) in the expression and create that breakpoint. The simulation will run until this solver condition is met and will then give you full control to debug that specific randomization.

Constraint inconsistency, for example, is another such solver condition. I can use a wildcard here select Inconsistency and then any time any randomize fails due to constraint inconsistency, the simulation will stop and give me control to debug further.

The use of the solver serial id is also useful. For example, if in your regression runs, some randomization fails. The error message will tell you which serial ID to use if you wish to debug further. Then you can enter it here and run your simulation till that point and debug from there.

# Debugging Inconsistent Constraints Using Verdi

## ■ Focusing on a minimal set of conflicting constraints

The screenshot shows two windows from the Verdi tool. The left window is a 'Warning' dialog box with the title '1> Constraints inconsistency failure' and the message 'diag.sv, 19 Constraints are inconsistent and cannot be solved. Check the inconsistent constraints being printed above and rewrite them.' The right window is a 'Solver Relation' browser. It displays a tree structure of constraints under a 'Randomize Call:1' node. A red box highlights a constraint '(a == (b - 3))'. A blue arrow points from the warning dialog to the solver browser. The browser also shows other nodes like 'Partition:1', 'Function Calls', 'Partition:2', and 'Inconsistent' nodes. A yellow callout box points to a variable 'fv\_4 /\*this.Q::add(a, b)\*/' with the text 'Variable: fv\_4 /\*this.Q::add(a, b)\*/ Type: bit[7:0] Random: State'. The bottom right corner of the slide has the text '7-62'.

7-62

In the case of constraint inconsistency, instead of giving you all constraints that exist in this randomize call, the tool will quickly do some automatic minimization of constraints causing conflict.

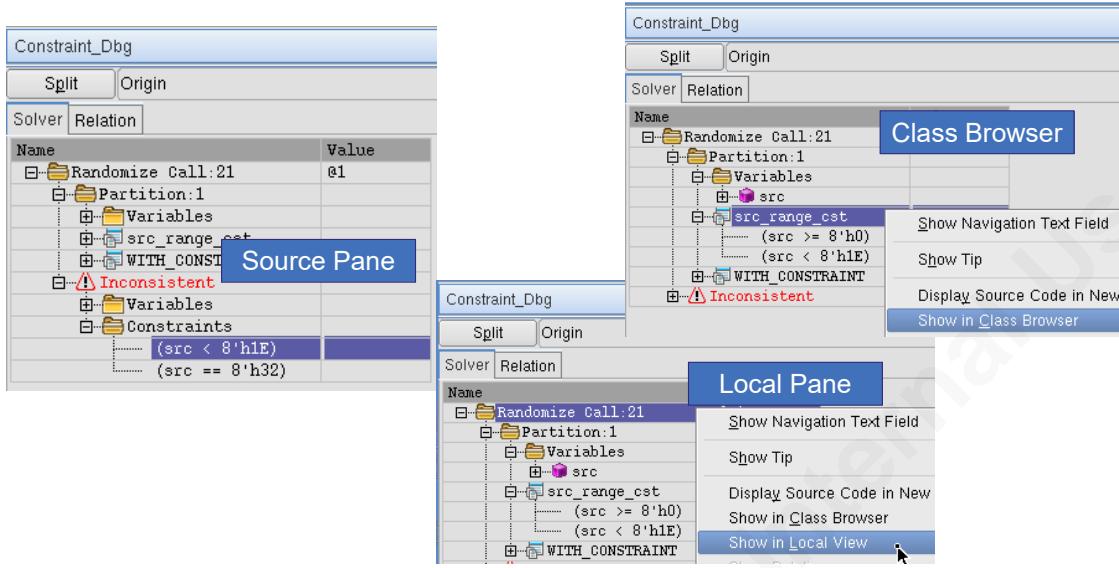
Remember debug is often about finding the most relevant thing to root cause the problem.

When a randomize call has no solution, somewhere in the sea of constraints, some things are contradicting with some others. This automatic minimization of constraints causing conflict can really quickly pin point the source of the problem. This will save time.

In this screen shot, you see the inconsistency is caused by the presence of a state variable, that is a return value of the function. You can scroll up to see how the function gets evaluated and what its arguments are and how the arguments are solved before the function evaluation.

# Cross Probing Using Verdi

## ■ Easy and accurate code navigation



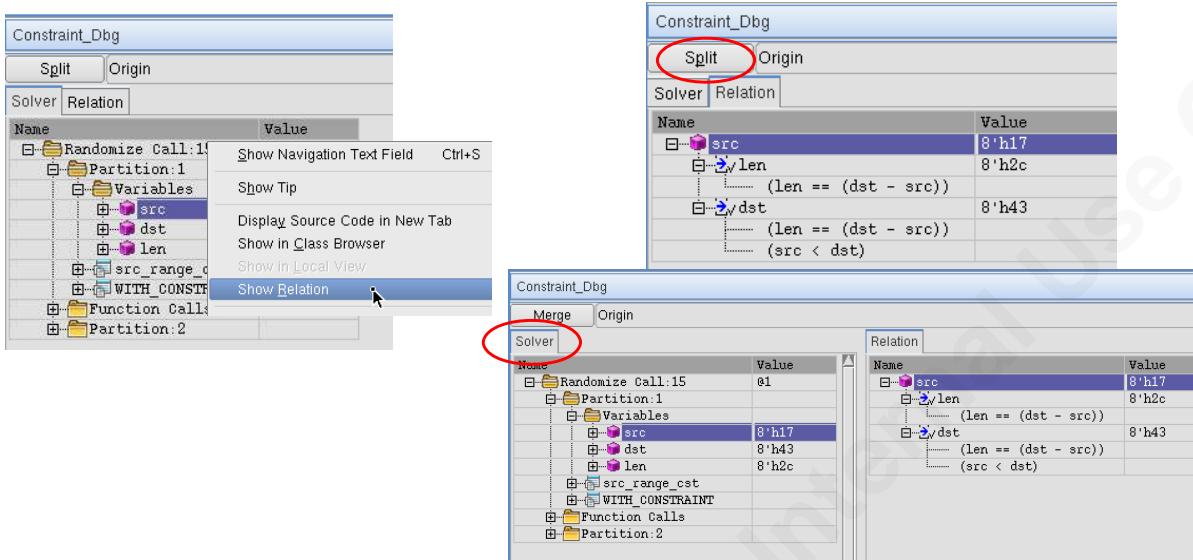
7-63

You'll often want to cross reference to other aspects of the debug views to see to look at the source code, to examine the class hierarchy and members, and to inspect the dynamic random variables and constraint blocks and their properties. This constraint debugger will take you to it quickly and accurately.

It simplifies code navigation – especially important if generally the constraint code includes ones that were written by someone else from other projects or from the verification IPs

# Relation Space Analysis Using Verdi

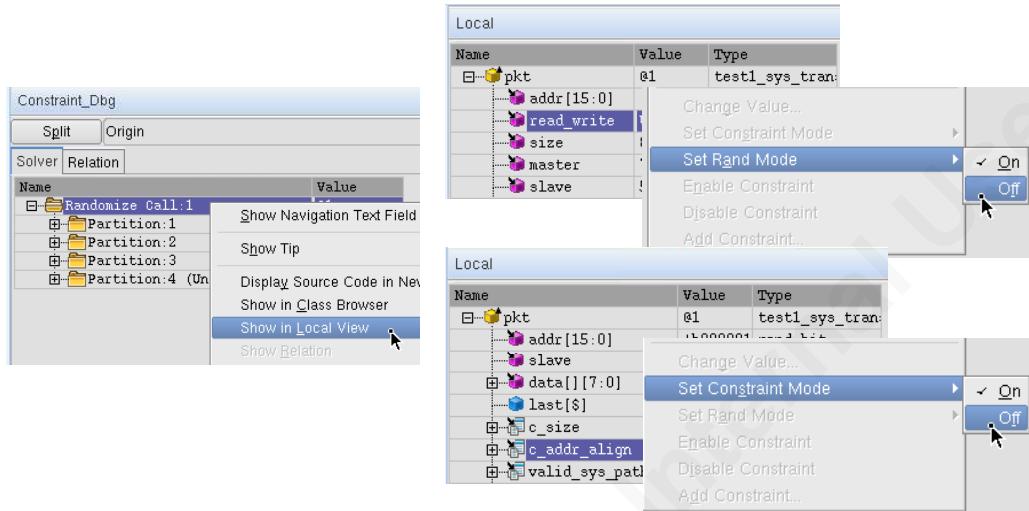
## ■ Better visualization of the constraint network



7-64

# Interactive Constraint Editing Using Verdi

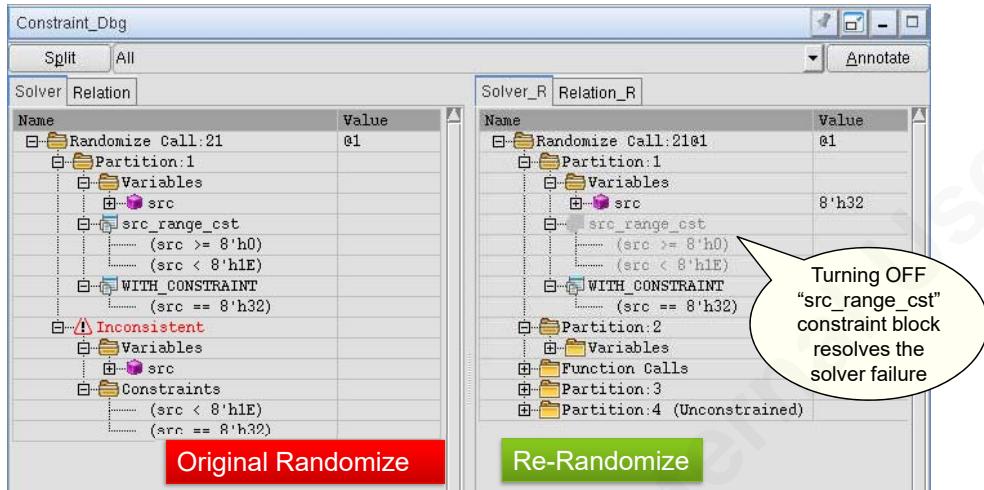
- Enabling what-if analysis without recompile
- Changing `rand_mode`, `constraint_mode`



7-65

# On-the-fly Re-randomization Using Verdi

- Immediate cause-effect analysis without restarting simulation

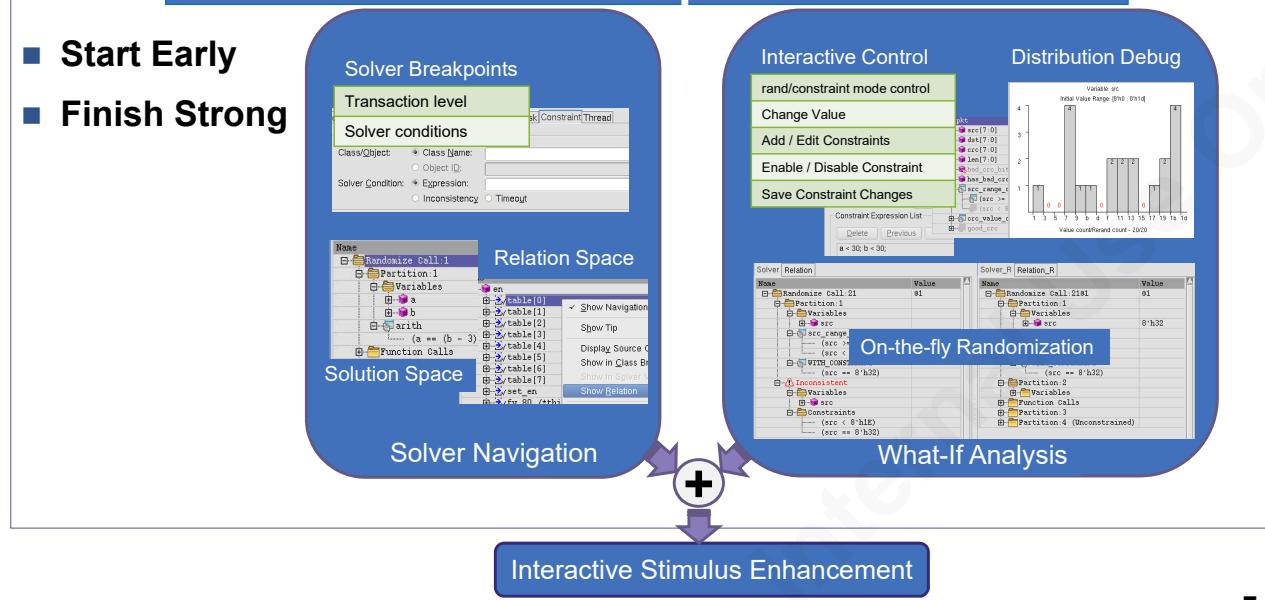


7-66

# Constraint Debug and Analysis Using Verdi

## Constraint Development Platform

- Start Early
- Finish Strong



7-67

## Interactive Constraint Debug Using DVE

7 - 68

# Debug - Constraint Conflicts

## ■ VCS Simplifies Constraint Debug

- Incorrect constraints can result in no legal solution
- Entire problem may be very large, 1000+ constraints
- VCS Identifies conflicting variables and constraints
- Problem can be analyzed immediately

```
Solver failed when solving following set of constraints

rand bit[7:0] a; // rand_mode = ON
rand bit[7:0] b; // rand_mode = ON

constraint cst1 { a > 5; }                      // test.sv:19
constraint cst2 { b > 10; }                       // test.sv:20
constraint cst3 { a + b == 12 ; }                  // test.sv:21
```

Conflicting  
Variables  
Subset

Conflicting  
Constraints  
Subset

7-69

The VCS constraint solver partitions the constraints

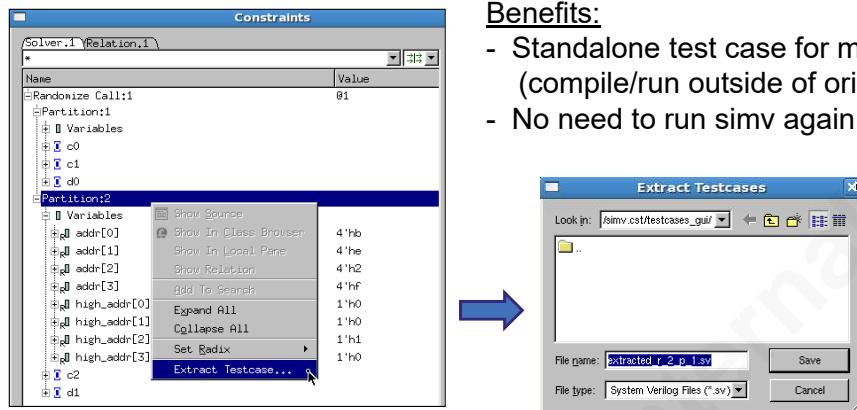
- Simpler to solve many small pieces than one large piece
- Unrelated variables will be placed into separate partitions
- Related variables solved in prior partitions become state variables

There are many constructs in constraints that may force separate partitions. For example

- solve before hard
- function calls
- \$void
- foreach loops

# Interactive Constraint Debug Using DVE (1/5)

- Standalone constraint test case for a given partition inside the randomize call can be extracted from constraint dialog



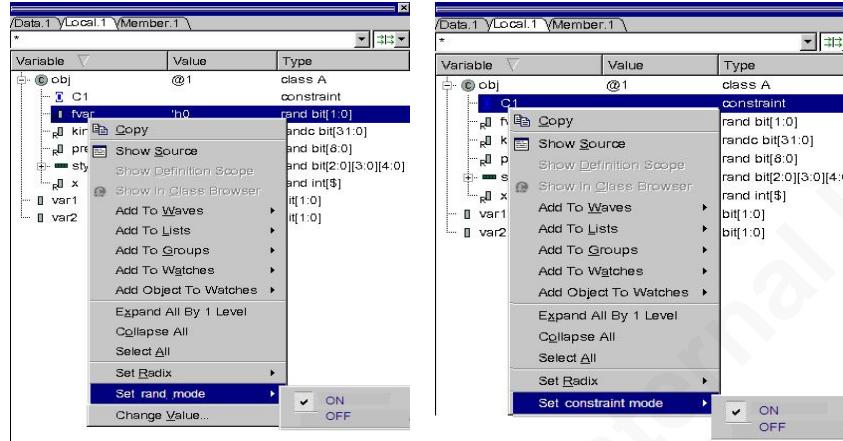
## Benefits:

- Standalone test case for more debug (compile/run outside of original design)
- No need to run simv again

7-70

## Interactive Constraint Debug Using DVE (2/5)

- Modification of `constraint_mode` and `rand_mode` from within the local pane

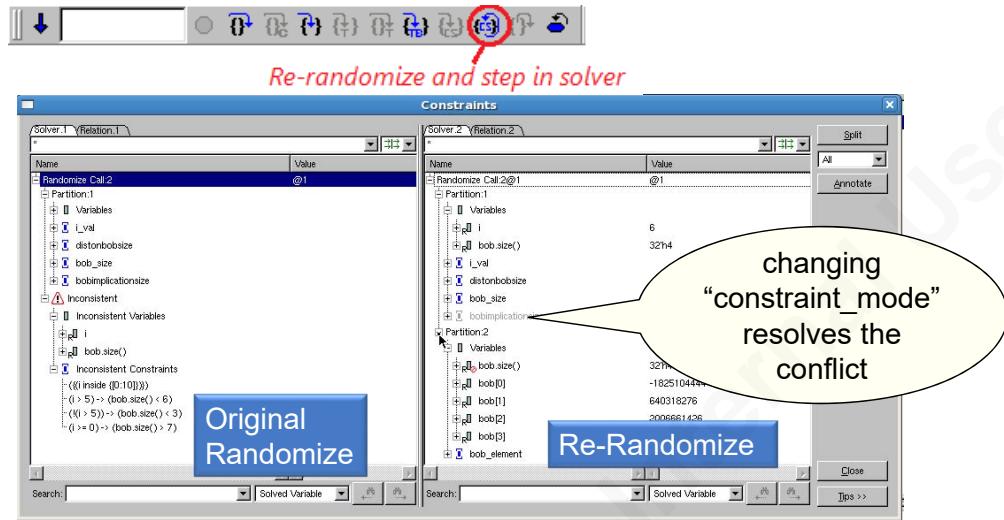


Benefits: User can explore and validate possible solutions without leaving DVE

## Interactive Constraint Debug Using DVE (3/5)

### ■ Randomize again while stopped inside the solver

- after changing the states of involved variables and constraint blocks, etc.



7-72

## Interactive Constraint Debug Using DVE (4/5)

- Highlighting the different solved results from before

The screenshot shows the DVE interface with two constraint solvers, Solver 1 and Solver 2.

**Solver 1 / Relation 1:**

Name	Value
Randomize Call:1	@1
Partition:1	
Variables	
+ R[i]	8
+ R[bob.size()]	32'h2
+ I[i_val]	
+ I[distonbobsize]	
+ I[bob_size]	
+ I[bobimplicationsize]	
Partition:2	
Variables	
+ R[bob.size()]	32'h2
+ R[bob[0]]	-1473512013
+ R[bob[1]]	-570674680
+ I[bob_element]	

**Original Randomize**

**Solver 2 / Relation 2:**

Name	Value	Original Value
Randomize Call:1@1	@1	
Partition:1		
Variables		
+ R[i]	8	8
+ I[i_val]		
+ I[distonbobsize]		
+ I[bob_size]		
+ I[bobimplicationsize]		
Partition:2		
Variables		
+ R[i]	8	32'h3
+ R[bob.size()]	32'h3	32'h2
+ I[distonbobsize]		
+ I[bob_size]		
+ I[bobimplicationsize]		
Partition:3		
Variables		
+ R[bob.size()]	32'h3	32'h3
+ R[bob[0]]	314890130	-1473512013
+ R[bob[1]]	-1912847359	-570674680
+ R[bob[2]]	857647740	
+ I[bob_element]		

**Annotate** (highlighted with a red circle)

**Re-Randomize**

7-73

## Interactive Constraint Debug Using DVE (5/5)

- Unconstrained randomization also shown in the constraint dialog

The screenshot shows the DVE interface with two main windows. On the left is a code editor window displaying Verilog-like code:

```
1 class A;
2   rand bit [7:0] x;
3   rand bit [7:0] y;
4   rand bit [7:0] z;
5   constraint d { x + y == 153; }
6 endclass
7
8 program automatic test;
9   A obj = new;
10  @1
11  initial
12    obj.randomize();
```

A blue callout box points to the variable 'z' in line 12 with the text "z is unconstrained". On the right is a "Constraints" dialog window titled "Solver.1 \ Relation.1". It shows a table of constraints:

Name	Value
Randomize Call:1	@1
Partition:1	
Variables	
+R x	8'h6d
+R y	8'h2c
+R d	
((x + y) == 153)	
Partition:2 (Unconstrained)	
Variables	
+R z	8'h9b

7-74

## Constraint Profiling

7-75

# Performance: Solver Diagnostics

## ■ Constraint Solver Diagnostic Report

- Constraint Expression Static Analysis
- Random Variable Dynamic Analysis
- Some constraints are mathematically hard to solve
- Focus attention on constraints causing slowdown

```
% ./simv +ntb_enable_solver_diagnostics=<N>
      +ntb_solver_diagnostics_filename=<filename>
```

# Constraint Profile Using Simprofile (1/2)

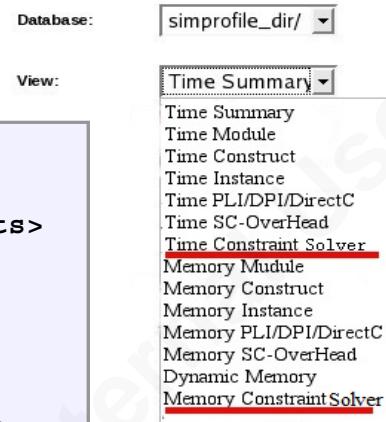
## ■ Use Model

- Starting with VCS 2012.09 constraint profiler has been integrated with vcs simprofile

```
% vcs -simprofile <vcs_opts>

% simv -simprofile [ mem | time ] <sim_opts>

% profrpt -view [time_all |
                  mem_all |
                  time_solver |
                  mem_solver ] <profile_db>
```



## Constraint Profile Using Simprofile (2/2)

**Simprofile Report**

Database: simprofile\_dir

View: Time Constraint

**Time Constraint Solver View**

Total user time: 12.180seconds  
Total system time: 0.380seconds  
Total randomize time: 0.030seconds  
Total randomize count: 2

Fileline@visit	serial#	time (sec)	variables	constraints	cnst blocks
/env/mvs_atapi_env_sv118@1	1	0.030	27	37	9
/env/mvs_atapi_env_sv120@1	2	0.000	3	3	1

**Memory Constraint Solver View**

Largest memory increment: 658KB

Fileline@visit	serial#	mem incr (KB)	variables	constraints	cnst blocks
/env/mvs_atapi_env_sv118@1	1	656	27	37	9
/env/mvs_atapi_env_sv120@1	2	8	3	3	1

**Top randomize calls based on memory increment**

Fileline	calls	mem incr (KB)
/env/mvs_atapi_env_sv118@1	1	656
/env/mvs_atapi_env_sv120@1	1	8

Database: simprofile\_dir

View: Memory Constraint

GO

**Top randomize Fileline**

**Top randomize Pand.Part**

**Fileline@visit**

**Links to the Hierarchical Debug trace**

Visit Count

7-78

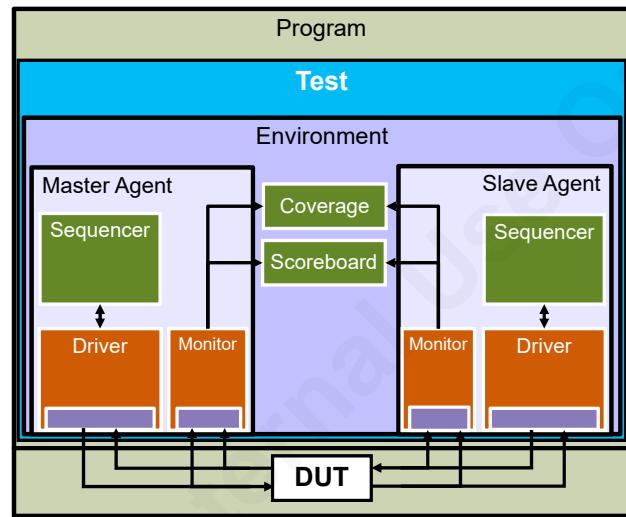
## **Methodology and Best Practices**

**7-79**

# Best Practices: Methodology

## ■ Use a Well Defined Methodology (VMM/UVM)

- Layered testbench to enable VIP reuse
- Name constraints consistently
- Separate test constraints from reusable VIP



7-80

# Best Practices: State Variables

## ■ State Variables (not rand)

- Regular non-random variables or variables with `rand_mode` set to 0
  - ◆ Can still be used in constraints
  - ◆ Often set once at the start of simulation
- Example: chip configuration settings

Non Random

```
class memory_transaction;  
    dma_config cfg;  
    rand bit [31:0] address;  
    constraint cst {  
        (cfg.page_size == 256) -> (address[6:0] == 0);  
        (cfg.page_size == 512) -> (address[7:0] == 0);  
        (cfg.page_size == 1024) -> (address[8:0] == 0);  
    }  
endclass: memory_transaction
```

```
class dma_config;  
    bit [31:0] page_size;  
endclass: dma_config
```

7-81

Alternately

```
class dma_config;  
    rand bit [31:0] page_size;  
    function new();  
        // randomization off by default. User can always turn it on.  
        this.page_size.rand_mode(0);  
    endfunction  
endclass: dma_config
```

## Best Practices: Variable Ranges

### ■ Limit the range to be no larger than necessary

- Use bit variables instead of signed (`int`, `byte`)
- Keep the size of each bit vector as small as possible

```
rand int i;           // 32 bits signed  
rand bit [7:0] v;    // 8 bits unsigned
```

Smaller Range  
Less Memory  
Faster Runtime

### ■ Set values if you don't care about the result

- Solver can optimize constants effectively

```
rand bit [15:0] addr;  
constraint dont_care { addr == 'hdead; }
```

Constant Value  
Less Memory  
Faster Runtime

## Performance: pre-/post-randomize()

- **post\_randomize()**

- Automatically called after **randomize()**
- Procedural Code – executes much faster
- Move arithmetic operations into **post\_randomize()**

```
class slow;
    rand bit [7:0] data[1500];
    constraint cst { foreach(data[i]) data[i]==i; }
endclass: slow
```

Large:  
1500  
Constraint

```
class fast;
    bit [7:0] data[1500];
    function void post_randomize();
        foreach(data[i]) data[i]= i;
    endfunction
endclass: fast
```

Fast:  
Procedural  
Code

7-83

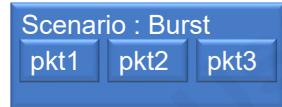
# Best Practices: Randomizing Scenarios

## ■ Scenario - Single Partition

- All random variables are solved at the same time
- Flexible - allowing constraints between all items

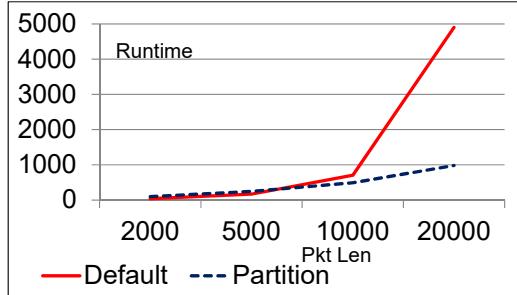
## ■ Scenario - Multiple Partitions

- Divide into multiple randomize calls
- Split scenario into envelope and contents
  - ◆ Identify and solve top-level control variables first
  - ◆ Randomize sub-objects with values from top-level
  - ◆ Post Randomize can also calculate values



7-84

## Performance Example



```
class burst; // Partition
  packet pkt[16];
  rand bit [7:0] id;
  function void post_randomize();
    foreach (pkt[i]) pkt[i].randomize() with {pkt.id==id;}
  endfunction
endclass: burst
```

```
class packet;
  rand bit [ 7:0] id;
  rand bit [15:0] data;
endclass: packet
class burst;//Default
  rand packet pkt[16];
  constraint same_id {
    foreach (pkt[i]) pkt[i].id ==
  pkt[0].id;
  }
endclass: burst
```

7-85

## SV Constraints Best Practices

- Coding Gotchas
- Solution Distribution
- Solver Performance

7-86

# Watch Out for Verilog Rules (1/4)

## ■ Operator Precedence

Requirement	User writes	Simulator does
x is not inside a {0,1,2}	<code>rand int x; constraint c { !x inside {0, 1, 2}; }</code>	(!x) inside {0,1,2} <b>x == 2 is a valid answer!</b>
if mask[i] is 1'b1, addr[i] is also 1	<code>rand bit [7:0] addr, mask; constraint c { addr &amp; mask == mask; }</code>	addr & (mask == mask) <b>addr is anything but 0!</b> <b>mask can be anything!</b>
k is in the (0..size) range	<code>rand int k; int size = 10; constraint c { 0 &lt; k &lt; size; }</code>	(0 < k) < size <b>k can be anything!</b>
if x is true, y is 1; else y is 2	<code>rand bit x; rand bit [15:0] Y; constraint c { y == x ? 1 : 2; }</code>	(y == x) ? 1 : 2 <b>y can be anything!</b>

VCS constraint solver trace (or Verdi) will put extra ( )'s to make it clear what gets evaluated first

7-87

Correct constraints should be:

```
! (x inside {0,1,2})  
  
(addr&mask) == mask  
  
(0<k) && (k<size)  
  
y == (x ? 1 : 2)
```

## Watch Out for Verilog Rules (2/4)

- Bit length promotion

```
class C;  
    rand bit [15:0] m;  
    constraint c { m == 32'hFFFF_FFFF; }  
endclass  
  
C obj = new;  
obj.randomize();
```

Constraint solver failure!  
>> No solution <<

m is 16 bits  
32'hFFFF\_FFFF is 32 bits

Bit length promotion rule says LHS will  
be zero extended to 32 bits (as in RHS)

Constraint (with bit length promoted) becomes:

0000 \_\_\_\_\_ == FFFF\_FFFF

Result: Constraint solver failure due to  
constraint inconsistency (no solution)

## Watch Out for Verilog Rules (3/4)

### ■ Mixing signed and unsigned operands

```
class A;  
    rand int x;  
    rand bit [3:0] y;  
constraint d {  
    x > y;  
    y == 3;  
}  
endclass  
  
A obj = new;  
obj.randomize();
```

x = -1624735218 is a solution

x (int) is signed 32-bit integer  
y (bit [3:0]) is unsigned 4-bit value

Mixing signed and unsigned →

1. Do everything unsigned first
2. Cast to signed, as required, last

x > 32'h0000\_0003, i.e.

valid x solution range: 0000\_0004 .. FFFF\_FFFF

Since x is signed (int)

All values with MSB=1 are negative values

## Watch Out for Verilog Rules (4/4)

### ■ Overflow of arithmetic operations

```
class D1;
    rand bit [31:0] a, b, c;
    constraint e {
        a + b + c == 10;
    }
endclass
```

```
D1 obj = new;
obj.randomize();
```

a = 3478619480
b = 870755917
c = 4240559211

```
class D2;
    rand bit [31:0] a, b, c;
    constraint e {
        a + b + c == 34'd10;
    }
endclass
```

```
D2 obj = new;
obj.randomize();
```

a = 8
b = 0
c = 2

bit length promotion

7-90

In constraint **e** of **class D2** properties **a,b,c** are promoted to 34 bits.

## Watch Out for State Variables (1/4)

- The presence of state variables may cause constraint inconsistency, or conflicts. (i.e. no solution)
- constraints are checked for state variables at call to `randomize()`
  - e.g. variable without `rand`

```
class A;  
  int x;  
  constraint c {  
    x > 0;  
  }  
endclass  
  
A obj = new;  
obj.randomize();
```

```
=====  
Solver failed when solving following set of constraints  
integer x= 0;           x is state variable (no 'rand');  
                           x = 0 (default for 'int')  
constraint c // (from this) (constraint_mode = ON) (t.sv:4)  
{  
  (x > 0);  
}  
=====
```

7-91

## Watch Out for State Variables (2/4)

- rand variable with rand\_mode OFF becomes state variable

```
class B;  
  rand int x;  
  constraint d {  
    x > 0;  
  }  
endclass  
  
B b = new;  
b.x.rand_mode(0);  
b.randomize();
```

```
=====  
Solver failed when solving following set of constraints  
rand integer x = 0; // rand_mode = OFF  
constraint d // (from this) (constraint_mode = ON) (t.sv:10)  
{  
  (x > 0);  
}  
=====
```

x is state variable  
('rand' with rand\_mode OFF);  
x = 0 (default for 'int') =====

## Watch Out for State Variables (3/4)

- Any variable not in randomize ([args]) argument list becomes state variable

```
class C;  
  rand int x,y;  
  constraint e {  
    x > 0;  
  }  
endclass  
  
C obj = new;  
obj.randomize(y);
```

```
=====  
Solver failed when solving following set of constraints  
integer x = 0;  
constraint e />(from this) (constraint_mode = ON) (t.sv:10)  
{  
  (x > 0);  
}  
=====
```

x is state variable  
(not in randomize() arguments);  
x = 0 (default for 'int')

## Watch Out for State Variables (4/4)

### ■ Function calls in constraints create partitions

- Return value of the function is treated as a state variable

```
class A;  
  rand bit [3:0] x, y, z;  
  function bit [3:0] add (bit [3:0] m, n);  
    return (m+n);  
  endfunction  
  constraint c {z == add (x,y);}  
  constraint d {z > 7;}  
endclass  
  
A obj = new;  
obj.randomize();
```

On some seeds this randomize  
may fail due to constraint  
inconsistency

```
=====  
Solver failed when solving following set of  
constraints  
bit[3:0] fv_1 /*this.A::add(x, y)*/ = 4'h7;  
rand bit[3:0] z; // rand_mode = ON  
constraint c // (constraint_mode = ON) (t.v:4)  
{  
  (z == fv_1 /*this.A::add(x, y)*/);  
}  
constraint d // (constraint_mode = ON) (t.v:5)  
{  
  (z > 4'h7);  
}
```

7 - 94

If the constraint is changed to `z== (x+y)`, then `randomize()` will not fail because `x,y` and `z` are solved together; with constraint `z==add (x,y)`, it will fail in some seeds because the solver will first solve `x` and `y` before solving `z` as they are arguments to the function.

## Watch Out for Hierarchical Constraints (1/4)

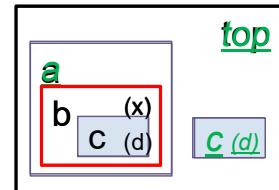
- Effects of rand\_mode on object
- Effects of object aliasing

```
class C;
  rand bit [15:0] d;
  constraint c_d { d == 16'hbeef; }
endclass

class B;
  rand C c;
  rand int x;
  constraint c_x { x == 1234; }
endclass

class A;
  rand B b;
endclass
```

Question:  
top.a.b.c.d = ??  
top.a.b.x = ??



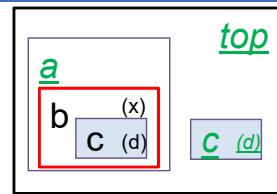
```
class Top;
  rand A a;
  rand C c;
endclass

// after constructing all objects
top.c = top.a.b.c; //obj alias
top.a.b.rand_mode(0);
top.randomize();
```

7-95

## Watch Out for Hierarchical Constraints (2/4)

- Effects of `rand_mode` on object
- Effects of object aliasing



```
top.a.b.rand_mode(0);  
top.randomize();
```

Answer:

`top.a.b.c.d = 'hbeef`

`top.a.b.x = 0`

default value for  
uninitialized 'int'

Variable	Value	Type
@ top	@1	class Top
@ a	@1	rand class A
@ b	@1	rand class B
@ c	@2	rand class C
c_d	'hbeef	constraint
d	'hbeef	rand bit[15:0]
c_x	0	constraint
x	0	rand int
@ c	@2	rand class C
c_d	'hbeef	constraint
d	'hbeef	rand bit[15:0]

7-96

## Watch Out for Hierarchical Constraints (3/4)

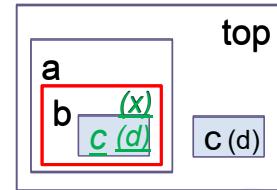
### Effects of rand\_mode on object

```
class C;
  rand bit [15:0] d;
  constraint c_d { d == 16'hbeef; }
endclass

class B;
  rand C c;
  rand int x;
  constraint c_x { x == 1234; }
endclass

class A;
  rand B b;
endclass
```

Question:  
top.a.b.x = ??  
top.a.b.c.d = ??  
top.c.d = ??



```
class Top;
  rand A a;
  rand C c;
endclass

// after constructing all objects
top.c = top.a.b.c; //obj alias
top.a.b.rand_mode(0);
top.a.b.randomize();
```

7-97

## Watch Out for Hierarchical Constraints (4/4)

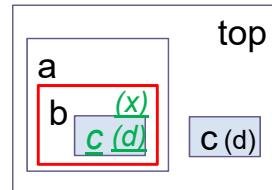
### Effects of `rand_mode` on object

```
top.a.b.rand_mode(0);  
top.a.b.randomize();
```

Answer:

```
top.a.b.x = 1234  
top.a.b.c.d = 'hbeef  
top.c.d = 'hbeef
```

SV LRM 2012 (Section 18.8): "If the random variable is an object handle, only the mode of the handle variable is changed not the mode of the random variables within that object (see global constraints in 18.5.9)"



Data.1\Local.1\Member.1\	
Variable	Value
top	01
a	01
b	01
c	02
x	'hbeef
c_d	1234
d	02
c_x	'hbeef
top	01
.a	'hbeef
.b	'hbeef
.x (rand_mode:ON)	

7-98

The setting

```
top.a.b.rand_mode(0);
```

only prevents randomization of `a.b` when object `a` is randomized. When directly randomizing `a.b`, it only looks at the `rand_mode` settings for the variables inside `b`.

## solve..before Changes the Probabilities

A B	00	01	10	11
00	10%	X	X	X
01	10%	10%	X	X
10	10%	10%	10%	X
11	10%	10%	10%	10%

```
//no solve...before – Uniform Distribution
rand bit [1:0] a, b;
constraint cst { a >= b; }
```

```
//solve a before b
rand bit [1:0] a, b;
constraint cst { a >= b; solve a before b; }
```

A B	00	01	10	11
00	25%	X	X	X
01	12%	12%	X	X
10	8%	8%	8%	X
11	6%	6%	6%	6%

} 25%  
} 25%  
} 25%  
} 25%

```
//solve b before a
rand bit [1:0] a, b;
constraint cst { a >= b; solve b before a; }
```

A B	00	01	10	11
00	6%	X	X	X
01	6%	8%	X	X
10	6%	8%	12%	X
11	6%	8%	12%	25%

25% 25% 25% 25%

7-99

## Recommendations on solve..before, dist

### ■ Recommendations

- First use **dist** to bias distribution
  - ◆ **dist** is more explicit and gives better control

```
rand bit [1:0] a, b;  
constraint cst {  
    a >= b;  
    b dist {  
        0 := 15, 1 := 30, 2 := 5, 3 := 50  
    };  
}
```

A	B	00	01	10	11
00		4%	X	X	X
01		4%	10%	X	X
10		4%	10%	3%	X
11		3%	10%	2%	50%

15%    30%    5%    50%

- If this distribution is not satisfactory, analyze the problem and see if **solve-before** helps

## Implementation Choices

- Given the same design requirement, there are sometimes more than one ways to describe it using constraints
- Understand their impact
  - Readability
  - Runtime Performance
  - Use model differences for the end users
- VCS constraint solver is continuously enhanced
  - Performance may have improved in newer VCS releases

7-101

# Implementation Choices: Example 1

## ■ 2048-bit addr & mask vectors

- If `mask[i]` is 1, `addr[i]` shall be 1, else `addr[i]` is don't-care

```
class cpu_trans;
    rand bit [2047:0] addr;
    rand bit [2047:0] mask;
    constraint slower {
        foreach (mask[i])
            if (mask[i] == 1) addr[i] == 1;
    }
endclass
```

```
class cpu_trans;
    rand bit [2047:0] addr;
    rand bit [2047:0] mask;
    constraint faster {
        (addr & mask) == mask;
    }
endclass
```

VCS	CPU Time
2011.12-2	4.8 seconds

VCS	CPU Time
2011.12-2	0.48 seconds

## Implementation Choices: Example 2 (1/3)

- Need to generate random non-overlapping ranges



7-103

## Implementation Choices: Example 2 (2/3)

- Start and end are not inside any other ranges

```
class mem;
    rand bit [31:0] low[$];
    rand bit [31:0] high[$];
    constraint cst {
        foreach (low[i]) {
            low[i] < high[i];
            foreach (low[j]) {
                (i != j) -> !(low[i] inside {[ low[j] : high[j] ]});
                (i != j) -> !(high[i] inside {[ low[j] : high[j] ]});
            }
        }
    }
endclass
```

low[k] ... high[k]    low[p] ... high[p]    low[m] ... high[m]

Problem : Number of constraints order( $n^{**}2$ )

Does not scale as number of ranges increases

7-104

## Implementation Choices: Example 2 (3/3)

- Start and end are in order, and increasing

```
class mem;
    rand bit [31:0] low[$]
    rand bit [31:0] high[$];
    constraint cst {
        foreach (low[i]) {
            (i > 0) -> (low[i] < high[i]);
            (i > 0) -> (low[i] > high[i-1]);
        }
    }
endclass
```

The diagram illustrates the constraints for three ranges. Three boxes labeled "low[0] ... high[0]", "low[1] ... high[1]", and "low[2] ... high[2]" are shown. Arrows point from each of these boxes to the corresponding range constraints within the "cst" block of the SystemVerilog code.

Better: Number of constraints order(n)  
Scales with number of ranges

7-105

This page was intentionally left blank

# Agenda

DAY  
3

- |    |  |   |
|----|--|---|
| 8  | Object Oriented Programming (OOP)<br>– Inheritance |  |
| 9  | Inter-Thread Communications                        |  |
| 10 | Functional Coverage                                |  |
| 11 | SystemVerilog UVM Preview                          |  |
| 12 | Customer Support                                   |  |

## Unit Objectives



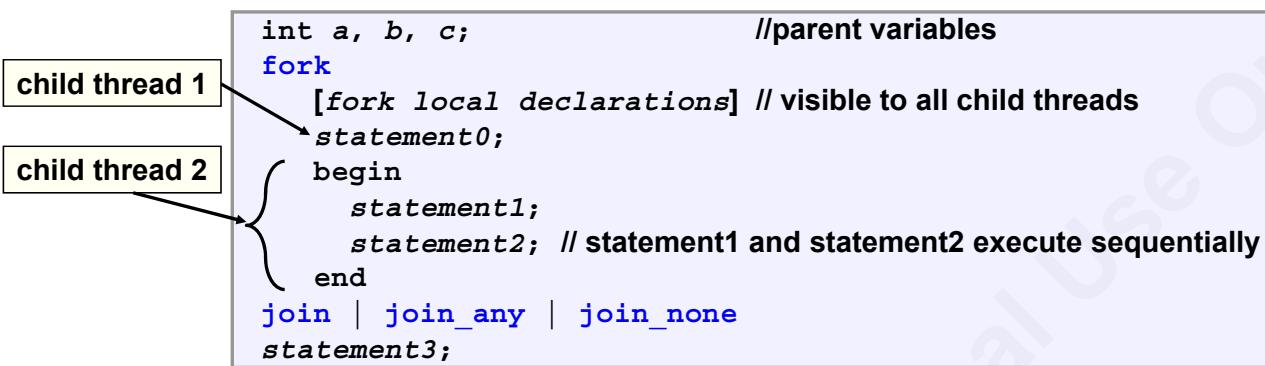
**After completing this unit, you should be able to:**

- Create OOP extended classes
- Access class members in inheritance hierarchy

8-2

## Day 2 Review - Creating Concurrent Threads

- Concurrent threads are created in a fork-join block



- Statements enclosed in **begin-end** in a **fork-join** block are executed sequentially as a single concurrent child thread
- No predetermined execution order for concurrent threads
- Parent variables cannot be referred to in **join\_any** or **join\_none** except to initialize variables in fork local declarations

8-3

Make copies of parent variable that are needed by any threads, using local declarations in the **fork** declarative area. These are visible to all threads.

## Day 2 Review - OOP Class

- Similar to a module, an OOP **class** encapsulates:
  - Variables (**properties**) used to model a system
  - Subroutines (**methods**) to manipulate the data
  - Properties & methods are called **members** of class

Class properties and methods  
are visible inside the class

```
class Packet;  
    string      name; //Packet properties  
    bit[3:0]   sa, da;  
    bit[7:0]   payload[];  
  
    extern task send();  
    extern task send_addrs();  
    extern task send_pad();  
    extern task send_payload();  
endclass: Packet  
  
task Packet::send(); //Packet methods  
    send_addrs();  
    send_pad();  
    send_payload();  
endtask: send  
task Packet::send_addrs();...endtask  
task Packet::send_pad();...endtask  
task Packet::send_payload();...endtask
```

8-4

Unlike procedural languages, classes allow data and functionality to be grouped together. This helps in creating and maintaining large verification projects.

## Day 2 Review - OOP Based Randomization

- **randomize()** function built into each class

- Randomizes each **rand** and **randc** property to a value within full range of its data type unless constrained with a **constraint** block.

```
class Packet;    Declare random properties in class
    randc bit[3:0] sa, da;
    rand  bit[7:0] payload[];
    constraint valid {
        payload.size() >= 2;
    }
    extern function void display(...);
    extern function Packet copy(...);
    extern function bit compare(...);
endclass: Packet
```

2      **Optionally constrain random properties**

```
program automatic test;
```

```
int run_for_n_pkts = 100;
Packet pkt = new();
```

```
initial begin
```

```
...
```

```
repeat (run_for_n_pkts) begin
```

```
    if(!pkt.randomize()) ...;
```

```
    fork
```

```
        send();
        recv();
    
```

```
    join
```

```
    check();

```

```
end
```

```
end
```

```
endprogram: test
```

3      **Construct an object to be randomized**

4      **Randomize properties of object**

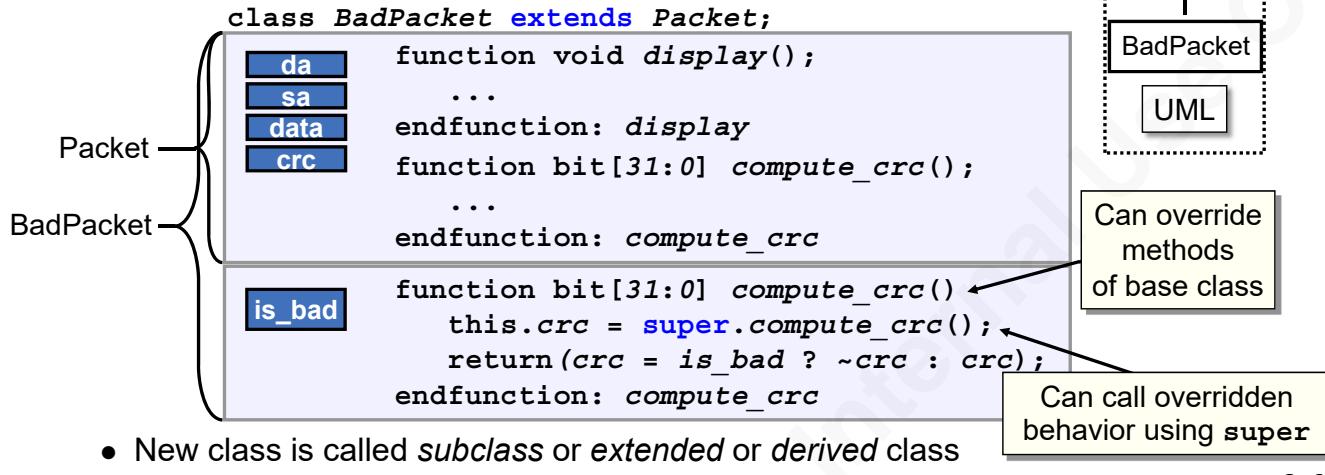
8-5

`randomize()` only generates integral types and packed structs. It cannot generate non-integral types like reals, strings, class objects, and unpacked structs and unions that contain non-integral members.

# Object Oriented Programming: Inheritance

## ■ Object-oriented programming

- New classes derived from original (*base or super*) class
- New class inherits all contents of base class

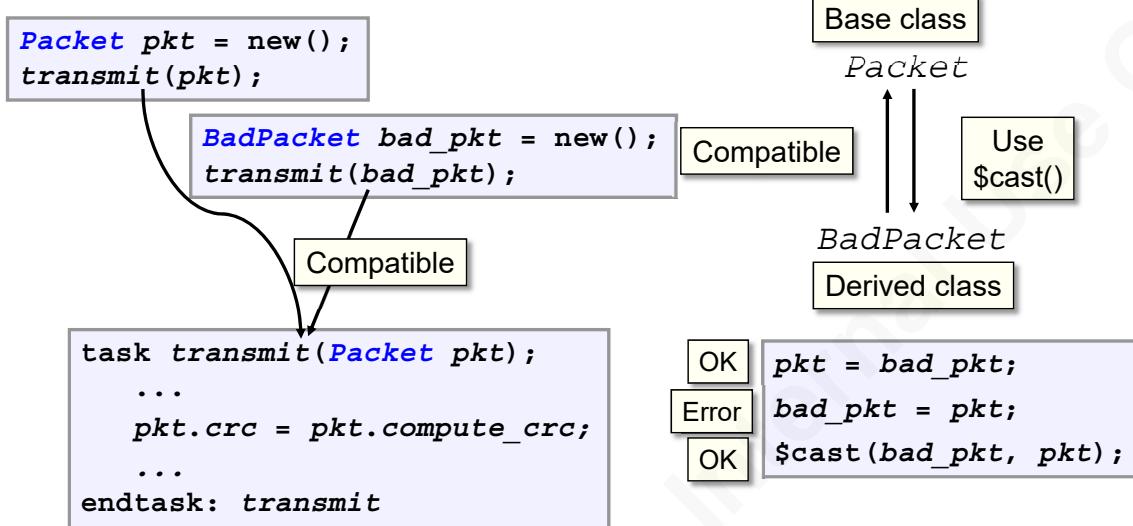


8-6

# Object Oriented Programming: Inheritance

## ■ Derived classes compatible with base class

- Can reuse code that uses the base class



8-7

Now look at how you would use these objects.

On the left is a diagram showing how a task can pass a *packet* object to the *transmit* method.

This method will read and write the class members such as *sa*, and *compute\_crc()* with *pkt.sa*, and *pkt.compute\_crc()*.

Or, a task could create a *bad\_packet*. It can pass this to *transmit()* as every *bad\_packet* is also a *packet*, with *sa*, and *compute\_crc()*.

On the lower right, this idea is shown by the assignment *pkt = bad\_pkt*; After this assignment, the handle *pkt* can still refer members such as *pkt.sa* and *pkt.compute\_crc()*.

But, the SystemVerilog compiler does not allow a base handle to be assigned to an extended handle. If it allowed *bad\_pkt = pkt*; then potentially the handle *bad\_pkt* could point to a *packet* object. If this happened, *bad\_pkt.bad* would refer to a variable that does not exist in the object. So this statement causes a compile-time error.

However, if *pkt* pointed to a *bad\_packet* object, then it is legal to make the handle *bad\_pkt* point to the same object. The only way to know is to check the type of the object pointed to by *pkt*. At run-time, the statement *\$cast(bad\_pkt, pkt)* first checks the type of the object that *pkt* points to. If the object is of type *bad\_packet* or is extended from *bad\_packet*, the handle *bad\_pkt* is assigned the value of the *pkt* handle. If *pkt* does not point to a compatible object, VCS will terminate simulation.

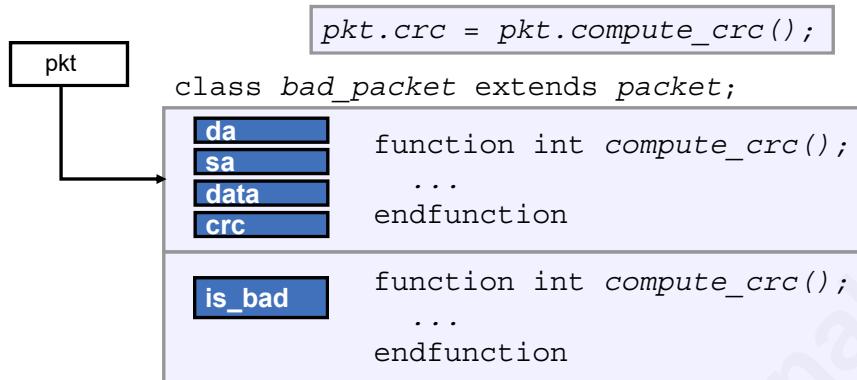
When you call *\$cast* as a task, it will cause termination of simulation if the source object is not type compatible with the destination handle. However, if you call *\$cast* as a function, it silently returns a 1 for success, and 0 for failure.

```
if (!$cast(bad_pkt, pkt))
```

```
$display("pkt type is not compatible with bad_pkt handle");
```

# OOP: Polymorphism

## ■ Which method gets called?



## ■ Depends on

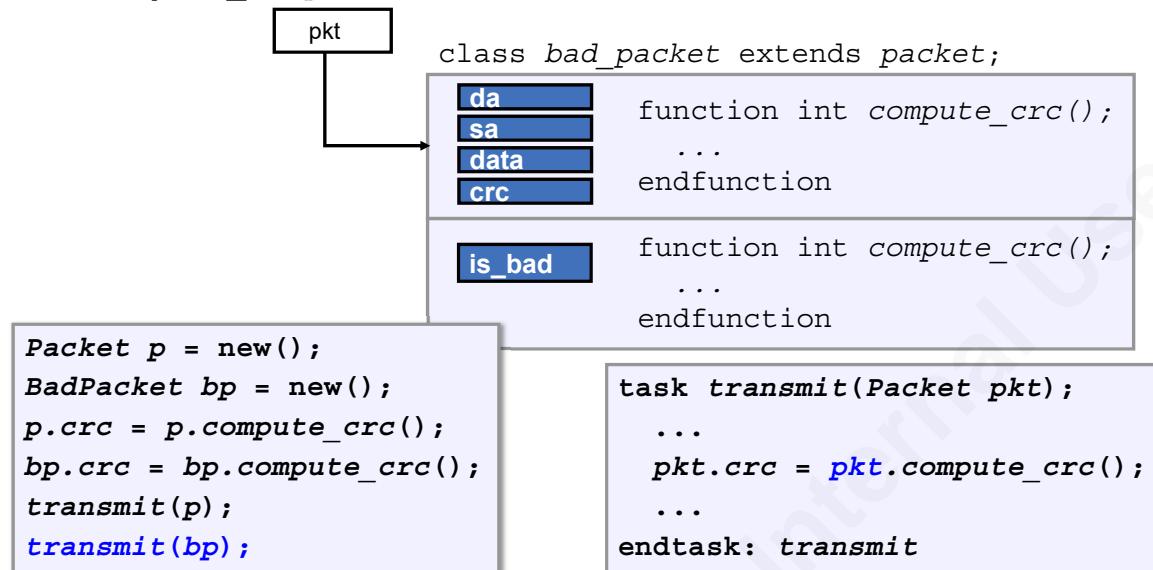
- Type of handle  $p$  (e.g. “`packet`” or “`bad_packet`” ?)
- Whether `compute_crc()` is `virtual` or not

8-8

There are two `compute_crc` methods. Which one is called when you call  $p.compute\_crc()$ ? It depends on the type of the  $p$  handle, `packet` or `bad_packet`, and if `compute_crc` is `virtual` or not.

# OOP: Polymorphism

- If `compute_crc()` is not virtual – base class method is called



8-9

In the example above,

`p.compute_crc()` executes the `compute_crc()` method in `packet`

`bp.compute_crc()` executes the `compute_crc()` method in `bad_packet`

And,

`transmit(p)` executes the `compute_crc()` method in `packet`

But,

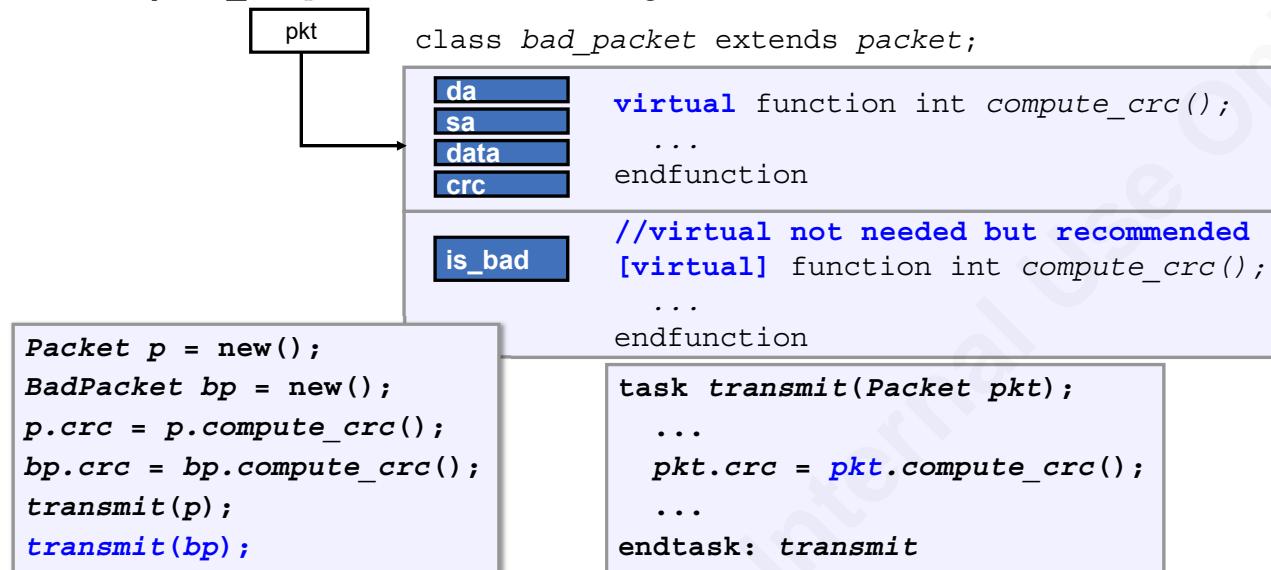
`transmit(bp)` also executes the `compute_crc()` method in `packet` method despite the fact that `bp` was the object passed in.

The reason is that the object executing `compute_crc()`, `pkt`, is a `packet` object. If a method within a scope is not declared as `virtual`, then the local scope version of that method will be executed.

If `compute_crc()` is not `virtual`, then polymorphism is disabled in methods.

# OOP: Polymorphism

- If `compute_crc()` is virtual – overriding method is called



8-10

Once a subroutine has been described as virtual it is always virtual even if you leave out the `virtual` keyword in derived classes. The signatures of virtual functions in all generations of derived classes must now match. This means that functions that return base class handles must continue returning base class handles that point to derived objects. The order, number, names, directions and types of all arguments must also match exactly.

You must use `$cast` when assigning return values of these functions to derived object handles.

```
class BadPacket extends Packet;
    ...
    virtual function Packet clone(); // returns Packet handle
    ...
endfunction: clone
...
endclass: BadPacket

initial begin
    BadPacket bpkt2, bpkt1 = new();
    $cast(bpk2, bpkt1.clone()); // because clone() returns handle of type Packet
end
```

# Modifying Constraints for Test Cases

## ■ Define test-specific constraints in derived classes

- Can also override existing constraints

```
class data;  
    rand bit[31:0] x, y;  
    constraint valid {  
        x > 0; y >= 0;  
    }  
endclass: data
```

```
class Generator;  
    data blueprint;  
    ...  
    while(...) ...  
    blueprint.randomize();  
...  
endclass: Generator
```

```
program automatic test_corner_case;  
    class test_data extends data;  
        constraint corner_case {  
            x == 5; y == 10;  
        }  
    endclass: test_data  
  
    initial begin  
        test_data tdata = new();  
        Generator gen = new();  
        gen.blueprint = tdata; //polymorphism  
        ...  
    end  
endprogram: test_corner_case
```

8-11

Constraints are virtual like virtual tasks and functions. So you can override existing constraint blocks in derived classes. You can also define new constraints in the derived classes.

A common way to change constraints on data classes is by replacing a blueprint base object inside a data generator with a derived object. The generator now randomizes the data objects using constraints defined in the derived class.

## Data Protection: local

- local members of a base class are not accessible in the derived class

```
class BadPacket extends Packet;
    local int DONE;
    function void display();
        ...
    endfunction: display
    function bit[31:0] compute_crc();
        ...
    endfunction: compute_crc

    function bit[31:0] compute_crc()
        this.crc = super.compute_crc();
        if ( is_bad ) crc = ~crc;
        DONE = 1'b1;
    endfunction: compute_crc
```

is\_bad    Error

Packet

BadPacket

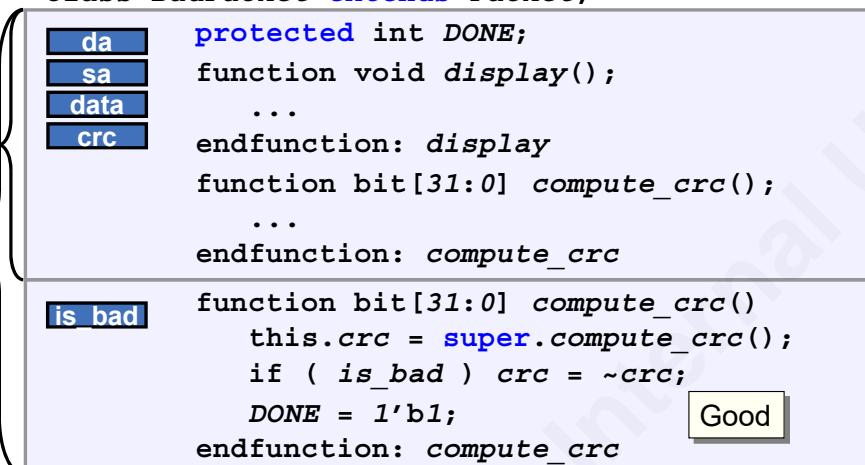
8-12

## Data Protection: protected

- **protected** members of a base class are accessible in the derived class, but not to external code

```
class BadPacket extends Packet;
    protected int DONE;
    function void display();
        ...
    endfunction: display
    function bit[31:0] compute_crc();
        ...
    endfunction: compute_crc
    function bit[31:0] compute_crc()
        this.crc = super.compute_crc();
        if (is_bad) crc = ~crc;
        DONE = 1'b1;
    endfunction: compute_crc
```

is bad      Good



Packet

BadPacket

8-13

# Constructing Derived Class Objects

## ■ When constructing an object of a derived class

- If the derived class does not have a constructor defined VCS inserts one

```
function new();
    super.new();
endfunction
```

- If the derived class defines a constructor SystemVerilog expects its first procedural statement to be

```
super.new([args])
```

- ◆ Must be called with the correct set of arguments
- ◆ A syntax error results if called anywhere except as the first statement
- ◆ If call is missing, the compiler inserts one without arguments, as the first procedural statement of the function

```
super.new()
```

# Test for Understanding 1

- Will the following code compile?
- Which one of the task new() is executed?



```
class A;  
  protected int a;  
  function int get_a();  
    get_a = a;  
  endfunction: get_a  
  function new(int b);  
    a = b;  
  endfunction  
endclass: A  
  
class B extends A;  
  protected int b = 1000;  
  task print_a();  
    $display("a is %d", get_a());  
  endtask: print_a  
endclass: B  
  
class C extends B;  
  function new(int c);  
    a = c;  
  endfunction  
endclass: C  
  
program automatic test;  
  C test_c = new(10);  
  test_c.print_a();  
endprogram: test
```

8-15

## Test for Understanding 1: Answers

- VCS will attempt to execute every task new() starting with new() of class C, resulting in a syntax error

```
class A;
    protected int a;
    function int get_a();
        get_a = a;
    endfunction: get_a
    function new(int b);
        a = b;
    endfunction
endclass: A
```

```
class B extends A;
    protected int b = 1000;
    task print_a();
        $display("a is %d", get_a());
    endtask: print_a
    function new();
        super.new();
    endfunction
endclass: B
```

```
class C extends B;
    function new(int c);
        super.new();
        a = c;
    endfunction
endclass: C
```

**Error: Mismatching argument list**

**Inserted by VCS**

8-16

## Test for Understanding 1: Guideline

- Always call `super.new()` as the first procedural statement in constructor, with correct argument set

```
class A;  
    protected int a;  
    function int get_a();  
        get_a = a;  
    endfunction  
    function new(int b);  
        a = b;  
    endfunction  
endclass: A
```

```
class B extends A;  
    protected int b = 1000;  
    task print_a();  
        $display("a is %d", get_a());  
    endtask  
    function new(int b);  
        super.new(b);  
    endfunction  
endclass: B
```

```
class C extends B;  
    function new(int c);  
        super.new(c);  
        a = c;  
    endfunction  
endclass: C
```

**Inserted by user**

```
graph TD; A["class A;"] --> B["class B extends A;"]; B --> C["class C extends B;"];
```

8-17

## Test for Understanding 2

- Given the following class inheritance hierarchy, is the program code legal?

```
class A;
    protected int a;
    function int get_a();
        get_a = a;
    endfunction: get_a
    function new(int b);
        a = b;
    endfunction
endclass: A
program automatic
test;
    C obj_c = new(10);
    B obj_b = obj_c;
endprogram: test
```

```
class B extends A;
    protected int b = 1000;
    task print_a();
        $display("a is %d", get_a());
    endtask: print_a
    function new(int b);
        super.new(b);
    endfunction
endclass: B
```

```
class C extends A;
    function new(int c);
        super.new(c);
        a = c;
    endfunction
endclass: C
```



8-18

## Test for Understanding 2: Answer

- Both classes, B and C extend from base class A, but they are unrelated. A handle of one object can not point to its sibling object

```
class A;  
    protected int a;  
    function int get_a();  
        get_a = a;  
    endfunction  
    function new(int b);  
        a = b;  
    endfunction  
endclass: A  
  
program automatic  
test;  
    C obj_c = new(10);  
    B obj_b = obj_c;  
endprogram: test
```

```
class B extends A; ← B and C, both derived from A  
    protected int b = 1000;  
    task print_a();  
        $display("a is %d", get_a());  
    endtask: print_a  
    function new(int b);  
        super.new(b); ← class C extends A;  
    endfunction  
endclass: B
```

```
class C extends A;  
    function new(int c);  
        super.new(c);  
        a = c;  
    endfunction  
endclass: C
```

8-19

## Quiz Time

8-20

# Inheritance: Quiz 1

```
program automatic test1 ;  
  
class abc ;  
    rand int a ;  
endclass  
  
class xyz extends abc ;  
    rand int b ;  
endclass  
  
initial begin  
  
    abc o1 = new() ; xyz o2 = new() ;  
    o1 = o2 ;  
    $display("test: o1 = %d", o1.b) ;  
  
end  
  
endprogram: test1
```

1. Will this code compile without errors?
  - If not, why not?
2. Will it throw any runtime errors?
3. What will the program display?

8-21

1. No - A base class handle can point to an extended class but not access properties of an extended class.  
Error-[MFNF] Member not found  
test1.sv, 15  
"o1."  
Could not find member 'b' in class 'abc', at "test1.sv", 3.  
"o1."  
I error
2. NA
3. NA

## Inheritance: Quiz 2

```
program automatic test1 ;
class abc ;
    rand int a = 10 ;
    function void prnt_a() ;
        $display("abc: a= ", a) ;
    endfunction
endclass
class xyz extends abc ;
    function void prnt_a() ;
        $display("xyz: a= ", a) ;
    endfunction
endclass
initial begin
    abc o1 = new() ; xyz o2 = new() ;
    o1 = o2 ;
    o1.prnt_a() ;
end
endprogram: test1
```

1. Will this code compile without errors?
  - If not, why not?
2. Will it throw any runtime errors?
3. What will the program display?

8-22

1. Yes
2. No
3. abc: a=10

## Inheritance: Quiz 3

```
program automatic test1 ;
class abc ;
    rand int a = 10 ;
    virtual function void prnt_a() ;
        $display("abc: a= ", a) ;
    endfunction
endclass
class xyz extends abc ;
    virtual function void prnt_a()
        $display("xyz: a= ", a) ;
    endfunction
endclass
initial begin
    abc o1 = new() ; xyz o2 = new() ;
    o1 = o2 ;
    o1.prnt_a() ;
end
endprogram: test1
```

1. Will this code compile without errors?
2. Will it throw any runtime errors?
3. What will the program display?
4. Why did display change from Quiz 2?

8-23

1. Yes
2. No
3. xyz:a=10
4. Because the function **prnt\_a()** is **virtual**

## Inheritance: Quiz 4

```
program automatic test1 ;
class abc ;
    rand int a = 10 ;
    virtual function void prnt_a() ;
        $display("abc: a= ", a) ;
    endfunction
endclass
class xyz extends abc ;
    virtual function void prnt_a() ;
        $display("xyz: a= ", a) ;
    endfunction
endclass
initial begin
    abc o1 = new() ; xyz o2 = new() ;
    o2 = o1 ;
    o2.prnt_a() ;
end
endprogram: test1
```

1. Will this code compile without errors?
  - If not, why not?
2. Will it throw any runtime errors?
3. What will the program display?

8-24

3. NA  
2. NA

1 error.

Please make sure that the lhs and rhs expressions are compatible.

be assigned to a class handle on lhs.

Expression 'o1' on rhs is not a class or a compatible class and hence cannot

"o2 = o1;"

test1.sv, 15

Error-[SV-ICA] Illegal class assignment

1. No - A extended class handle can not point to a base class object.

## Inheritance: Quiz 5

```
program automatic test1 ;
class abc ;
    rand int a = 10 ;
    virtual function void prnt_a() ;
        $display("abc: a= ", a) ;
    endfunction
endclass
class xyz extends abc ;
    virtual function void prnt_a() ;
        $display("xyz: a= ", a) ;
    endfunction
endclass
initial begin
    abc o1 = new() ; xyz o2 = new() ;
    $cast(o2 , o1) ;
    o2.prnt_a() ;
end
endprogram: test1
```

1. Will this code compile without errors?
  - If not, why not?
2. Will it throw any runtime errors?
  - If yes, why?
3. What will the program display?

8-25

3. NA

Please ensure matching types for dynamic cast

due to type mismatch.

Castimg of source class type 'abc' to destination class type 'xyz' failed

test1.sv, 15

Error-[DCF] Dynamic cast failed

2. Yes - because at runtime o1 is not compatible with o2. The \$cast will fail.

1. Yes

## Unit Objectives Review

**Having completed this unit, you should be able to:**

- **Create OOP extended classes**
- **Access class members in inheritance hierarchy**



**8-26**

# **Appendix**

## **Interface Class**

**8-27**

## Interface Class

8-28

# Interface Class

## ■ Interface class is defined using keyword `interface`

- Contains only `pure virtual` methods, type declarations and parameter declarations
- Can not contain properties, cover groups or constraint blocks
- Is implemented by non interface class using `implements` keyword
- Can inherit from multiple interface classes (`multiple inheritance`) using `extends` keyword

```
interface class intf_cls;  
    pure virtual task print();  
endclass
```

***myclass must implement the  
methods prototyped in the  
interface classes it implements***

```
class myclass implements intf_cls;  
    virtual task print();  
        $display("myclass");  
    endfunction  
endclass
```

8-29

Nothing in an interface class is inherited by the implementing class. All parameters and typedefs within an interface class are static and can be accessed through the class scope resolution operator ::

# Interface Class – Rules

## ■ implements v/s extends

- **extends** - used to add to or modify the behavior of a base class
- **implements** - a requirement to provide implementations for the pure virtual methods defined in an interface class

## ■ Interface class

- May extend zero or multiple interface classes
- Can not implement another interface class
- Can not extend a non-interface class

## ■ Non interface class

- Can not extend interface class
- Can implement zero or more interface classes
- Can extend only one other class
- Can simultaneously implement one or more interface class and extend a class

8-30

# Interface Class – Multiple Inheritance

- By implementing multiple interface classes one can achieve multiple inheritance

```
interface class PutImp#(type PUT_T = logic);
    pure virtual function void put(PUT_T a);
endclass

interface class GetImp#(type GET_T = logic);
    pure virtual function GET_T get();
endclass

class Fifo#(type T = logic, int DEPTH = 1) implements PutImp#(T), GetImp#(T);
    T myFifo [:$:DEPTH-1];
    virtual function void put(T a);
        myFifo.push_back(a);
    endfunction
    virtual function T get();
        get = myFifo.pop_front();
    endfunction
endclass
```

Fifo "inherits" from both PutImp and GetImp

8-31

We can now define another class Stack that also implements the same two interface classes.

```
class Stack#(type T = logic, int DEPTH = 1) implements
    PutImp#(T), GetImp#(T);

    T myFifo [:$:DEPTH-1];
    virtual function void put(T a);
        myFifo.push_front(a);
    endfunction
    virtual function T get();
        get = myFifo.pop_front();
    endfunction
endclass
```

Stack is not related in any way to the Fifo class.

# Interface Class – Partial Implementation

## ■ Partial implementation of interface class

- Use virtual class to create partially defined class
  - ◆ Virtual class can implement some interface methods
  - ◆ Virtual class must define prototype for methods not implemented by it

```
interface class IntfClass;  
    pure virtual function bit funcA();  
    pure virtual function bit funcB();  
endclass
```

```
virtual class ClsA implements IntfClass;  
    virtual function bit funcA();  
        return (1);  
    endfunction  
    pure virtual function bit funcB();  
endclass
```

```
class ClsB extends ClsA;  
    virtual function bit funcB();  
        return (0);  
    endfunction  
endclass
```

ClsA only implements funcA  
ClsA must be virtual  
funcB prototype required

ClsB is complete implementation  
funcA inherited from ClsA  
funcB implemented

8-32

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract (virtual)

An inherited virtual method can provide the implementation for a method of an implemented interface class.

ClsB fulfills its requirement to implement IntfClass by providing an implementation of funcB and by inheriting an implementation of funcA from ClsA.

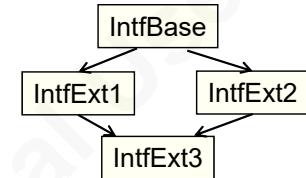
An inherited non-virtual method does not provide an implementation for a method of an implemented interface class

# Interface Class – Multiple Extends and Diamond Relationship

## ■ Only interface classes can extend from multiple interface classes

- A diamond relationship occurs if an interface class is implemented by the same class or inherited by the same interface class in multiple ways
- Only one copy of the symbols from any single interface class will be merged, to avoid name conflict

```
interface class IntfBase;  
parameter SIZE = 64;  
endclass  
interface class IntfExt1 extends IntfBase;  
    pure virtual function bit funcExt1();  
endclass  
interface class IntfExt2 extends IntfBase;  
    pure virtual function bit funcExt2();  
endclass  
interface class IntfExt3 extends IntfExt1, IntfExt2;  
endclass
```



8-33

A *diamond relationship* occurs if an interface class is implemented by the same class or inherited by the same interface class in multiple ways.

Only one copy of the symbols from any single interface class will be merged so as to avoid a name conflict.

When a class implements multiple interface classes, or when an interface class extends multiple interface classes, identifiers are merged from different name spaces into a single name space. When this occurs, it is possible that the same identifier name from multiple name spaces may be simultaneously visible in a single name space creating a name conflict that must be resolved.

This page was intentionally left blank

# Agenda

DAY  
3

- |    |  |   |
|----|--|---|
| 8  | Object Oriented Programming (OOP)<br>– Inheritance |  |
| 9  | Inter-Thread Communications                        |  |
| 10 | Functional Coverage                                |  |
| 11 | SystemVerilog UVM Preview                          |  |
| 12 | Customer Support                                   |  |

## Unit Objectives



After completing this unit, you should be able to:

- Establish order of execution using events
- Avoid resource collision with Semaphores
- Pass data between threads via Mailbox

# Inter-Thread Communications (ITC)

- Concurrent threads require communication to establish control for sequence of execution
- Three types are covered in this unit

**Event based**



**Resource sharing**



**Data passing**



9-3

## Event Based ITC



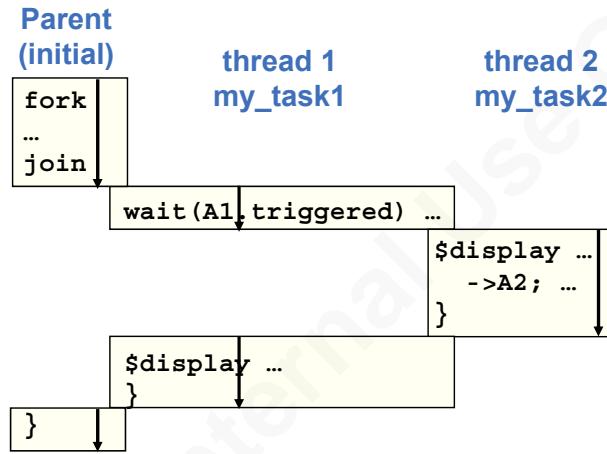
- **Synchronize operation of concurrent threads via event variables:**
  - Thread **waits** for **event** to be triggered
  - An executing thread triggers the **event** to enable the waiting thread to be placed into the READY queue
- **Mainly used to control sequence of execution**

9-4

# Event Based ITC Example

## ■ Controlling order of execution

```
initial begin
    event A;
    fork
        my_task1(A); //thread 1
        my_task2(A); //thread 2
    join
end
task my_task1(event A1);
    wait(A1.triggered);
    $display("Print 2nd");
endtask
task my_task2(event A2);
    $display("Print 1st");
    ->A2;
endtask
```



9-5

## Event Wait Syntax

- **Wait until event has been triggered (one shot):**

```
@(event_var [, event2]);
```

- Will be satisfied by events which occur after the execution of this statement

- **Wait until event has been triggered (persistent):**

```
wait(event_var.triggered);
```

- Similar to @(event\_var), but will also be satisfied by events which happened earlier during the same simulation time
- Can be used to eliminate potential race condition

# Trigger Syntax

->eventN;

- event variables are handles
- Assigning an event to another makes both events the same event

**Example:**

```
event a, b, c;  
a = b;  
-> c;  
-> a; // also triggers b  
-> b; // also triggers a  
a = c;  
b = a;  
-> a; // also triggers b and c  
-> b; // also triggers a and c  
-> c; // also triggers a and b
```

9-7

# Controlling Termination of Simulation

```
program automatic test(router_io.TB rtr_io);
  event DONE;
  initial begin
    fork
      gen();
      check();
    join_none
    for (int i=0; i<16; i++) begin
      int j = i;
      fork
        send(j);
        recv(j);
      join_none
    end
    wait(DONE.triggered)
  end
  ...
endprogram:
```

task check();  
 forever begin  
 ...  
 if (\$get\_coverage() == 100)  
 ->DONE;  
 end  
endtask: check

Trigger event when termination condition is detected

Blocking statement to prevent termination of simulation until done

9-8

## Resource Sharing ITC



- **Synchronize operation of concurrent threads via access to shared resources:**

- Thread requests for a shared resource before executing a critical section of code
- Thread waits if the requested resource is unavailable
- Thread resumes execution when the requested resource becomes available

## Semaphores (1/2)

- A Semaphore is a bucket in which keys can be deposited and removed:
  - A thread tries to **acquire** keys from a **semaphore** bucket
  - Thread execution **waits** until keys requested are available in the **semaphore** bucket
  - When the requested keys becomes available, thread execution **resumes** and the requested keys removed from the **semaphore** bucket
- Mainly used to prevent multiple threads from accessing the same hardware signal or using same software resource

9-10

## Semaphores (2/2)

- Semaphores are supported via built-in semaphore class:

```
class semaphore;
    function new(int keyCount = 0);
    task put(int keyCount = 1);

    // the following is blocking:
    task get(int keyCount = 1);

    // the following is non-blocking: 1 for success 0 for failure
    function int try_get(int keyCount = 1);
endclass
```

9-11

# Using Semaphores

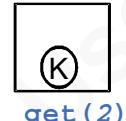
## ■ Semaphore buckets created using the constructor `new()`

```
semaphore sem;  
sem = new(3);           // bucket with 3 keys is created
```



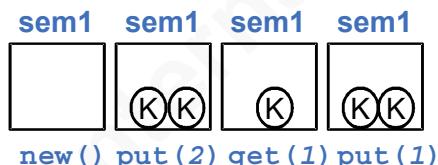
## ■ Acquiring Keys

```
sem.get(2);           // bucket with 1 key left  
if (!sem.try_get(2)) // does not block  
    $display("Failed");  
sem.get(2);           // blocks until 2 keys available
```



## ■ Returning/Creating Keys

```
semaphore sem1;  
sem1 = new(); // 0 keys  
sem1.put(2); // Create 2 keys  
sem1.get(1); // Use 1 key  
sem1.put(1); // Return 1 key
```



9-12

The `get()` implementation is a greedy algorithm. If there are not enough keys to service the `get()` call, the keys are left in the bucket without reservation. Be careful using semaphore schemes which require a `get()` for multiple keys. If implemented improperly, a starvation issue may arise.

Example:

```
semaphore sem = new(2);  
sem.get(1);  one key is left  
fork  
begin:thread1 sem.get(2); end // does not reserve key  
begin:thread2 sem.get(1); end // grabs the one remaining key  
join_any  
sem.put(1);   // thread1 does not reserve this key  
sem.get(1);   // grabs the one key  
sem.put(2);   // thread1 finally gets served
```

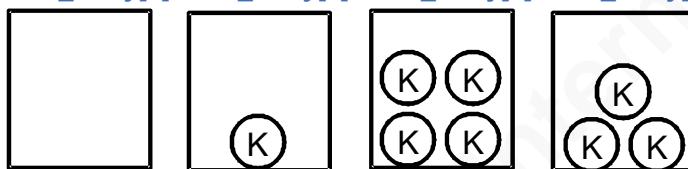
There is no limit on number of keys that can be put in a bucket. Also, there is no requirement that a thread returns only as many keys as it took. A thread can deposit keys into a semaphore bucket even without having taken any. This is often useful when resources do not exist at the start of simulation. During simulation, as the resources are created, keys can be deposited into the semaphore buckets to represent these dynamically created resources.

# Using Semaphore Arrays

- Semaphores in an array must be constructed before use

```
semaphore sem_array[];  
sem_array = new[4]; // array of 4 null semaphore handles created  
sem_array[2].put(2); // ERROR at run time: semaphore bucket does not exist!  
foreach (sem_array[i])  
    sem_array[i] = new(i); // construct semaphore with i keys  
sem_array[2].put(2); // okay. sem_array[2] now has 4 keys
```

sem\_array[0] sem\_array[1] sem\_array[2] sem\_array[3]



9-13

## Arbitration Example Using Semaphore

```
program automatic test(router_io.TB rtr_io);
    semaphore sem[];
    ...
    sem = new[16];
    foreach(sem[i])
        sem[i] = new(1);           Create semaphore array to
                                represent each output port
    task send();
        sem[da].get(1);          Construct each individual
        send_addrs();            semaphore bucket
        send_pad();
        send_payload();
        sem[da].put(1);          Block if others are driving the
                                chosen output port
    endtask: send             Re-deposit keys when done
    ...;                      with driving port
endprogram: test
```

9-14

# Mailbox



- **Messages are passed between threads via mailbox:**
  - A thread **sends data** by putting messages a mailbox
  - A thread **retrieves data** by getting messages from the mailbox
  - If a message is not available, the thread can **wait** until there is a message to retrieve
  - Thread **resumes** execution once the message becomes available
- **Mainly used for passing data between concurrent threads**

9-15

## Mailbox Class

- Mailboxes are supported via built-in mailbox class:

```
class mailbox #(type T = dynamic_type);
    function new(int bound = 0);
    function int num(); // return # of messages
    task put(T message); // wait if mailbox full
    task get(ref T message); // wait if no message
    task peek(ref T message); // wait if no message
    function int try_put(T message);
    function int try_get(ref T message);
    function int try_peek(ref T message);
endclass
```

9-16

If a type is not provided the mailbox can deliver any type of data. This is not recommended.

The **try\_** methods are non-blocking.

**peek()** retrieves a copy of the content of the mailbox without removing the content from the mailbox.

## Creating Mailboxes

```
mailbox #(Packet) mbx[];           // Packets only mailbox array
mailbox #(instr_e) mbx_unbound;    // enum type instr_e
mbx_unbound = new();              // mailbox size is unbounded
mbx = new[4];                    // array of 4 null mailbox handles created
for (int i=0; i<mbx.size(); i++)
begin
    mbx[i] = new(i+2);           // bound to max of i+2 messages
end
```



9-17

# Putting Messages into Mailboxes

```
task put(message); // block if exceeds bound  
function int try_put(message); // non-blocking
```

Example:

```
int status;  
typedef enum {ADD=1, SUB, MUL, DIV} instr_e;  
instr_e instr = ADD;  
mailbox #(instr_e) mbox = new(1);  
Packet pkt = new();  
mbox.put(instr);  
status = mbox.try_put(instr); //status = 0 - mailbox full  
mbox.put(pkt); //incorrect type - compiler error
```



9-18

Mailbox shown after compiler error corrected.

## Retrieving Messages from Mailboxes (get)

```
task get(ref message); // block if mailbox is empty  
function int try_get(ref message); // non-blocking  
// return number of messages in mailbox before try_get, 0 indicates no messages
```

**Example:**

```
int status;  
typedef enum {ADD=1, SUB, MUL, DIV} instr_e;  
instr_e instr1 = MUL, instr2; Packet pkt;  
mailbox #(instr_e) mbox = new(1);  
mbox.put(instr1); mbox.get(instr2);  
$display("instr2 = %s", instr2.name()); // instr2 = MUL  
status = mbox.try_get(instr2); // status = 0  
status = mbox.try_put(instr1); // status = 1  
mbox.get(instr2); // instr2 = MUL  
mbox.get(instr2); // blocking - mailbox empty  
mbox.get(pkt); // incorrect type - compiler error
```



9-19

Mailbox shown after compiler error corrected.

## Retrieving Messages from Mailboxes (peek)

```
task peek(ref message); // block if mailbox is empty
function int try_peek(ref message); // non-blocking
// return number of messages in mailbox before try_peek, 0 indicates no messages
```

**Example:**

```
int status;
Packet pkt_obj1 = new();
Packet pkt_obj2 = new(), pkt2drv;
mailbox #(Packet) mbox = new();
mbox.put(pkt_obj1);
status = mbox.try_put(pkt_obj2);           // status = 1
$display("%0d messages", mbox.num());      // 2 messages
mbox.peek(pkt2drv);                      // still 2 messages left
$display(mbox.try_peek(pkt2drv));          // displays 2
$display(mbox.try_get(pkt2drv));           // displays 2
```



9-20

If the type of mailbox is a base class then you can use that mailbox for all derivatives of the base class.

You may also define a typed mailbox type using **typedef**

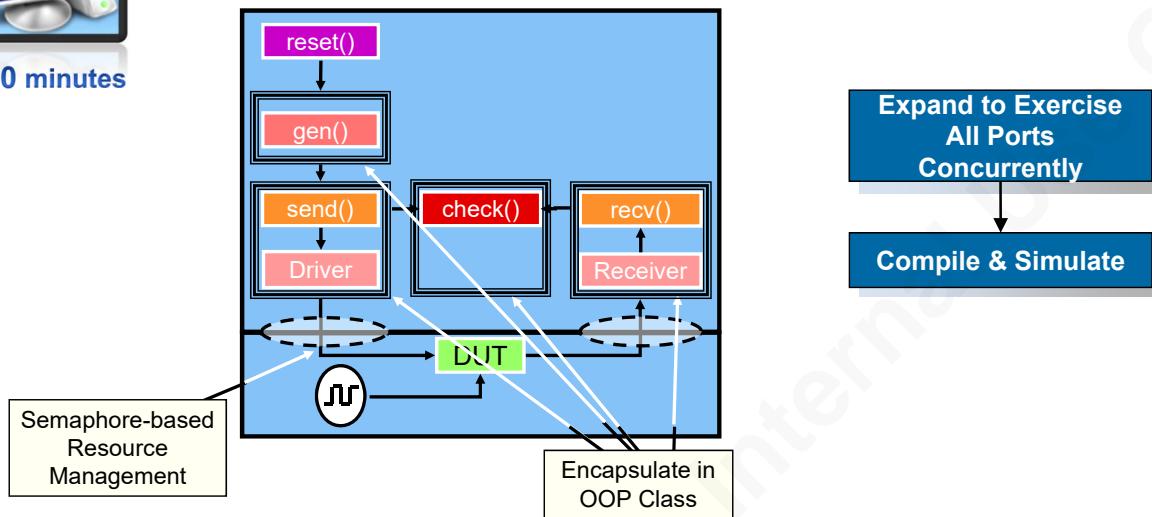
```
typedef mailbox #(Packet) pkt_mbox;
pkt_mbox gen_chan;
```

# Lab 5 Introduction



90 minutes

Expand to broad-spectrum verification



9-21

## Unit Objectives Review

**Having completed this unit, you should be able to:**

- Establish order of execution using events
- Avoid resource collision with Semaphores
- Pass data between threads via Mailbox



9-22

# Agenda

DAY  
3

- |    |  |   |
|----|--|---|
| 8  | Object Oriented Programming (OOP)<br>– Inheritance |  |
| 9  | Inter-Thread Communications                        |  |
| 10 | Functional Coverage                                |  |
| 11 | SystemVerilog UVM Preview                          |  |
| 12 | Customer Support                                   |  |

# Unit Objectives



**After completing this unit, you should be able to:**

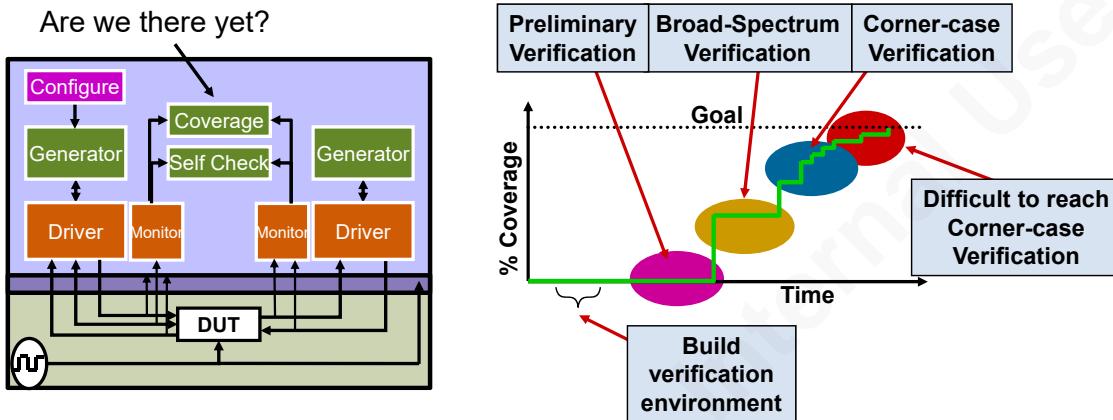
- **Define functional coverage structures**
  - Specify the coverage sample mechanisms
  - Define signals and variables to be sampled
  - Specify expected values that indicate functionality
  - Use parameters to make coverage instances unique
  - Use coverage attributes to customize individual coverage structures
- **Measure coverage dynamically**
- **Generate coverage reports after running VCS**

10-2

# Phases of Functional Verification

## ■ Implement functional coverage to answer

- Has verification goal been reached?
- When to switch to corner-case verification?
- When to write directed tests for difficult to reach corner-case verification?



10-3

The process of reaching the verification goal starts with the definition of the verification goal. What does it mean to be done with testing? Typically, the answer lies in the functional coverage spec within a verification plan. The goal then, is to reach 100% coverage of the defined functional coverage spec in the verification plan.

Once the goal has been defined, the first step in constructing the testbench is to build the verification environment.

To verify that the environment is set up correctly, preliminary verification tests are usually executed to wring out the preliminary RTL and testbench errors.

When the testbench environment is deemed to be stable, broad-spectrum verification based on random stimulus generation is utilized to quickly detect and correct the majority of the bugs in both RTL code and testbench code.

Based on functional coverage analysis, the random-based tests are then constrained to focus on corner-cases not yet reached via broad-spectrum testing.

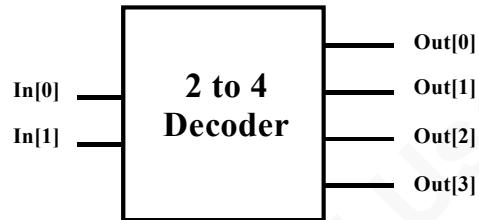
Finally, for the very difficult to reach corner cases, customized directed tests are used to bring the coverage up to 100%.

Verification is complete when you reach 100% coverage as defined in the verification plan.

## Combinational Logic Example

- Goal: Check all combinations of input and output patterns

In[1]	In[0]	Out[3]	Out[2]	Out[1]	Out[0]
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

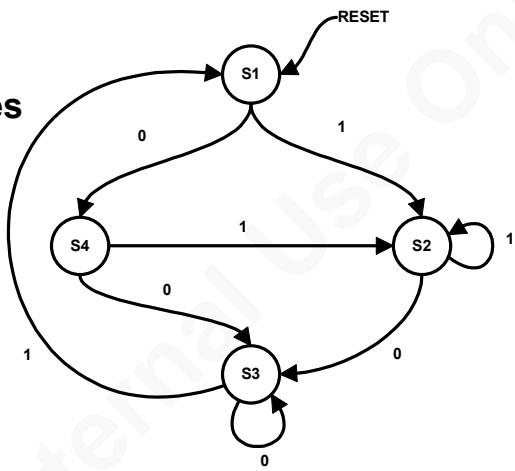


- Create state coverage bins to track input & output bit patterns
- Define sample timing for coverage bins
- Track state coverage bin results

10-4

## State Transition Example

- Goal: Check temporal state transitions
- Create state coverage bins to monitor states
- Create transition coverage bins to monitor state transitions
  - e.g.
  - $S1 \Rightarrow S2 \Rightarrow S3 \Rightarrow S1 \Rightarrow S4 \Rightarrow S2 \Rightarrow S2 \Rightarrow S3 \Rightarrow S1$
- Define sample timing for coverage bins
- Track transition coverage bin results



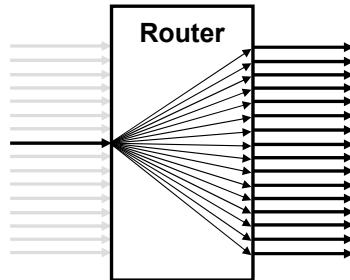
10-5

State machine coverage is built into VCS.

## Cross Correlation Example

- Goal: Check all input ports have driven all output ports

Input port	Output port
0	0
0	1
0	2
...	...
15	14
15	15



- Create cross coverage bins to correlate activities of input ports with respect to output ports
- Define sample timing for coverage bins
- Track cross coverage bin results

10-6

# Functional Coverage in SystemVerilog

- **Create a covergroup**
  - Standalone - Recommended for reusability
  - Inside classes
- **Covergroup encapsulates:**
  - Coverage bins definitions - State, Transition, Cross correlation
  - Coverage bins sample timing definition
  - Coverage attributes e.g. coverage goal
- **Instantiate coverage object(s) from covergroup using new()**
  - Scope of variable visibility is same as subroutines
- **Query coverage object to determine progress of verification**
- **Only 2-state values covered**
  - Sampled values with X or Z excluded from coverage

10-7

# Functional Coverage Example

```
program automatic test;
  covergroup fcov() @(port_event);
    coverpoint sa;
    coverpoint da;
  endgroup: fcov
  bit[3:0]   sa, da;
  event      port_event;
  real       coverage = 0.0;
  fcov port_fc = new();           // Declare & construct coverage object (required)

  initial while (coverage < 99.5) begin
    ...
    sa = pkt_ref.sa;
    da = pkt_ref.da;
    ->port_event;
    // port_fc.sample();          // alternative form of updating of bins
    coverage = $get_coverage();    // overall coverage
    // coverage = port_fc.get_inst_coverage(); // instance coverage
  end
endprogram: test
```

Create coverage group

Declare & construct coverage object (required)

Query coverage result

// alternative form of updating of bins

// overall coverage

// instance coverage

10-8

The covergroup can also be embedded inside a class. The advantage of creating the covergroup inside the class definition is that the covergroup automatically has access to all properties of the class without having an I/O argument. The down side is that, this covergroup can not be reused.

```
class Scoreboard;
  Packet pkt2send, pkt2cmp;  bit[3:0] sa, da;
  covergroup router_cov;
    coverpoint sa; coverpoint da; //class variables visible in covergroup
    cross sa, da;
  endgroup
  function new(...);    router_cov = new(); // must be constructed
  endfunction //instance name same as group
  task check();
    ...
    if (pkt2send.compare(pkt2cmp, message)) begin
      sa = pkt2send.sa;        da = pkt2send.da;
      router_cov.sample(); //update coverage bins
      coverage_result = $get_coverage();
      if ((coverage_result > 99.0) || (...)) ->DONE;
    end
  endtask
endclass
```

## State Bin Creation (Automatic)

- SystemVerilog can automatically create state bins

```
bit [3:0] sa,da;  
covergroup cov1 @(posedge rtr_io.clock);  
    coverpoint sa; // 16 bins  
    coverpoint da; // 16 bins  
    compare: coverpoint (sa > da); // 2 bins  
    concat: coverpoint {sa, da} { // 256 bins  
        option.auto_bin_max = 256; // else 64 bins default  
    }  
endgroup: cov1
```

- Bin name is "auto[value\_range]"

- The value\_range are the value range which triggered that bin

- Maximum number of bins is set by auto\_bin\_max attribute

- Bins are allocated with equal number of states

10-9

# Measuring Coverage

Without auto-binning:

- Coverage is:

$$\frac{\text{# of bins covered (have at least hits)}}{\text{# of total bins}}$$

With auto-binning:

- `auto_bin_max` limits the number of bins used in the coverage calculation

- Coverage is:

$$\frac{\text{# of bins covered (have at least hits)}}{\min(\text{possible values for data type} \mid \text{auto\_bin\_max})}$$

10-10

## Automatic State Bin Creation Example

```
bit[3:0] sa, da;  
covergroup cov_addr @(rtr_io.cb);  
    coverpoint sa;  
    coverpoint da;  
    option.auto_bin_max = 2; // max of 2 auto-created bins  
endgroup: cov_addr  
...  
  
cov_addr cov1 = new();  
  
sa = 1; da = 8;  
@(rtr_io.cb);  
$display("%0d covered", $get_coverage());  
  
sa = 9; da = 9;  
@(rtr_io.cb);  
$display("%0d covered", $get_coverage());  
  
sa = 3; da = 5;  
@(rtr_io.cb);  
$display("%0d covered", $get_coverage());
```

(50% covered) → (75% covered) → (100% covered)

Var	Bin	#Hit
sa	auto[0:7]	1
da	auto[8:15]	1
sa	auto[0:7]	1
da	auto[8:15]	1
sa	auto[0:7]	2
da	auto[8:15]	2

10-11

# State and Transition Bin Creation (User)

- Define state bins using ranges of values
- Define transition bins using state transitions

```
covergroup MyCov() @(cov_event);  
    coverpoint port_number {  
        bins s0 = { 0 };           // one bin for port_number == 0  
        bins lo = { [0:7] };       // creates one state bin for port_number in range 0 to 7  
        bins hi[] = { [8:15] };     // creates 8 state bins hi_8 through hi_f  
        ignore_bins ignore = { 16, 20 }; // ignore if hit  
        illegal_bins bad = default; // terminates simulation if illegal_bins hit  
                                // default refers to undefined values  
        bins t0 = (0 => 8, 9 => 0); // creates one transition bin  
        bins t1[] = (8, [0:7] => [8:15]); // creates 72 transition bins  
        bins other_trans = default sequence; // all other single state transitions  
//      illegal_bins bad_trans = default sequence; // terminates simulation if hit  
    }  
endgroup: MyCov
```

10-12

Bins for default states or transitions are never used in calculating coverage.

The default sequence only counts single state transitions.

# Cross Coverage Bin Creation

- VCS can automatically create cross coverage bins

```
covergroup cov1() @(<u>rtr</u>_io.cb);  
    coverpoint sa;  
    coverpoint da;  
    cross sa, da;  
    cross p_count, mode; //automatic bins for p_count,mode  
endgroup: cov1
```

- Users can also define cross bins to cover (See Note)

```
src: coverpoint sa; //can label coverpoints  
dest: coverpoint da { bins d1 = { 0 }; bins d2 = default; }  
cx: cross src, dest {  
    bins lo = binsof(src) intersect {[0:4]};  
    bins mid = binsof(src) && binsof(dest.d1); //expression  
    bins hi = ! binsof(src) intersect {[0:7]};  
}
```

10-13

VCS will automatically create bins for cross products that do not intersect cross products specified by any user defined cross bin. Bins for default states are never used in cross coverage even if the default state bin is used in a user defined cross coverage bin.

# Wildcard Bins

## ■ Can use wildcards to express bin ranges

- Use **wildcard** keyword in bin definition
- Similar to ==? operator
- ?, x or z may be used as wildcard
  - ◆ ? preferred to avoid confusion when covering 4-state variables

```
covergroup myCov() @(cov_event);
    coverpoint port_number {
        wildcard bins b1 = {4'b01??};      // {[4'b0100:4'b0111]}
        wildcard bins b2[] = {4'b011? => 4'b?000}; // 4 transitions
        wildcard bins w1[2] = {4'b11??}; // w1[4'b1100,4'b1101] and w1[4'b1110,4'b1111]
    }
endgroup: myCov
```

10- 14

The b2 [] bins in the example above cover the following four transitions:

4'b0110 => 4'b0000  
4'b0111 => 4'b0000  
4'b0110 => 4'b1000  
4'b0111 => 4'b1000

VCS extends the LRM to allow wildcards to be used in cross coverage bin.

```
c: cross a, b {
    wildcard bins c1 = binsof(a) intersect {4'b11??};
    wildcard bins c2 = !binsof(b) intersect {4'b1?0?};
}
```

## Bin Coverage – with

- Use **with** to limit values that are included in a bin

- The **with** clause specifies that only those values that satisfy the given expression (for which the expression evaluates to true) are included in the bin
- Expression has access to sampled value via **item**

```
covergroup cg0;
    gfc_cp: coverpoint gfc {
        bins lo[] = {[0:8'h0f]}      with (item % 2 == 0);
        bins med[] = {[8'h10:8'hc0]} with (item % 3 == 0);
    }
endgroup
```

Expression(s) must be true to update bin(s)

Creates 8 bins for lo  
Creates 60 bins for med

10-15

# Specifying Sample Event Timing

```
covergroup definition_name [(argument_list)] [sample_event];  
  [label:] coverpoint coverage_point { ... }  
}
```

- **sample\_event** defaults to with function sample()
  - Can be overridden with one of:
    - ◆ User defined arguments to sample() method
    - ◆ @([specified\_edge] signals | variables)
- **Bins are updated asynchronously as the sample\_event occurs**
  - To update bins at end of simulation time slot
    - ◆ set type\_option.strobe\* = 1 in covergroup

10-16

\* type\_option discussed later in this unit.

## Parameterized Sample Method

- Can override `sample()` with a triggering function `sample()` that accepts arguments

```
covergroup pkt_cg with function sample(Packet pkt);
```

- Allows sampling of coverage data from contexts other than the scope enclosing the covergroup declaration
- Can be called from
  - ◆ Automatic task or function
  - ◆ Sequence or property of a concurrent assertion
  - ◆ Procedural block
- Formal arguments of the sample method should
  - ◆ Only designate a coverpoint or conditional guard expression
  - ◆ Be different from list of covergroup arguments since they belong to the same lexical scope as the formal arguments to the covergroup
  - ◆ Not designate an output direction

10- 17

## Parameterized Sample Method: Example

```
covergroup pkt_cg with function sample(Packet pkt);
    src: coverpoint pkt.sa;
    dst: coverpoint pkt.da;
    cross src, dst;
endgroup: pkt_cg
pkt_cg p_cg = new();

function bit Packet::check();
    if (actual_pkt.compare(ref_pkt, message)) begin
        p_cg.sample(actual_pkt);
    ...
endfunction: check
```

10-18

# Determining Coverage Progress

- `$get_coverage()` returns testbench coverage percentage as a real value

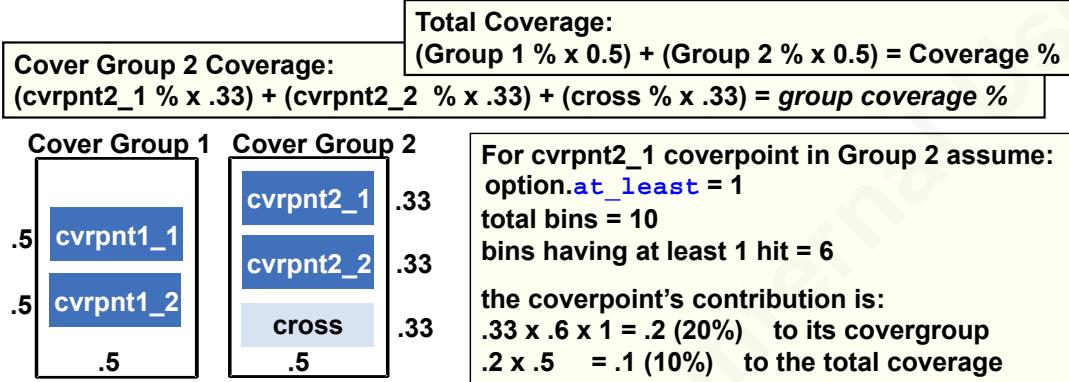
```
covergroup cov1(ref int x, int y) @(rtr_io.cb);
    model: coverpoint x {
        bins S_1[] = { [34:78], [1:27] };
    }
endgroup: cov1

program automatic Main;
    int a;
    real cov_percent;
    cov1 c_obj1 = new(a, b);
    initial forever begin
        cov_percent = $get_coverage();
        $display("%0d covered\n", cov_percent);
        ...
    end
endprogram: Main
```

10-19

## Coverage Measurement Example

- Each covergroup contributes equally to total coverage
- Each coverpoint/cross block contributes equally to the covergroup's coverage
- Attributes may affect contributions



10-20

## Coverage Attributes (1/2)

- Coverage attributes are defined for entire coverage groups or for individual coverpoints
  - Defined using `option` or `type_option` keyword
  - Attributes at the coverage group level may be overridden at the sample and cross level
  - Attributes may be different for each instance of a coverage object by passing arguments

```
covergroup cov1(int var1) @cov_event;
    option.auto_bin_max = var1; // entire group
    coverpoint x {
        option.auto_bin_max = 4; // just for x
        bins lower = { [1:8] };
        ...
    }
endgroup: cov1
```

10-21

## Coverage Attributes (2/2)

### ■ Coverage attributes are of two kinds

- Instance-specific coverage attributes: `option.<attr> = <value>;`
- Type-specific (static) coverage attributes: `type_option.<attr> = <value>;`

### ■ Common attributes available for both kinds - `weight`, `comment`

- `option.<common_attr>`
  - ◆ Will work only if attribute `option.per_instance = 1`
  - ◆ Will save separate instance based coverage
- Use `type_option.<attr>` for cumulative coverage attributes

```
covergroup cov1 @(cov_event);
    coverpoint sa { type_option.weight = 0; }
    coverpoint da { type_option.weight = 0; }
    cross sa, da; //only cross is covered
endgroup: cov1
```

10-22

If the above example were written as

```
covergroup cov1 @(cov_event);
    option.per_instance = 1;
    coverpoint sa { option.weight = 0; }
    coverpoint da { option.weight = 0; }
    cross sa, da;
endgroup: cov1
```

There would be multiple sections in the coverage report. One for the cumulative coverage for all instances of the covergroup and one each for every instance of the covergroup. The bins for cumulative statistics would all have weights equal to 1.

# Major Coverage Attributes

- **at\_least (1):**
  - Minimum hits for a bin to be considered covered
- **auto\_bin\_max (64):**
  - Maximum number automatically created bins
    - ◆ Each bin contains equal number of values
- **comment\*:**
  - A comment that is saved in the database and appears in the report
- **weight (1)\*:**
  - Multiplier for coverage bins
- **per\_instance (0):**
  - Specifies whether to also collect coverage statistics per instance for a coverage group
  - Cumulative statistics always collected

\* also available as a  
type\_option

10-23

# Control and Query of Covergroups

## ■ Methods for control and query of covergroups

- `sample()` only works on a `covergroup` instance
- All others work on `covergroup`, `coverpoint` or `cross`

Method	Function
<code>void sample()</code>	Triggers sampling of the covergroup Only works on covergroup instance
<code>real get_coverage()</code>	Calculates type coverage number (0...100)
<code>real get_inst_coverage()</code>	Calculates the coverage number (0...100) Must set <code>option.per_instance = 1;</code>
<code>void start()</code>	Starts collecting coverage information
<code>void stop()</code>	Stops collecting coverage information

10-24

# Parameterized Coverage Group

```
covergroup MyCov(ref reg[3:0] data, ← must be ref – covered variable
    pass by value ←
    reg reset_1, ←
    input reg[3:0] middle) ← reset_1 – guard variable
    can be ref or pass by value
    coverpoint data { // sampled parameter
        bins s_lo (0:middle) ;
        bins s_hi[] (middle+1:15) iff (reset_1); ← guard expression
    }
endgroup: MyCov
```

■ Variables passed by reference or value  
■ Variables used in guard expressions

```
program automatic test(router.io.TB rtr_io);
    reg [3:0] mydata; reg reset_1;
    MyCov cov1;
    initial begin
        cov1 = new(mydata, reset_1, 7);
        reset_1 = 0; mydata=3;
        cov1.sample(); ← data not sampled when
        #1 reset_1 = 1; cov1.sample(); end ← data sampled this time
    endprogram: Example
```

Bin s\_lo (0:7),  
8 bins s\_hi[8]...s\_hi[15]

data not sampled when  
reset\_1=0

data sampled this time

10-25

Variable *middle* cannot be passed as a **ref** since it is used in the range for coverage state bins. Its value must be known when coverage object is constructed. Just like with arguments to a SystemVerilog task or function, the types and directions of arguments to a coverage group are sticky.

# Coverage Result Reporting Utilities

- VCS writes coverage data to a binary database file
  - The database directory is named simv.vdb
- Generate HTML report:  
`urg -dir <directory>`  
example: `urg -dir simv.vdb`
- Generate Text Report:  
`urg -dir <directory> -format text`  
example: `urg -dir simv.vdb -format text`
- The data in all coverage database files in that directory are merged and reported

10-26

# Sample URG HTML Report

The screenshot displays two main windows from the Synopsys Dashboard:

- Dashboard (Left Window):**
  - Date: Mon Jun 20 16:18:53 2016
  - User: aoha
  - Version: L-2016.06
  - Command line: urg -dir simv.vdb
  - Number of tests: 1

**Total Coverage Summary**

SCORE	LINE	TOGGLE	FSM	GROUP
94.25	94.27	92.89		95.57

**Hierarchical coverage data for top-level instances**

SCORE	LINE	TOGGLE	FSM	NAME
98.66	100.00	97.32		router_test_top
0.00	0.00	0.00		rtslicef

**Total Module Definition Coverage Summary**

SCORE	LINE	TOGGLE	FSM
71.62	59.32	83.92	

**Total Groups Coverage Summary**

SCORE	WEIGHT
95.57	1
- Testbench Group List (Right Window):**

**Total Groups Coverage Summary**

SCORE	WEIGHT
95.57	1

Total groups in report: 1

**Group : router\_test\_top.t::Scoreboard::router\_cov**

NAME	SCORE	WEIGHT	GOAL	AT LEAST	PER INSTANCE	AUTO BIN MAX	PRINT MISSING	COMMENT
router_test_top.t::Scoreboard::router_cov	95.57	1	100	1	0	64	64	

**Summary for Cross router\_cov\_cc**

Variable	sa	da	Cross : router_cov_cc
Samples crossed	sa da		
CATEGORY	EXPECTED UNCOVERED COVERED PERCENT MISSING		
Automatically Generated Cross	256 34 222 86.72 34		

**Automatically Generated Cross Bins for router\_cov\_cc**

Variables	sa	da	COUNT	AT LEAST	NUMBER
[auto[0]] [auto[3]]	0	1	1	1	1
[auto[0]] [auto[6]]	0	1	1	1	1
[auto[0]] [auto[9]]	0	1	1	1	1
[auto[1]] [auto[2]]	0	1	1	1	1
[auto[1]] [auto[4]]	0	1	1	1	1
[auto[1]] [auto[6]]	0	1	1	1	1
[auto[1]] [auto[12]]	0	1	1	1	1
[auto[2]] [auto[4]]	0	1	1	1	1
[auto[2]] [auto[7]]	0	1	1	1	1
[auto[3]] [auto[9]]	0	1	1	1	1
[auto[3]] [auto[11]]	0	1	1	1	1
[auto[4]] [auto[15]]	0	1	1	1	1
[auto[4]] [auto[17]]	0	1	1	1	1
[auto[4]] [auto[7]]	0	1	1	1	1
[auto[5]] [auto[2]]	0	1	1	1	1
[auto[6]] [auto[11]]	0	1	1	1	1

10-27

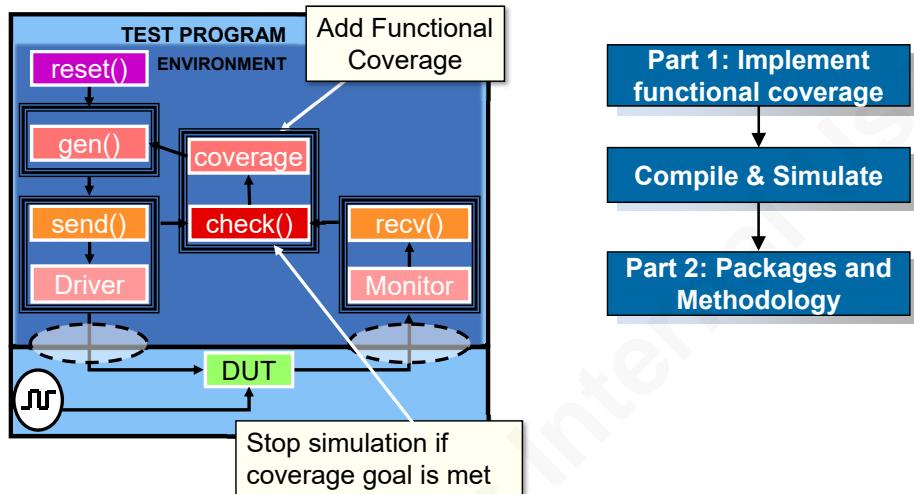
The concept of "shapes" is important to detect the instances of a covergroup that have different parameters passed in to them. The concept of "shapes" was introduced to match identical instances of covergroups so that merging yields useful results. When instances of covergroups are to be merged, a mechanism is needed not only to ensure that the targeted instances are of the same covergroup, but also that the instances have the same parameter(s), since instances of the same covergroup can be instantiated with different parameters.

# Lab 6 Introduction



60 minutes

## Implement Functional Coverage, Packages using an OOP Methodology



10-28

# Unit Objectives Review

**Having completed this unit, you should be able to:**

- **Define functional coverage structures**
  - Specify the coverage sample mechanisms
  - Define signals and variables to be sampled
  - Specify expected values that indicate functionality
  - Use parameters to make coverage instances unique
  - Use coverage attributes to customize individual coverage structures
- **Measure coverage dynamically**
- **Generate coverage reports after running VCS**



10-29

## **Appendix**

**Merging Coverage Results**

**Covergroup Exclusion**

**Test Grading**

**10-30**

## Merging Coverage Results

10-31

# Merging Coverage Results

- **Providing multiple coverage directories to URG creates merged report**

```
% urg -dir test1/simv.vdb -dir test2/simv.vdb -dir ...
```

- Does not create a merged database

- **Create merged database using**

```
% urg -dbname mrgdb -dir test1/simv.vdb -dir test2/simv.vdb
```

- Merged database looks as if a single test produced the results
- Merged report is also generated

- **Recommendation**

- Create separate coverage database for each test
- Only merge coverage databases of tests that passed
- Re-run all tests to collect coverage if design/testbench has changed a lot

10-32

Separate databases are created using the `-cm_dir` argument to simv

```
%simv -cm_dir test1/simv.vdb <other simulation arguments>
```

# Merging Coverage Results – Parallel Merging

## ■ Parallel Merging

- Distributes the merging operations on several machines or in multiple parallel processes on one machine
  - ◆ Can specify a set of machines as a list
  - ◆ Can use LSF or GRID options
- Can provide improvement in performance for merge
  - ◆ e.g. improved the time from 3 hrs to 27 min (6.7x ) using 8 CPUs

```
% urg -parallel \
    -noreport \
    -dir test1/simv.vdb \
    -dir test2/simv.vdb \
    ...
```

Use -noreport to  
merge databases,  
but avoid generating  
URG report

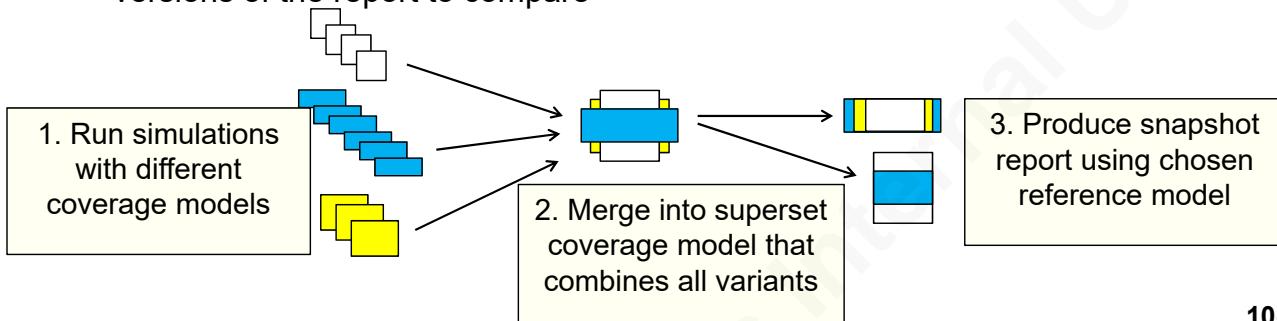
## ■ For help

```
%> urg -parallel -help
```

10-33

# Merging Functional Coverage

- **Functional coverage model often changes during the verification cycle**
  - In inflexible flows, data collected for a given coverpoint, cross or group is discarded when the shape is modified
- **With union merge data for old and new versions is saved**
  - Keep collecting data as coverage model changes
  - Choose which version to use for report generation later, or create multiple versions of the report to compare



10- 34

## Union Merge Example (1/3)

- Coverpoint bins changed

Version 1

```
coverpoint Q {  
    bins x={ [0] };  
    bins y={ [1] };  
};
```

Version 2

```
coverpoint Q {  
    bins x={ [0] };  
    bins z={ [2] };  
};
```

Bin y deleted and  
bin z added

Union merge

```
coverpoint Q {  
    bins x={ [0] };  
    bins y={ [1] };  
    bins z={ [2] };  
};
```

Coverage data is  
preserved for *all* bins

10-35

One simple example of how coverage data is preserved even as the coverage model changes. Here we delete bin y and add bin z in a later version. In the union merge result, the data for both y and z is preserved. A reference merge could be done to show only y, only z, or both y and z's data even though they were never collected together.

## Union Merge Example (2/3)

### ■ Bin redefined

Version 1

```
coverpoint Q {  
    bins x={ [0] };  
    bins y={ [1] };  
};
```

Version 2

```
coverpoint Q {  
    bins x={ [0] };  
    bins y={ [2] };  
};
```

Bin definition has changed

Union merge

```
coverpoint Q {  
    bins x={ [0] };  
    bins y={ [1:2] };  
};
```

- Coverage data is preserved for all values
- Reference model will determine which bins count in final report

10-36

One simple example of how coverage data is preserved even as the coverage model changes. Here, the definition of bin 'y' has changed from value 1 to value 2. In union merge, the coverage data collected for 'y' in version 1 is preserved and merged with the data from Version 2.

## Union Merge Example (3/3)

### ■ Incompatible coverpoint change

Version 1

```
logic [3:0] P;  
logic [3:0] Q;  
covergroup G;  
    coverpoint P;  
    coverpoint Q;  
endgroup
```

Version 2

```
logic [3:0] P;  
logic [7:0] Q;  
covergroup G;  
    coverpoint P;  
    coverpoint Q;  
endgroup
```

Signal size has changed

Union merge

```
covergroup G;  
    coverpoint P;  
    coverpoint Q; (4 bits)  
    coverpoint Q; (8 bits)  
endgroup
```

- Coverage data is preserved for both versions. All data for P is saved from all runs.
- Final reference model determines which data is used for Q.

10-37

One simple example of how coverage data is preserved even as the coverage model changes. Here, the width of Q changed, so we collected some data with 4-bit Q and some with 8-bit Q. Both sets of hits are preserved, and all the data collected for P in both versions is retained.

# Reference Merge

## ■ Generate final report from union merged vdbs

unionMerge.vdb

```
covergroup G;
  coverpoint P;
  coverpoint Q; (4 bits)
  coverpoint Q; (8 bits)
endgroup
```

```
urg -dir filter.vdb unionMerge.vdb
  -flex_merge reference
```

```
logic [3:0] P;
logic [3:0] Q;
covergroup G;
  coverpoint P;
  coverpoint Q;
endgroup
```

hit count  
20  
15  
8

hit count  
20  
15

- Coverage data preserved for all versions. To report on specific version, use **-flex\_merge reference**
  - First test passed to URG is used as "filter"
  - You can create a zero-hits filter vdb with **-reset\_coverage**
- urg -dir version1.vdb  
 -dbname **filter** -reset\_coverage
- If there are multiple tests in the filter, the first one seen determines what is saved
  - Be careful! Different tests in same vdb can have different shapes



10-38

One simple example of how coverage data is preserved even as the coverage model changes. Here, the width of Q changed, so we collected some data with 4-bit Q and some with 8-bit Q. Both sets of hits are preserved, and all the data collected for P in both versions is retained.

# Test Records and Merging

- VCS automatically records a test record for each simulation run
- List using Verdi preference or URG -show tests option

Data from the following tests was used to generate this report

TEST NO	TEST NAME	USER TEST NAME	STATUS	STARTED	FINISHED	SIMULATION TIME	config
T1	simv/mytest0	reset0	pass	Oct 07 04:52:52 PM PDT 15	Oct 07 04:52:53 PM PDT 15	13255000 ms	4core

Test simv/mytest0  
Short name T1  
Simulation time 13255000 ms  
CPU time 0.87 seconds  
Started Oct 07 12:29:46 PM PDT 15  
Finished Oct 07 12:29:48 PM PDT 15  
Peak memory 96080 kb  
Host vgintsb141  
User vernon  
Command simv -cm assert+line+cond+fsm -cm\_name mytest0 -cm\_test reset0 +ntb\_random\_seed=0  
Directory /remote/us01/home25/vernon/cover/jukeboxes/jukeboxgrade

10-39

## Covergroup Exclusion

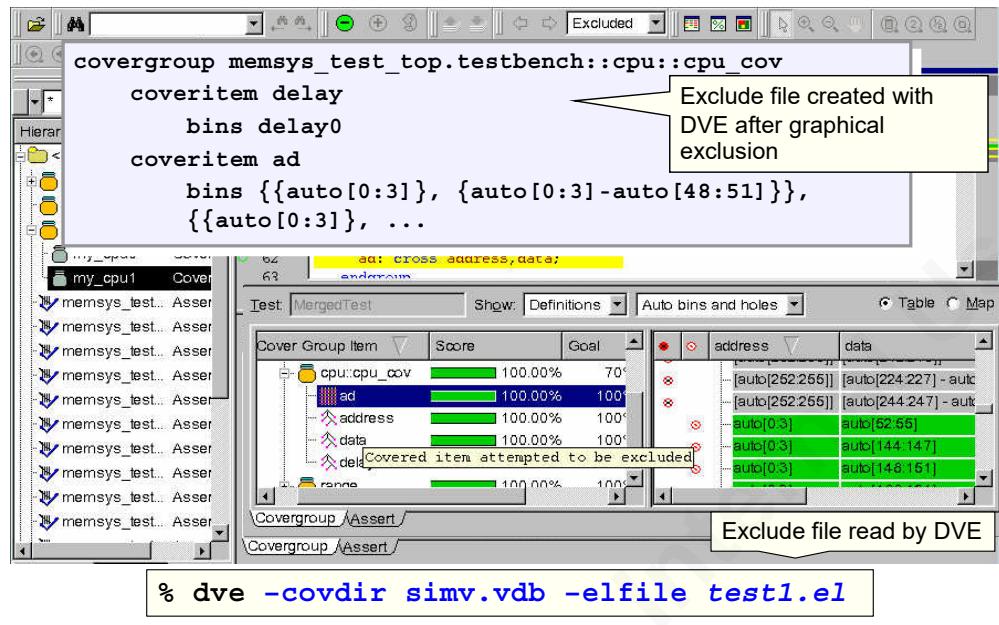
10-40

## Covergroup Exclusion

- Part of common methodology involves excluding various types of coverage (code, assert, covergroup,...) as needed
- You can
  - Exclude don't-care objects
  - Exclude uncoverable objects
  - Exclude covergroups, coverpoints, crosses, and bins
    - ◆ Instance and module level groups
    - ◆ User-defined and automatic bins and bin ranges
- Exclude file for use by URG can be created
  - Interactively using Verdi or DVE
  - Manually by user
- URG prints summary of excluded data

10-41

## Exclusion through DVE



10-42

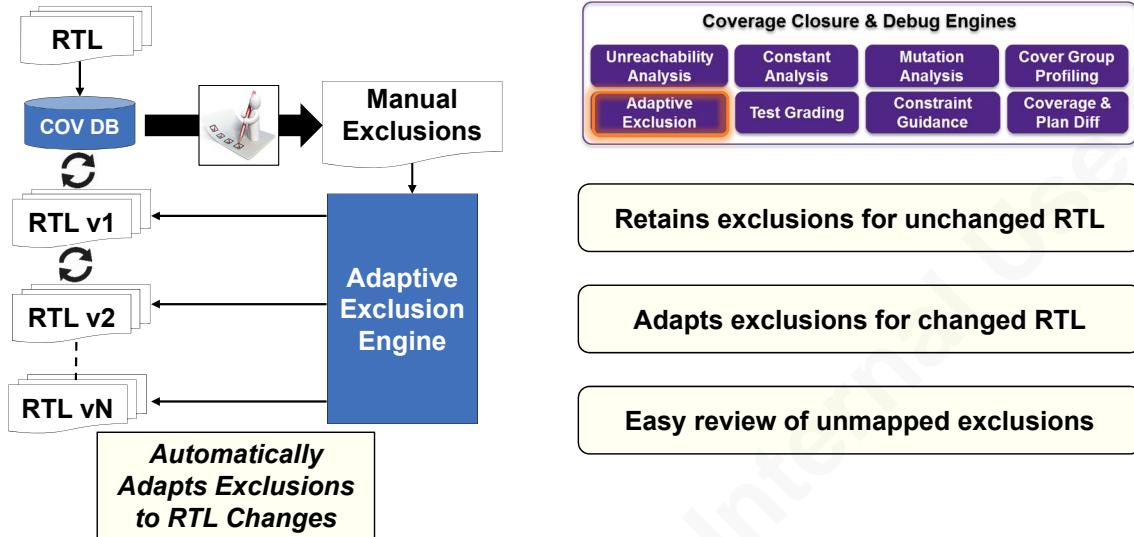
Note: To disallow exclusion of covered objects, use `dve -cov -excl_strict`

Coverage related options to `dve`:

- cov # Start up in coverage mode
- covdir # Open the coverage database in the given directory
- covf # Open coverage directories listed in the given file
- covtests # Open coverage tests listed in the given file
- elfile # Exclusion files to be loaded
- excl\_strict # Do not allow covered objects to be excluded
- covavailabletests # List available tests for the given design

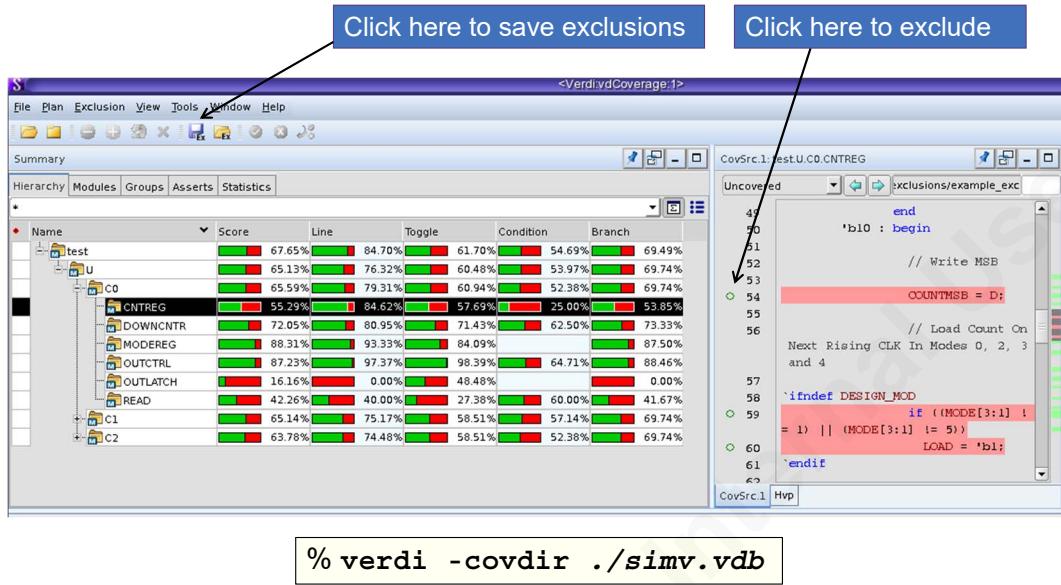
# Adaptive Exclusion

- Eliminate Manual Effort in Exclusion Management for RTL Revisions



10-43

# Verdi Adaptive Exclusions



10-44

# Verdi Coverage Exclusion Manager

- Create Exclusions
- Load Exclusions on Modified Design
- Review Mismatches
- Accept or Reject Exclusions

The screenshot shows the Verdi Coverage Exclusion Manager interface. At the top, there's a summary bar with coverage metrics: Score (69.01%), Line (85.43%), Toggle (61.70%). Below it is a tree view of the hierarchy. The main area is titled "List of All Exclusions". It displays a table with columns: Scope, Object, Details, Annotation, and Actions. The table contains several entries, each with a checkbox for "Excluded" and "Including". Annotations provide details for each exclusion. At the bottom right of the table are "Accept" and "Reject" buttons. Above the table, a toolbar has buttons for "Exclude", "Unexclude", "Excluded scope (instance / covergroup)", "Excluded metric", and "Excluded event". A callout labeled "annotation" points to the annotation column in the table. Another callout labeled "Excluded metric" points to the "Annotation" column for the first entry. A callout labeled "Excluded event" points to the "Annotation" column for the second entry. A callout labeled "Excluded scope (instance / covergroup)" points to the "Annotation" column for the third entry. A callout labeled "Exclude / Unexclude (single click)" points to the "Excluded" checkbox in the table header.

Scope	Object	Details	Annotation
env.dma_controller:data_cg	Cover Group : data	upto1000	unreachable
top dut.i2c_tb_top0.i2c_slave	FSM : state	gma_ack>idle	invalid transition
top dut.i2c_tb_top0.i2c_slave	Line : i2c_slave_model.v...		to be removed
top dut.i2c_tb_top0.i2c_slave	Toggle : debug	1 -> 0	illegal toggle

10-45

## Exclusion through URG

- `% urg -dir simv.vdb -elfile test.el`
  - Reports excluded bins

```
//sample test.el file
covergroup
    test.delay_cov
    coveritem delay
        bins delay0
        bins delay_2
        bins delay_7
        bins delay_8
        bins ignore_vals
        ...
    
```

Summary for Variable delay

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	6	0	6	100.00

User Defined Bins for delay

Uncovered bins

NAME	COUNT	AT LEAST	EXCLUSION
delay0	44	50	1 Excluded

Excluded/Illegal bins

NAME	COUNT
delay_2	0 excluded
delay_7	0 excluded
delay_8	0 excluded
ignore_vals	0 excluded

Covered bins

NAME	COUNT	AT LEAST	EXCLUSION
delay_1	53	50	

10-46

## Test Grading

10-47

# Fast Test Grading

- **Grading measures redundancy/ROI on regression suite**
  - Create short list of best tests for quick regression checks
  - Decide how many seeds to run for different tests
- **Different algorithms**
  - Index-based grading (default) finds the smallest number of tests
  - Cost-based grading selects tests that have the lowest total cost (CPU time, clock time, or cycles)
- **Grade on test vdbs, merged vdbs, or in parallel**
- **Customize which scores and values are shown in report**

10-48

## Cost-Based Grading Example

- **% urg –dir simv.vdb –grade index cost cputime**

Shows the accumulated (Total), standalone (Test), and incremental (Incr) score overall and for each metric  
Tests were selected using CPU time as the cost of each test

SCORE			LINE			GROUP			CPU	Total CPU	Seed 48	Seed	NAME
TOTAL	TEST	INCR	TOTAL	TEST	INCR	TOTAL	TEST	INCR					
39.98	39.98	39.98	79.26	79.26	79.26	0.71	0.71	0.71	0.75	0.75	23	23	simv/test23
43.93	43.89	3.94	85.51	85.51	6.25	2.34	2.28	1.63	1.07	1.82	45	45	simv/test45
45.11	44.56	1.19	86.44	86.17	0.93	3.79	2.95	1.45	1.35	3.17	11	11	simv/test11
45.89	45.01	0.78	86.44	86.17	0.00	5.35	3.84	1.56	1.67	4.84	97	97	simv/test97
46.92	45.49	1.03	86.44	86.30	0.00	7.41	4.68	2.06	2.29	7.13	79	79	simv/test79
47.77	45.20	0.85	86.44	86.17	0.00	9.10	4.40	1.60	2.04	9.17	76	76	simv/test76

10-49

## Test Grading: Additional Options

<b>reqtests &lt;file&gt;</b>	When used with greedy, specifies reading a list of test names from file for inclusion in the grading report. These tests are included at the top of the graded list, regardless of their scores or effectiveness for coverage
<b>cost &lt;costtype&gt;</b>	Put tests in best-first order, with the goal of minimizing the total cost ( <b>cputime</b> , <b>simtime</b> or <b>clocktime</b> )
<b>precision &lt;N&gt;</b>	Show N digits past the decimal point for scores in the grading report.
<b>scores [total+incr+score]</b>	Display the listed score columns. Recommend you omit score, because it is a lot more expensive to compute the standalone score for each test as well as do grading.
<b>columns [col1+col2+...]</b>	Instead of using the default information columns in the grading report, show the specified columns along with the score for each test. The columns supported are: simtime, cputime, clocktime, seed
<b>indexlimit &lt;N&gt;</b>	Track at most N covering tests for each coverable object when using index or cost grading. Higher values will produce more accurate results, but use more memory.

10- 50

# Agenda

DAY  
3

- 8 Object Oriented Programming (OOP)  
– Inheritance**
- 9 Inter-Thread Communications**
- 10 Functional Coverage**
- 11 SystemVerilog UVM Preview**
- 12 Customer Support**

## Unit Objectives



After completing this unit, you should be able to:

- Understand that UVM uses OOP concepts to create a standard library of classes to help you develop your test infrastructure
- Understand that UVM provides standardized mechanisms for messaging, handshaking and synchronizing
- Understand that UVM provides standardized mechanisms for configuring and running your tests

11-2

# UVM – Universal Verification Methodology

- An effort (by an Accellera committee) to define a standard verification methodology & base class library

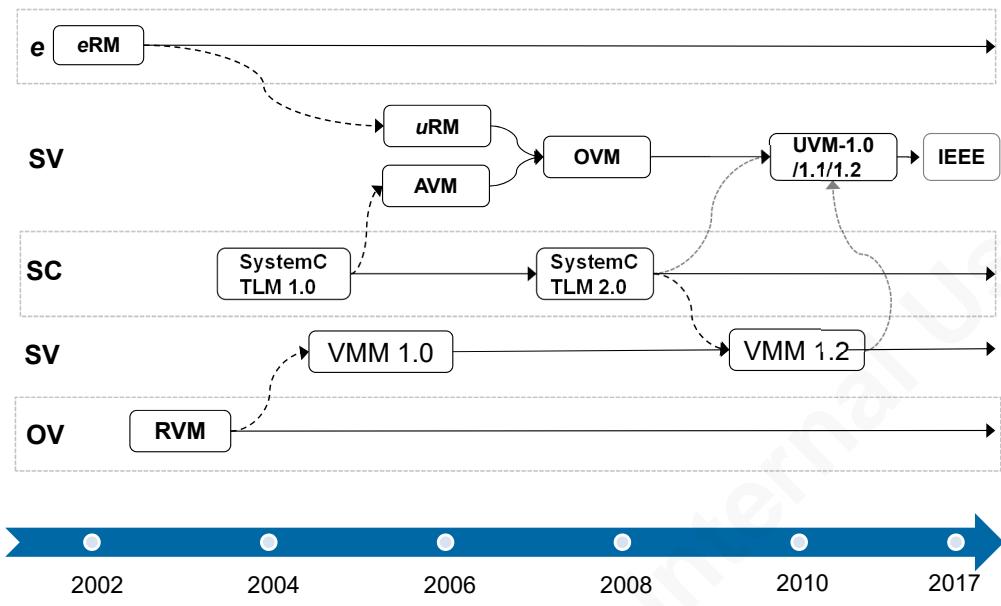
- Uses classes and concepts from VMM, OVM
- Still work in progress
- Published after all vendors' approval

- Related Websites:

- Public Website - <http://www.accellera.org/activities/vip/>
- Member Website - <http://www.accellera.org/apps/org/workgroup/vip/>
- Mantis (Bug Tracking) - [http://eda.org/svdb/view\\_all\\_bug\\_page.php](http://eda.org/svdb/view_all_bug_page.php)
- Sourceforge - <http://uvm.git.sourceforge.net>
- UVM World
  - ◆ <http://www.uvmworld.org/>
  - ◆ <http://www.uvmworld.org/forums/>

11-3

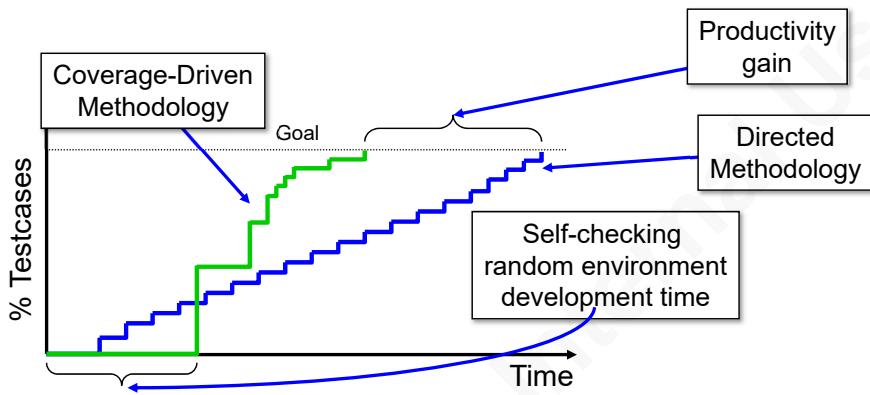
# Origin of UVM



11-4

# Coverage-Driven Verification

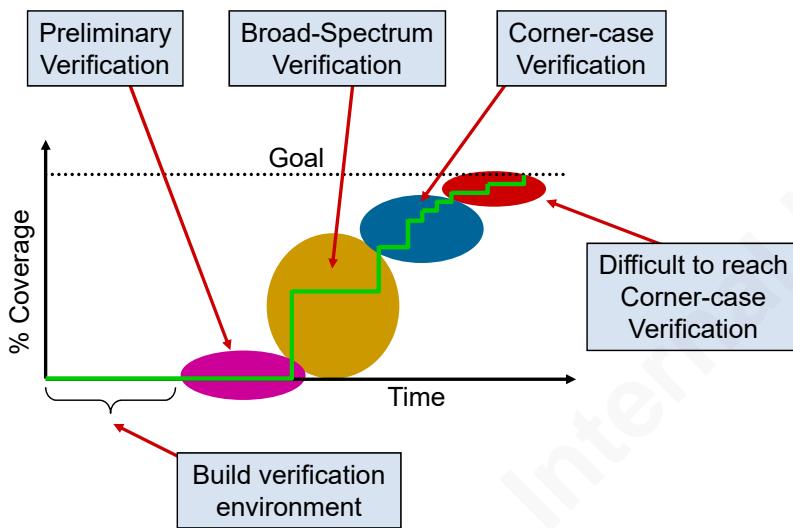
- Focus on uncovered areas
- Trade-off authoring time for run-time
- Progress measured using functional coverage metrics



11-5

# Phases of Verification

**Start with fully random environment. Continue with more and more focused guided tests**



11-6

What does it mean to be done with testing?

Typically, the answer lies in the functional coverage spec within a verification plan.

Once the goal has been defined, the first step in constructing the testbench is to build the verification environment.

In order to verify the environment is set up correctly, preliminary verification tests are executed to wring out basic RTL and testbench errors.

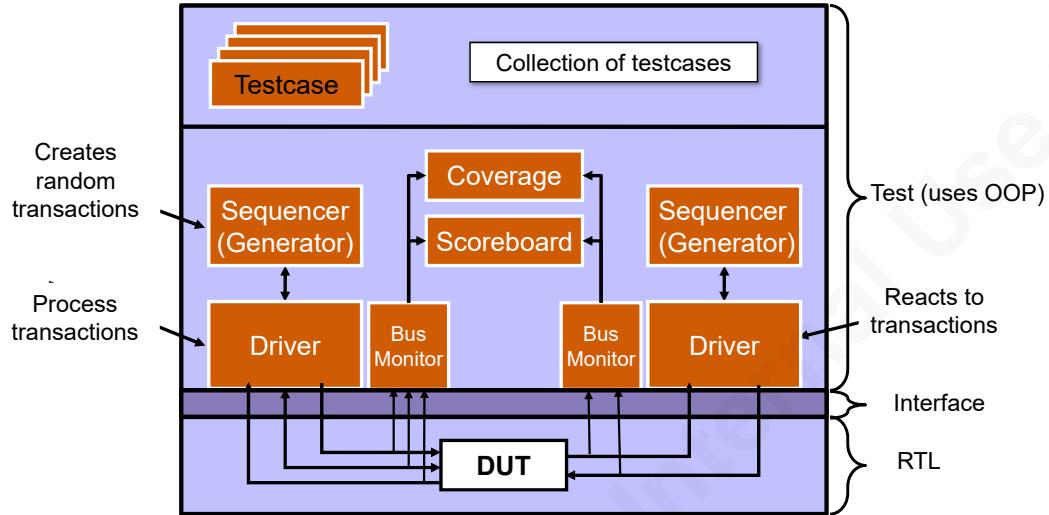
When the testbench environment is deemed to be stable, broad-spectrum verification based on random stimulus generation is utilized to quickly detect and correct the majority of the bugs in both RTL code and testbench code.

During this broad-spectrum testing phase, each simulation run is terminated when a pre-determined goal for that run is reached. Post-simulation, an analysis of the functional coverage result is done. Then, the random stimulus constraints are adjusted to focus on cases not reached during simulation.

Finally, for the very difficult to reach corner cases, customized directed tests are used to close the coverage gap between test runs and the verification plan requirements.

# The Testbench Environment/Architecture

- SystemVerilog testbench structure – Environment uses OOP



11-7

UVM uses the term Sequencer for a transaction Generator.

## Goal - Run More Tests, Write Less Code

- **Environment and component classes rarely change**
  - Sends good transactions as fast as possible
  - Keeps existing tests from breaking
  - Leave "hooks" so test can inject new behavior
    - ◆ Virtual methods, factories, callbacks
- **Test extends testbench classes**
  - Add constraints to reach corner cases
  - Override existing classes for new functionality
  - Inject errors, delays with callbacks
- **Run each test with hundreds of seeds**

11-8

# UVM Guiding Principles

- **Top-down implementation methodology**

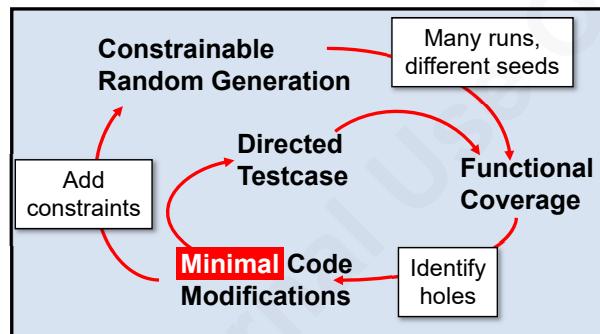
- Emphasizes "Coverage Driven Verification"

- **Maximize design quality**

- More testcases
  - More checks
  - Less code

- **Approaches**

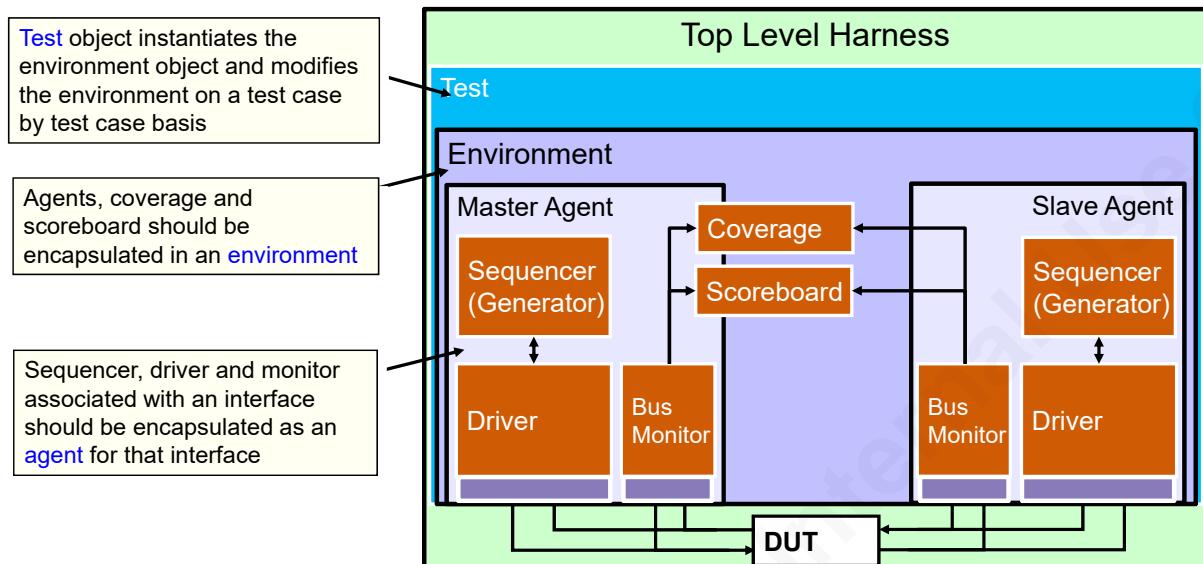
- Reuse across
    - ◆ tests, blocks, systems, projects
  - **One verification environment, many tests**
  - Minimize test-specific code



11-9

# UVM Encourages Encapsulation for Reuse

- Structure should be architected for reuse



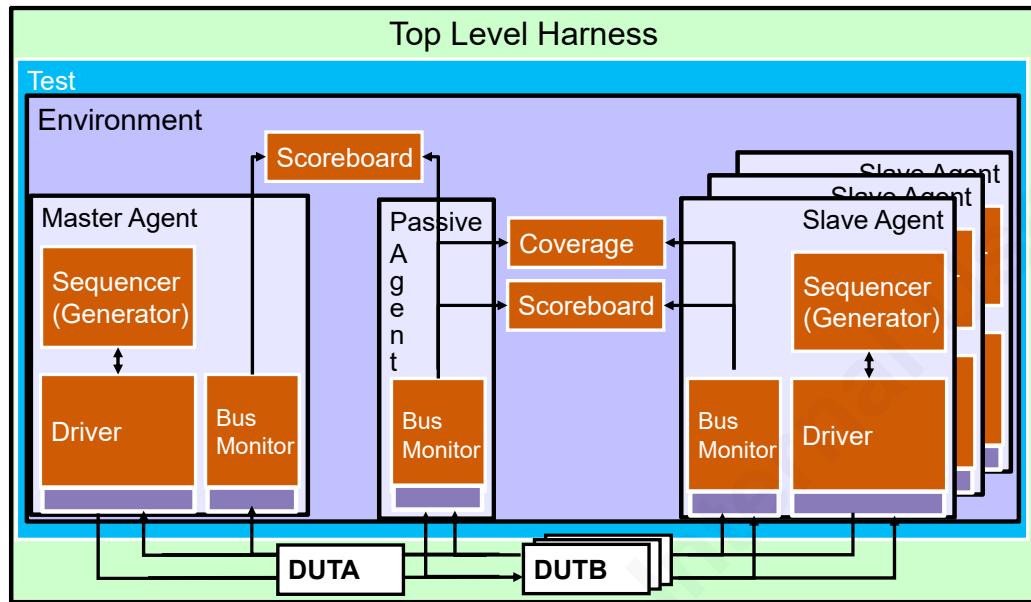
11-10

The protocol-specific blocks (sequencer, driver, monitor) are grouped together to create an agent that can be instantiated as a unit. This reduces the work to connect to a protocol.

The default environment and its components should be written once and, except for correcting bugs or embedding missing features, rarely changed for the rest of the project. If the default environment is constantly changing, many existing tests will break, resulting in bad consequences! The default environment should send good transactions, as fast as possible. Leave "hooks" in the classes so that this behavior can be changed by the testcase, without modifying the original component code.

# UVM Structure is Scalable

- Agents are the building blocks across test/projects



11-11

# Standards: Structural Support in UVM

- Structural/Behavioral base classes

- ◆ `uvm_component`

- `uvm_test`
  - `uvm_env`
  - `uvm_agent`
  - `uvm_sequencer`
  - `uvm_driver`
  - `uvm_monitor`
  - `uvm_scoreboard`
  - `uvm_subscriber`

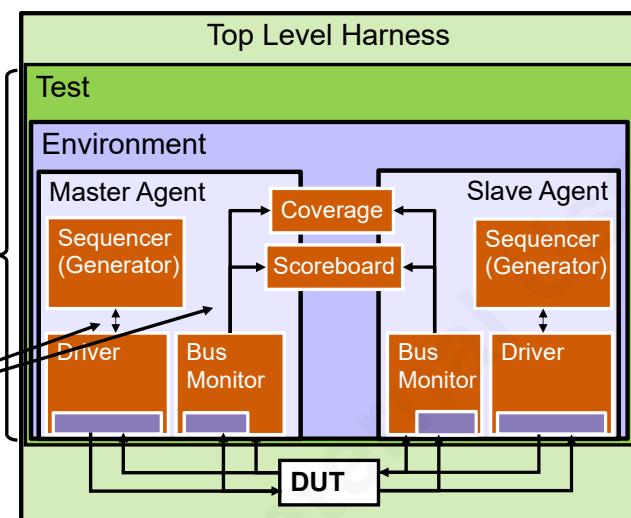
- Communication base classes

- ◆ `uvm_*_port`

- ◆ `uvm_*_socket`

- Base class for data

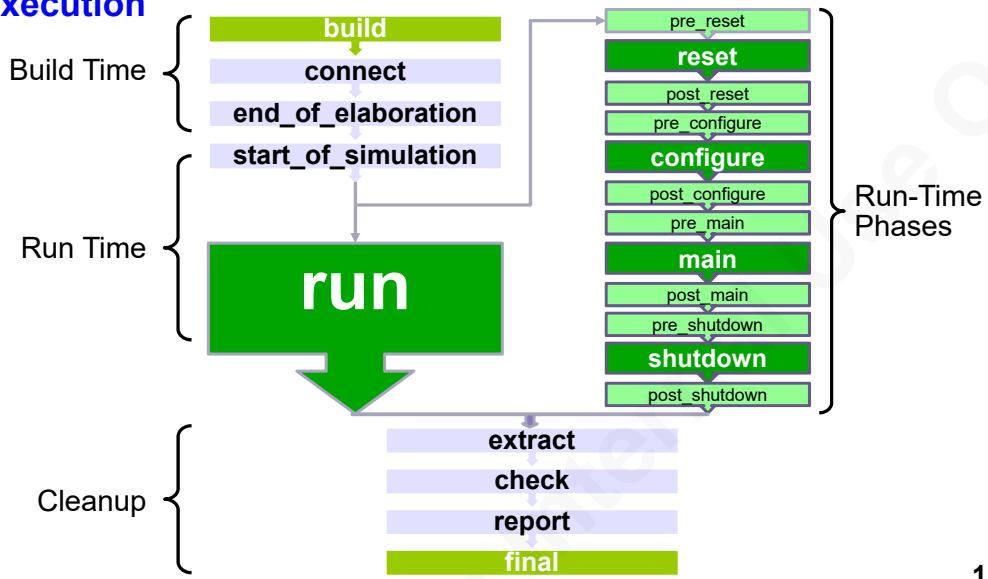
- ◆ `uvm_sequence_item`



11-12

# Standards: Component Phasing

- UVM defines standard phases for component synchronization and automatic execution



11-13

The build phase is called top down so that the higher level components (environments, agents) can decide whether or not to build child components, based on the random configuration.

In the connect phase, the testbench connects TLM ports

In the end\_of\_elaboration phase, check the connections

In the start\_of\_simulation phase, you could print the testbench configuration

In the Run Time task phases, the verification code executes

In the extract phase, data is extracted from the testbench

In the check phase, any final checking is performed

In the report phase, the scoreboard and other checkers report the simulation results

In the final phase, simulation is about to end, print final message

For the Run Time task phases, the run phase should be used to emulate the behavior of the always block in the RTL code. The Run-Time phases (pre\_reset through post\_shutdown) execute concurrently with the run phase and should be used to manage the execution order of sequences.

# Standards: Component Configuration

## ■ Component configuration using a resource database

```
class router_env extends uvm_env;
    // utils macro and constructor not shown
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db #(int)::set(this, ".drv", "port_id", 10);
    endfunction

class driver extends uvm_driver #(packet);
    // constructor not shown
    int port_id = -1; // user configurable
    `uvm_component_utils_begin(driver)
        `uvm_field_int(port_id, UVM_DEFAULT)
    `uvm_component_utils_end

    virtual task build_phase(...); ...
        if (!uvm_config_db #(int)::get(this, "", "port_id", port_id))
            `uvm_info("DRVCFG", {get_full_name(), " using default port_id"}, UVM_MEDIUM);
    ...
    ... For debugging, print full name
```

Set port\_id value in resource database

Retrieve value from resource database

For debugging, print full name

11-14

The resource facility supports both regular expression and glob syntax. Regular expressions are identified as such when they are surrounded by '/' characters. All others are treated as glob expressions.

# Standards: Reporting and Handshaking

## ■ Standard message macros

- Can filter, promote or demote messages as needed on a global or instance basis

```
'uvm_fatal(string ID, string MSG);
'uvm_error(string ID, string MSG);
'uvm_warning(string ID, string MSG);
'uvm_info(string ID, string MSG, verbosity);
```

## ■ Standard handshaking mechanisms

- **uvm\_event** class
  - ◆ Wait for one trigger to releases all waits
- **uvm\_barrier** class
  - ◆ wait for n waiters before opening barrier
- **uvm\_pool** class
  - ◆ common pool of user-defined resources for global lookup

# Standards: Implementing UVM Test

- Test encapsulates verification environment

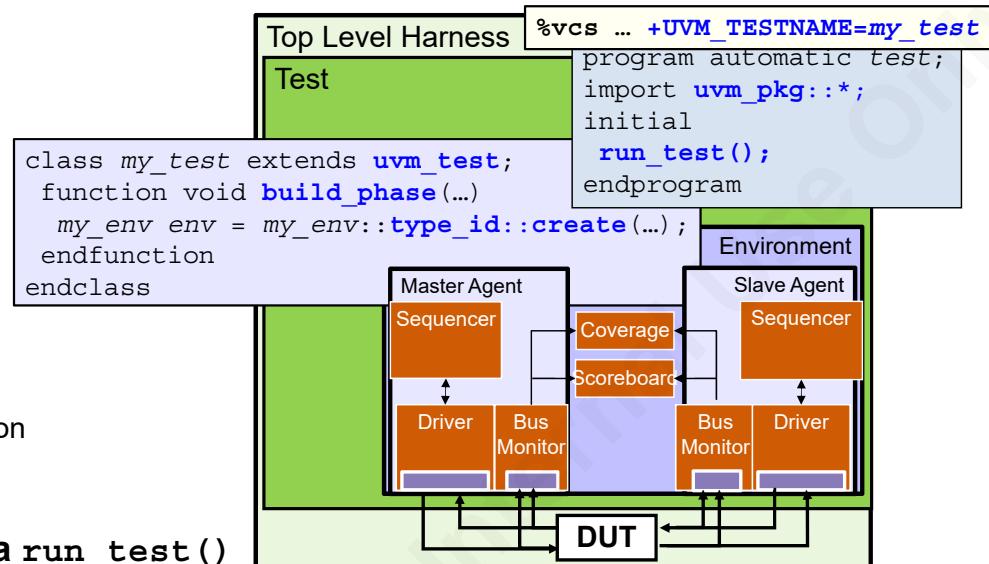
- Test instantiates

- Agents
  - ◆ Sequencers
  - ◆ Drivers/Monitors
- Scoreboards
- Coverage model
- Signal Interfaces

- Test controls

- Configuration
- Start of simulation
- Phases of simulation
- Pass/Fail report
- Factory

- Test executed via `run_test()`



11- 16

## Unit Objectives Review

Having completed this unit, you should be able to:

- Understand that UVM uses OOP concepts to create a standard library of classes to help you develop your test infrastructure
- Understand that UVM provides standardized mechanisms for messaging, handshaking and synchronizing
- Understand that UVM provides standardized mechanisms for configuring and running your tests



11- 17

**That's all Folks!**



**11- 18**

# Customer Support

Synopsys Customer Education Services

© 2019 Synopsys, Inc. All Rights Reserved

# Synopsys Support Resources

## ■ Build a solid foundation:

Hands-on training for Synopsys tools and methodologies

<https://synopsys.com/support/training.html>

- Workshop Schedule and Registration
- Download Labs (SolvNet ID required)

## ■ Drill down to areas of interest:

SolvNet online support

<https://solvnet.synopsys.com>

- Online technical information and access to support resources
- Documentation & Media

## ■ Ask an Expert:

Synopsys Support Center

<https://onlinecase.synopsys.com>

## Training & Education

Hands-on training and education for Synopsys tools and methodologies



Learn from our experts who know Synopsys tools and industry best practices better than anyone else.

- Learn how to get the most out of your Synopsys tools
- Flexible options for learning online or in the classroom
- Tailor the curriculum to meet your requirements

### Ready to get started?

Browse our training and education curriculum by product or service:

#### TRAINING COURSES

eLearning FreeView >	Physical Implementation >	RTL Synthesis >
Sign-Off >	Verification >	FPGA Design >
Software Security & Quality >	Optical Design >	DFM >

<https://training.synopsys.com>

CS - 2

# SolvNet Online Support

- Immediate access to the latest technical information
- Product Update Training
- Methodology Training
- Thousands of expert-authored articles, Q&As, scripts and tool tips
- [Open a Support Center Case](#)
- Release information
- Online documentation
- License keys
- Electronic software downloads
- Synopsys announcements (latest tool, event and product information)

The screenshot shows the SolvNet® support center interface. At the top, there's a navigation bar with links for Documentation, Support, Downloads, Training, Methodology, and My Profile. Below the navigation is a search bar with options for 'SELECT TYPE' (Articles and Documentation) and 'SEARCH' fields for 'AUTOCOMPLETE ON' and 'Search Help' or 'Search IP'. To the right, there are sections for 'SNUG Session Recordings', 'Former Altrex Products', 'Laker-Verdi Support Forum', and 'CODE V, LightTools, RSoft, and LucidShape Users'. A sidebar on the right includes links for 'OPEN A SUPPORT ISSUE', 'GLOBAL SUPPORT CENTERS', 'SYNOPSYS TRAINING', 'SYNOPSYS USERS GROUP', and 'LAKER-VERDI FORUM'. The main content area displays a list of search results, each with a title and a date.

How to Find Scan Data Pins and Export to a Pin Path File (04-17-2017)  
How to Check for Missing Generated Clock Definitions on Divider Circuits (04-17-2017)  
Understanding Timing Exceptions and Masking Behavior (04-17-2017)  
[Short Video] Reverse Debug with Verdi (04-17-2017)  
Migrating From VCS/VCS MX J-2014.12 to VCS/VCS MX K-2015.09 (04-17-2017) *Updated!*  
Analyzing the Performance of SOC (04-16-2017)  
How to Prevent Unwanted Clock Pulses During Transition Delay Testing? (04-12-2017)  
How to Select a Portion of a Pattern Set and Evaluate the New Test Coverage (04-12-2017)  
How Do I Find the Expected Value of a STIL Pattern? (04-12-2017) *Updated!*  
Effects of Display Options on Time-Based Power Analysis Results Due to VCS Delay Options (04-12-2017) *Updated!*  
TetraMAX Does Not Honor False Path in SDC File (04-11-2017)  
Synopsys Test-Case Packager (STP) (04-11-2017) *Updated!*  
Script to Find the Objects Belong to Two Objects (04-11-2017)  
Highlighting Generated Clock Objects That Violate the Nondefault Width (04-11-2017)  
Querying 45-Degree Routes (04-11-2017)  
Specifying a Package to the VCS Command so that the Source Code is Active in Verdi (04-11-2017)  
How Can I Specify Per-Partition scan and clock\_gating Scan-Enable Signals? (04-10-2017)  
Calculating the Total Assert Number for the Different Sections of the Statistics Tab (04-08-2017)  
Using Simultaneous Multivoltage Analysis (SMA) with ECOs (04-07-2017)  
Extremely Small Net Delay Reported for a High Capacitance Net (04-07-2017)  
The Checked Out License Feature for Debugging or Loading AMS Design in Verdi (04-07-2017)  
Disabling and Enabling Messages in the Question Form (04-06-2017)  
The eco\_nellist\_by\_verilog\_file Command Generates a Large Number of Unnecessary Netlist Editing Commands (04-06-2017)

<https://solvnet.synopsys.com>

CS - 3

# SolvNet Registration

1. Go to SolvNet page:
  - <https://solvnet.synopsys.com/>
2. Click on:
  - “Sign Up for an Account”
3. Pick a username and password.
4. You will need your “Site ID”
  - For Information on how to find your Site ID, select the “Synopsys Site ID” link
5. Authorization typically takes just a few minutes.

The figure consists of three screenshots of the Synopsys New User Registration interface. The first screenshot shows the initial registration page with a large red arrow pointing to the 'SIGN UP FOR AN ACCOUNT' button. The second screenshot shows the user input fields for email, username, password, and re-enter password, with a red arrow pointing to the same 'SIGN UP FOR AN ACCOUNT' button. The third screenshot shows the final step where a 'Synopsys Site ID' field is highlighted with a red arrow, indicating it's required for site access.

<https://solvnet.synopsys.com/ProcessRegistration> CS - 4

# Support Center

## ■ Industry seasoned Application Engineers:

- 50% of the support staff has >5 years applied experience
- Many tool specialist AEs with >12 years industry experience
- Engineers located worldwide

## ■ Great wealth of applied knowledge:

- Service >2000 issues per month

## ■ Remote access, and interactive debug, available via WebEx

Contact us:  
Open a support case

SYNOPSYS

PRODUCTS SOLUTIONS SERVICES COMMUNITY ABOUT US SolvNet Global Sites Q

Home / Support / Global Support Centers

### Global Support Centers



Expert Support is Just a Click Away

Synopsys's network of global support resources helps you keep your design on schedule. For fastest support, open your case online where it will immediately be routed to the right expert with the product and design knowledge to expedite resolution. You can also contact local support resources through email or on the phone.

No matter how you let us know about your problem, you can be assured that Synopsys' team of product support experts—including R&D, where necessary—will work with you to find the best possible solution in the shortest possible time to minimize impact on your project and schedule.

**North America**

[Open a case online](#)  
Telephone: 1-800-245-8005  
Hours: 7:00 a.m. - 5:30 p.m. (Pacific Time)

**CODE V Support**

**Europe / Israel**

[Open a case online](#)

**Central Europe**

[Open a case online](#)

**Northern Europe**

[Open a case online](#)

**CUSTOMER SUPPORT**  
CODE V, LightTools, LucidShape & RSoft

**VIRAGE EMLT PRODUCT USERS**  
Use SolvNet To Open Your Cases

**RELEASE NOTIFICATIONS**  
Subscribe for latest information about your products

**UPDATE TRAINING VIDEOS**  
Now Available on SolvNet

**Related items**  
Global Training Catalog  
SolvNet  
Software and Key Download Services

<https://www.synopsys.com/support/global-support-centers.html>

CS - 5

## Other Technical Sources

- **Application Consultants (ACs):**

- Tool and methodology pre-sales support
- Contact your Sales Account Manager for more information

- **Synopsys Professional Services (SPS) Consultants:**

- Available for in-depth, on-site, dedicated, custom consulting
- Contact your Sales Account Manager for more details

- **SNUG (Synopsys Users Group):**

<https://www.synopsys.com/community/snug.html>

CS - 6

# Summary: Getting Support

## ■ Customer Training

<https://www.synopsys.com/support/training.html>

- Register for a Class
- Download Labs

## ■ SolvNet

<https://solvnet.synopsys.com>

- Tool Documentation and Support Articles
- Product Update and Methodology Information / Training
- Open a Support Case (Support Center)

## ■ Other Technical Resources

- Synopsys Users Group (SNUG)
- Application Consultants
- Synopsys Professional Services

CS - 7

This page was intentionally left blank.