



CUSTOMER EDUCATION SERVICES

SystemVerilog Assertions Workshop

Student Guide

50-I-053-SSG-011

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 <https://www.synopsys.com/company/legal/trademarks-brands.html>
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

Document Order Number: 50-I-053-SSG-011
SystemVerilog Assertions Workshop Student Guide

Table of Contents

Unit i: Introductions & Overview

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

Unit 1: Introduction to SVA

Agenda	1-1
Unit Objectives	1-2
What is an Assertion?	1-3
Who Writes Assertions?	1-4
Where is an Assertion Placed?.....	1-5
What does an SV Assertion Look Like?.....	1-6
SV Assertions in Simulation Flow.....	1-7
SV Assertions in Formal Verification Flow	1-8
Additional Resources	1-9
Unit Objectives Review	1-10

Unit 2: Immediate and Concurrent Assertions

Agenda	2-1
Unit Objectives	2-2
Types of Assertions.....	2-3
What is an Immediate Assertion?	2-4
Deferred Immediate Assertion.....	2-5
System Functions in Assertions	2-6
What is a Concurrent Assertion?	2-7
Anatomy of a Concurrent Assertion	2-8
Clocking (when to sample)	2-9
Concurrent Assertion Scheduling	2-10
Action for Success or Failure.....	2-11
Common Assertion Coding Errors.....	2-12
Assertion Severity System Tasks.....	2-13
Property Expressions	2-14
Temporal Delay Operators in Sequences.....	2-15
Delay Examples	2-16

Table of Contents

Sampled Value System Functions	2-17
Sampled Value System Function Examples	2-18
Implication Operator	2-19
Implication Operator Examples	2-20
Disabling Assertion Threads	2-21
Named Property	2-22
not Operator	2-23
and & or Operators.....	2-24
Choosing Property Expression.....	2-25
Embedding Assertions	2-26
Example - Embedding SVA in Separate File	2-27
Property Operators Review	2-28
Control of Assertions	2-29
Control of Assertions: \$assertcontrol (1/2).....	2-30
Control of Assertions: \$assertcontrol (2/2).....	2-31
Example - \$assertcontrol.....	2-32
Runtime Report for Disabled Assertions	2-33
Test for Understanding 1	2-34
Test for Understanding 1: Solution.....	2-35
Test for Understanding 2	2-36
Test for Understanding 2: Solution.....	2-37
Test for Understanding 2: Alternate Solution.....	2-38
Lab 1 Introduction.....	2-39
Unit Objectives Review	2-40
Appendix.....	2-41
VCS Compile Switches for SVA.....	2-42
VCS Runtime Switches for SVA.....	2-43
Appendix.....	2-44
Use of Parameter	2-45
Appendix.....	2-46
Creating Assertion Arrays.....	2-47
Appendix.....	2-48
Assertion Failures	2-49
SVA Debug.....	2-50
Embedded \$display Statements	2-51
Assertions Report.....	2-52
Assertion Debug With DVE	2-53
DVE Debug Flow	2-54
Invoking DVE: Interactive Mode	2-55
Invoking DVE: Post-Processing Mode.....	2-56
DVE Assertion Debug Overview	2-57
Complied Assertions.....	2-58
Assertion Pane	2-59
Attempted Assertions.....	2-60
SVA Source Code	2-61

Table of Contents

SVA Waveforms	2-62
SVA Failure in the Waveform Window	2-63
Assertion Breakpoints – DVE Interactive Mode	2-64
Debug Summary	2-65
Appendix.....	2-66
In-lining SVA in VHDL via pragmas	2-67
SVA Inlined in VHDL Design.....	2-68
SVA Bind with VHDL.....	2-69
SVA in VHDL Compilation Flow	2-70
Appendix.....	2-71
Assertion Performance Mode	2-72

Unit 3: Sequences

Agenda	3-1
Unit Objectives	3-2
Sequences.....	3-3
Sequence Expressions.....	3-4
Named Sequences	3-5
Local Variables in Sequences and Properties	3-6
Boolean Repetition Operator	3-7
Special Case: Repetition with Zero Range	3-8
Boolean Goto Repetition.....	3-9
Boolean Non-Consecutive Repetition.....	3-10
Sequence Repetition Operator	3-11
Repetition Summary	3-12
Sequence Operators: and, or	3-13
Sequence Operators: intersect, within	3-14
intersect Example.....	3-15
within Example	3-16
Sequence Operators: first_match	3-17
first_match Example	3-18
Sequence Operators: throughout.....	3-19
Sequence Operators Summary.....	3-20
Operator Precedence	3-21
Sequences in Assertion Properties.....	3-22
Exercise 1 – Arrival of Requests	3-23
Exercise 2 – Min Arrival Rate of Requests	3-24
Unit Objectives	3-25
Lab 2 Introduction.....	3-26
Appendix.....	3-27
Solution 1 – Arrival of Requests.....	3-28
Solution 2 – Min Arrival Rate of Requests.....	3-29
Appendix.....	3-30

Table of Contents

Sequence Method triggered	3-31
triggered Statement Example.....	3-32
Using triggered to Join Parallel Sequences	3-33
Appendix.....	3-34
Sequences with Multiple Clocks.....	3-35
Sequences with Multiple Clocks.....	3-36
matched Example in Multi-Clock	3-37
Difference Between matched and triggered.....	3-38

Unit 4: SVA Coverage

Agenda	4-1
Unit Objectives	4-2
Assertion Coverage: assert Statement.....	4-3
Default Assertion Coverage.....	4-4
Functional Coverage: cover property.....	4-5
Coding Guidelines: cover property.....	4-6
Adding Coverage Example (1/2)	4-7
Adding Coverage Example (2/2)	4-8
Compile-Time Options	4-9
Run-Time Options	4-10
Unified Report Generator (URG)	4-11
Unit Objectives Review	4-12
Lab 3 Introduction.....	4-13
Appendix.....	4-14
Methodology Recommendations (1/3)	4-15
Methodology Recommendations (2/3)	4-16
Methodology Recommendations (3/3)	4-17
Appendix.....	4-18
Coverage Report: Dashboard View	4-19
Coverage Report: Assertions View.....	4-20

Unit 5: SVA Libraries

Agenda	5-1
Unit Objectives	5-2
What are Assertion Libraries?	5-3
Synopsys Checker Library Benefits	5-4
Checker Library vs. Custom Assertions	5-5
Synopsys SVA Checker Library	5-6
Checker Library Flow	5-7
Checker Binding	5-8
VCS Compilation Switches	5-9

Table of Contents

Global Macros	5-10
Checker Parameters	5-11
OVL-Like Example 1	5-12
OVL-Like Example 2	5-13
Protocol Example 1	5-14
Protocol Example 2	5-15
Protocol Example 3 (1/2)	5-16
Protocol Example 3 (2/2)	5-17
Adding Coverage: VCS Checker Library	5-18
Adding Coverage: Checker Library Levels	5-19
Elaboration checks on SVA Checker Library	5-20
Unit Objectives	5-21
Lab 4 Introduction	5-22
That's all Folks!	5-23

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 intentionally left blank

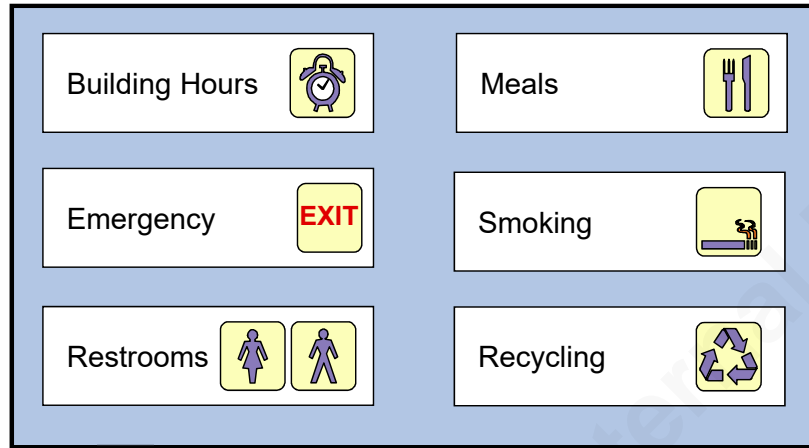
SystemVerilog Assertions

VCS 2018.09

Synopsys Customer Education Services
© 2019 Synopsys, Inc. All Rights Reserved

Synopsys 50-I-053-SSG-011

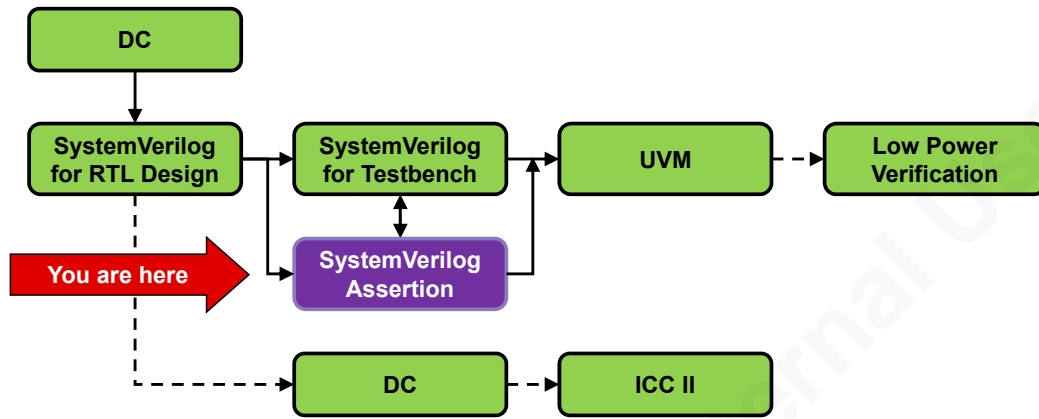
Facilities



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
implementing SVA in RTL or
Testbenches



i-4

Workshop Goals



- ☐ ***Acquire the skills to implement SystemVerilog Assertions (SVA) for verifying VHDL, Verilog or SystemVerilog RTL code in simulation***

i-5

Workshop Prerequisite Knowledge

■ **You must have experience in the following areas:**

- Familiarity with the SystemVerilog language
- Familiarity with a UNIX text editor
- Basic programming skills in Verilog, VHDL or C
- Debugging experience with Verilog, VHDL or C

i-6

Introductions

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

i-7

Agenda

DAY

1

1 Introduction to SVA

2 Immediate and Concurrent Assertions



3 Concurrent Assertions: Sequence



4 SVA Coverage



5 SVA Libraries



i-8

Icons Used in this Workshop



Lab Exercise



Caution



Recommendation



Question



For Further Reference



Exercise



Definition of
Acronyms

i-9

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.

This page was intentionally left blank

Registered For Synopsys Internal Use Only

Agenda

DAY

1

1 Introduction to SVA

2 Immediate and Concurrent Assertions



3 Concurrent Assertions: Sequence



4 SVA Coverage



5 SVA Libraries



Unit Objectives



After completing this unit, you should be able to:

- **Describe what an assertion is**
- **Describe who codes assertions**
- **Describe where assertions can be used**
- **Identify a basic SystemVerilog Assertion**
- **Describe the different flows that use assertions**

1-2

What is an Assertion?

- **An assertion is a statement about a design's intended behavior**

- Captures designer's interpretation of the specification
- Describes **property** of the design
 - ◆ E.g. Output of decoder must have only one bit turned on at any given time

- **An assertion does not contribute in any form to the entity being designed**

- Assertions may be synthesized by some tools to embed hardware monitors in the device

- **An Assertion Captures:**

- Correct / Illegal Behavior
- Assumptions / Constraints
- Coverage Target

- **Examples**

- Combinational
 - ◆ An event should never / always occur
 - ◆ A FSM state encoding is one-hot
 - ◆ A bus has even / odd parity
- Concurrent/Temporal
 - ◆ Event A is always followed by event B
 - ◆ If X is 1, sequence A->B->C never occurs
 - ◆ Interface Specifications/Protocol Checkers

1-3

Fifty years ago Alan Turing made the following observation about partitioning a large verification problem into a set of assertions:

“How can one check a large routine in the sense of making sure that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily flows” 1

In other words an assertion is a statement in a regular language like English about the behavior of any system.

An RTL or Verification assertion is like a continuous monitor on certain signals/variables in the RTL or Testbench. Monitoring presence(occurrence) or absence of certain conditions/values on the signals/variables allows for verification of correct behavior. It is also an alternate description of the behavior which can be used as assumptions or constraints for Formal analysis.

1. As quoted in “Assertion-Based Design 2nd Edition”, Foster et al, Kluwer 2004.

Who Writes Assertions?

■ Design Engineers

- Capture design assumptions (e.g., about interfaces)
- Record design intentions and internal error conditions
- Design behavior (mutex, one_hot, never, always, etc.)

■ Verification Engineers

- Interface, cross-block, higher-level behavior (arbiter, fifo, etc.)

■ IP Providers (PCI / Utopia / SPI-4)

- VCS Checker Library & AIP

1-4

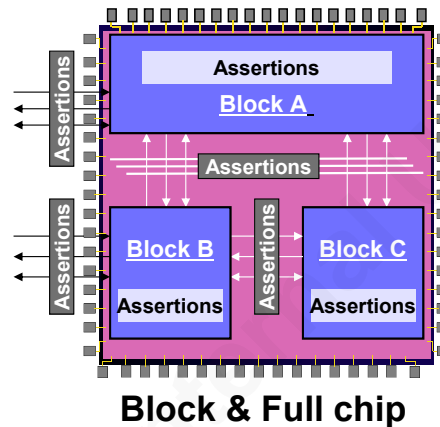
Where is an Assertion Placed?

■ Interfaces

- Between modules and between DUT and Testbench
 - ◆ Check communications
 - ◆ IP protocol adherence
 - ◆ Stimulus constraints

■ Internally

- Inside modules and Testbenches
 - ◆ Assumption checking
 - ◆ Corner-case detection
 - ◆ Design validation

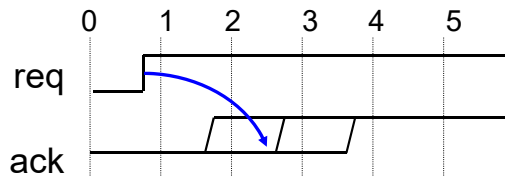


1-5

What does an SV Assertion Look Like?

A concise declarative description of complex temporal behavior:

“After request is asserted, acknowledge must come 1 to 3 cycles later”



```
// SystemVerilog Assertions (SVA) Code for Protocol:  
assert property  
  @(posedge clk) $rose(req) | -> ##[1:3] $rose(ack);
```

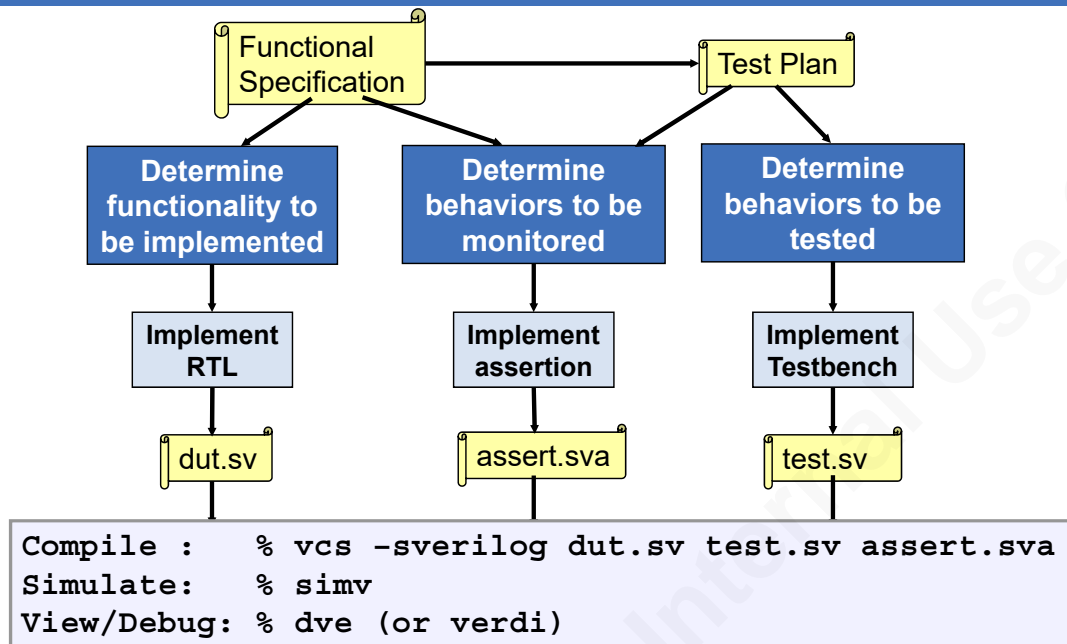
1-6

This type of specification can easily be described in an Assertion Language like SystemVerilog Assertions.

Here this simple assertion declares the active clock, and says that when there is a positive edge on the req signal, then make sure that between 1 and 3 clock cycles later, there is a positive edge on the ack signal.

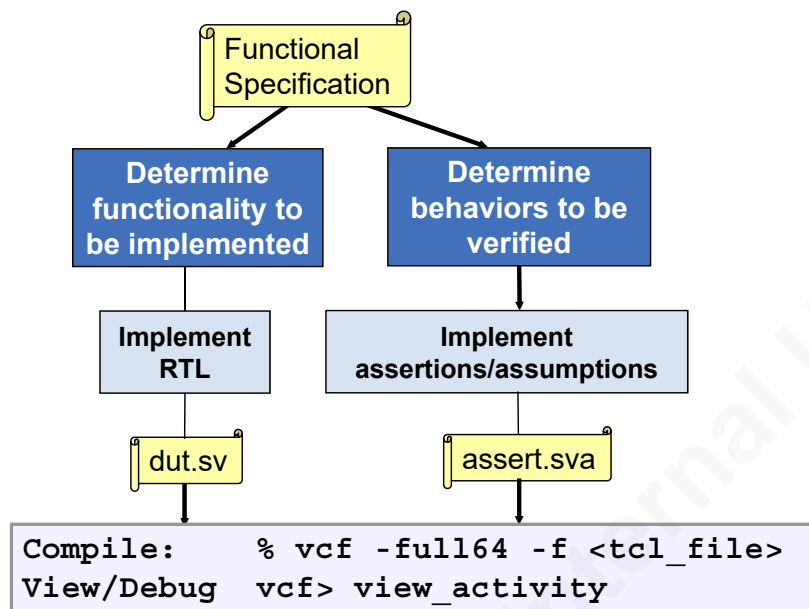
As you can see, this assertion is easily readable to clearly show the intended behavior.

SV Assertions in Simulation Flow



1-7

SV Assertions in Formal Verification Flow



1-8

```

% vcf -full64 -f <tcl_file>

## Parameters [MUST] #####
set_app_var fml_witness_on true
set_app_var fml_mode_on true          # Enable FV feature
set_app_var hierarchy_delimiter "." # delimiter to . from /

## Design #####
read_file -top m -sva -format sverilog -vcs "-sverilog -assert
svaext testRange.v"

## Clock, Reset and constant #####
create_clock clk -period 100
create_reset rst -high
sim_run -stable;      # Run simulation till sequentials get stable
sim_save_reset;      # Save sim state as starting state for formal

## Properties #####
## Run #####
check_fv -block
report_fv -list
  
```

Additional Resources

■ Recommended Reading:

- “Assertion-Based Design” by Harry D. Foster, Adam C. Krolnik, David J. Lacey
- “A Practical Guide for SystemVerilog Assertions” by Srikanth Vijayaraghavan, Meyyappan Ramanathan
- “SVA: The Power of Assertions in SystemVerilog” by Eduard Cerny, Surrendra Dudani, John Havlicek, Dmitry Korchemny
- Verification Methodology Manual for SystemVerilog” by Janick Bergeron, Ed Cerny, Alan Hunter, Andrew Nightingale

■ SNUG papers:

- <http://www.snug-universal.org/papers/papers.htm>

■ Tutorial:

- \$VCS_HOME/doc/examples/assertion/systemverilog



Unit Objectives Review

Having completed this unit, you should be able to:

- Describe what an assertion is
- Describe who codes assertions
- Describe where assertions can be used
- Identify a basic SystemVerilog Assertion
- Describe the different flows that use assertions



1-10

Agenda

DAY

1

1 Introduction to SVA

2 Immediate and Concurrent Assertions



3 Concurrent Assertions: Sequences



4 SVA Coverage



5 SVA Libraries



Unit Objectives



After completing this unit, you should be able to:

- Describe immediate and concurrent assertions
- Describe use of action blocks within assertions
- Describe the scheduling of assertions in SystemVerilog
- Describe property construct and property expressions
- Place assertions in the test environment

2-2

Types of Assertions

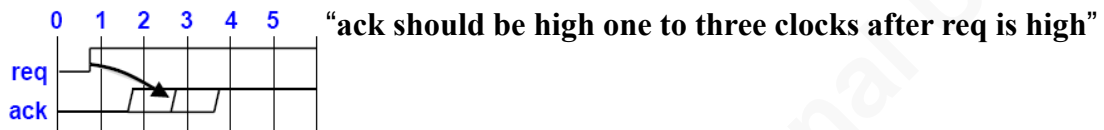
■ Immediate assertions

- Test behavior that does not span time - “state vector should never be 0”

```
assert (state != 0 ) else $error;
```

■ Concurrent Assertions

- Test behavior that can span time (0 to multiple clock cycles)



```
req_cycle : assert property  
  @(posedge clk) req |-> ##[1:3] ack  
else $error; //$error is default severity
```

2-3

What is an Immediate Assertion?

- Immediate assertion checks correctness of state
 - Like an “if” statement
 - Simple immediate assertion embedded within a procedural block
 - Deferred immediate assertion can be outside procedural block
- Immediate assertions cannot do temporal checks
- Example spec: “state vector should never be zero”

```
1 simple_state_not_zero: assert (state != 0)
   else
   3 $error; //
```

1) Label (identifier, recommended)
2) Behavior Specification (assert expression)
3) Action block (optional)

2-4

Syntax: `assert (assert_expression)`
`assert_expression` is any Verilog boolean expression.

Examples:

```
assert((data == 42) || clr)
assert(!(rd && wr))
```

Caution: Avoid the `===` comparator. Formal tools cannot handle this.

Deferred Immediate Assertion

- **Uses #0 or final after the verification directive**
 - Delays reporting of success or failure to end of time-step
 - Can only contain a single subroutine call in action block
 - ◆ Action block executed in: Reactive for #0 and Postponed for final
 - Behaves like `always_comb` outside procedural blocks
- **Used to prevent false alarms at same time-step**
 - Checks assertion each time signal(s) changes but queues result
 - Flushes previous result whenever assertion is triggered again in same time-step
 - Allows all signals in assertions to settle before reporting

```
deferred_state_not_zero: assert #0 (state != 0)
else err_f(); // only one user/system subroutine
```

2-5

To enable deferred immediate assertions with VCS you must use the `-assert_svaext` switch with the compiler and at simulation run.

System Functions in Assertions

Common SystemVerilog system functions can be used in assertions

Function	Returns
<code>\$countones</code>	the number of 1's in a bit vector
<code>\$onehot0</code>	true if at most one bit of the expression is high
<code>\$onehot</code>	true if only one bit of an expression is high
<code>\$isunknown</code>	true if any bit of the expression is 'x'
<code>\$countbits</code>	number of bits that have a specific set of values in a bit vector (e.g. 0, 1, X, Z).

```
state_not_unknown: assert (!$isunknown(state))  
else $error; //
```

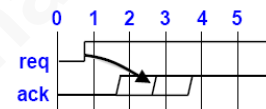
2-6

What is a Concurrent Assertion?

■ Concurrent Assertion checks correct behavior that spans over time

- Keyword `property` distinguishes concurrent from immediate assertions
- When placed outside procedural code
 - ◆ Behaves like “`always`” block that retriggers at each clock
 - ◆ Typically embedded in or bound to DUT
- When placed inside procedural code
 - ◆ Queued each time when assertion line is executed
 - ◆ Each queued assertion checks once at clock

Spec: “ack goes high one to three clocks after req goes high”



```
ack_check: assert property(@(posedge clk) req ##[1:3] ack);
```

2-7

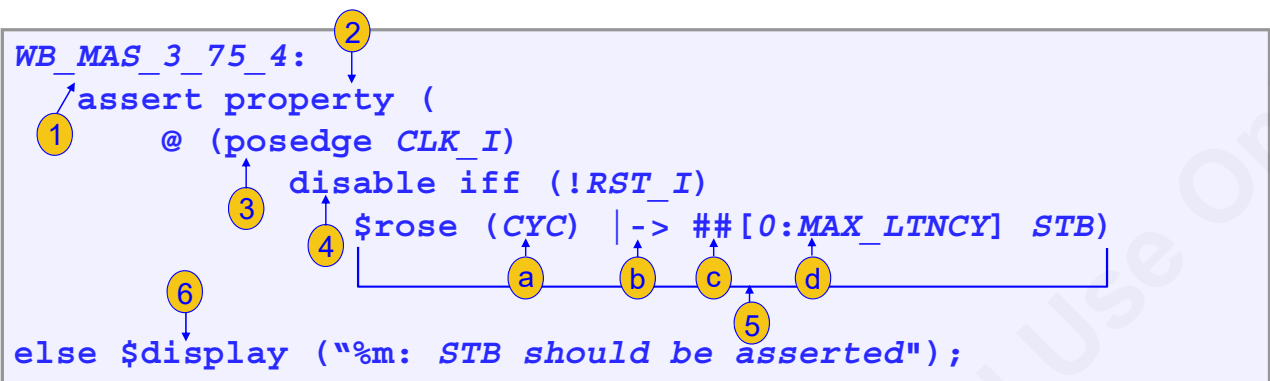
Syntax: `assert property (property_spec) action_block`

`property_spec ::=`

`[clocking_event] [disable iff (expression_or_dist)]`

`property_expr`

Anatomy of a Concurrent Assertion



- | | |
|------------------------------------------|-------------------------------------------------------|
| 1) Label (optional) | 5) Property Expression(behavior specification) |
| 2) Property definition | a) System function (\$rose, \$fell, \$past) |
| 3) Clocking (when to sample) | b) Implication operator (conditional) |
| 4) Asynchronous reset (if needed) | c) Cycle operator (temporal delay) |
| | d) delay range |
| | 6) Action block – optional |

2-8

The optional statement label creates a named block around the assertion statement and can be displayed using the %m format specification. It is highly recommended that you label all assertions.

Clocking (when to sample)

- The clocking expression indicates the sampling event the assertion is evaluated at

- Sampling event can be any simple clock, gated clock or complex expression
- Clocking must be glitch-free for correct behavior
- Sampling event inferred from context if not specified

```
//explicit clock
ack_check: assert property(@(posedge clk) req ##[1:3] ack);

always@(posedge clk) //inferred clock
ack_check : assert property( req ##[1:3] ack ) else $error;
```

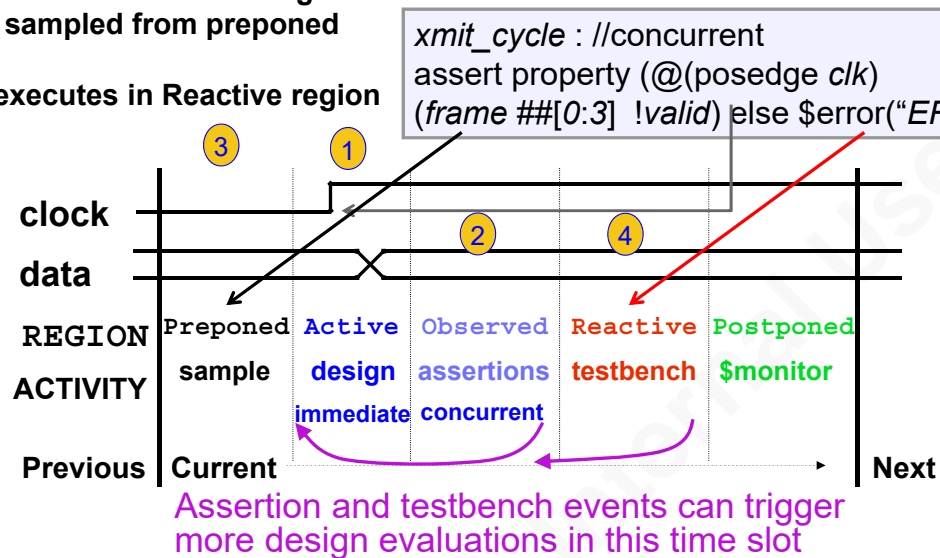
- SystemVerilog has well defined regions for assertions in its event queue

2-9

Clocks can be inferred from the enclosing procedural block.

Concurrent Assertion Scheduling

1. Assertion triggered by clocking event
2. Assertion Executes in Observed region
3. Signal values sampled from preponed region
4. Action block executes in Reactive region



2-10

Each System Verilog time slot 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

Caution: Since action block executes in the reactive region where sampled signals may have changed (due to assignment in active region), use the system function `$sampled(<signal>)` to access the value of `<signal>` in the sampled region.

Action for Success or Failure

```
assert (assert_expression) / property @(posedge clk)
    (<property_expression>) [action_block]
    [else action_block | default: else $error];
```

■ Assertion action blocks

- Specify actions taken upon success/failure of the assertion
 - ◆ Success action block defaults to no-op
 - ◆ Failure action block defaults to `$error`
- May contain any legal SystemVerilog procedural statement
- Execute in Active region for immediate assertions
- Execute in Reactive region for concurrent assertions

```
assert_foo: assert (foo) $display("%m passed");
    else $error("failed"); //
```

Optional message for `$error`
Scope always printed by `$error`

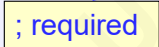
2-11

`%m` prints hierarchical path to assertion instance.


Common Assertion Coding Errors

- A common error made, is using a ; when there is no success action block, but there is a failure action block defined

```
assert_foo: assert property
    @(posedge clk) !(rd_ && wr_) $display("%m passed");
else $error("%m failed");
```



```
assert_foo: assert property
    @(posedge clk) !(rd_ && wr_); //syntax error
else $error("%m failed");
```



2-12

Assertion Severity System Tasks

- **Failure of assertion has severity associated with it**
 - Defaults to `$error` if no failure action block specified
- **Standard SystemVerilog severity system tasks**
 - Embed the following in the failure action block to modify severity level
 - ◆ `$fatal (finish_number, [(message)])` fatal severity – stops simulation
 - ◆ `$error [(message)]` error severity – continue simulation
 - ◆ `$warning [(message)]` warning severity - continue
 - ◆ `$info [(message)]` information severity - continue
 - All severities including `$info` are reported as failures of the assertions
 - All severities print an appropriate default failure message if the optional message is left out

2-13

If an assertion fails and no `else` clause is specified, `$error` will be called by default.

`finish_number` is the level of diagnostic information reported by the tool.

VCS simulation uses this switch to control assertion reporting at runtime

`-assert <keyword_argument>`

The keyword arguments to control failure count are:

`maxfail=N` (requires `-assert enable_hier` at compile)

Limits the number of failures for each assertion to N. When the limit is reached, VCS disables the assertion. You must supply N, otherwise no limit is set.

`finish_maxfail=N` (requires `-assert enable_hier` at compile)

Terminates the simulation if the number of failures for any assertion reaches N. You must supply N, otherwise no limit is set.

`global_finish_maxfail=N` (does not require `-assert enable_hier`)

Stops the simulation when the total number of failures, from all SystemVerilog assertions, reaches N. You must supply N, otherwise no limit is set.

Property Expressions

■ Property expressions consist of

- Sequence expressions which return Match (1) or No Match (0)
 - ◆ Simple Sequence expression: boolean expressions
 - Boolean operators: `&&`, `||`, `!` etc.
 - ◆ Complex Sequence expression: temporal expressions
 - Delay operators: `##n`, `##[m:n]` etc.
 - System Functions: `$rose`, `$fell` etc.
 - System Operators: `and`, `or`, `within` etc.
- Property Operators (operate on results of sequences)
 - ◆ `and`, `or`, `not`, `| ->`, `| =>` etc.

■ Property returns True (1) when the embedded property expression matches simulation behavior otherwise it returns a False (0)

2-14

Temporal Delay Operators in Sequences

■ Fixed Time - `##n`

- `##1` - One clock edge from now

– `assert property @(posedge clock) req ##1 ack);`

■ Time interval (range) - `##[m:n]`

- $n > m$ and $m \geq 0$;
- `##[0:3]` - Any clock edge from now to three clocks later

■ Open ended, eventually - `##[m:$]`

- `##[1:$]` - Between next clock edge and the end of simulation



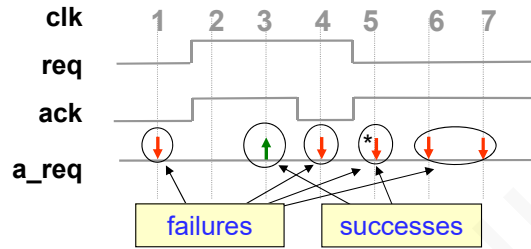
Avoid unbounded properties

Remember: The delay is in clock (sampling) cycles, not timescale delays

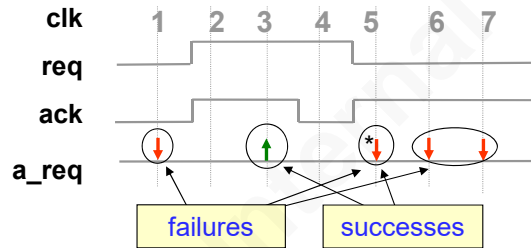
Delay Examples

- **Mismatch causes immediate failure**

```
a_req: assert property
    @(posedge clk)
    req ##1 ack
);
```



```
a_req: assert property
    @(posedge clk)
    req ##[1:3] ack
);
```



2- 16

Sampled Value System Functions

Function	Returns
<code>\$rose (expression [,clocking_evnt])</code> ⚠ only evaluates LSB of expression	true if expression changed to 1
<code>\$fell (expression [,clocking_evnt])</code> ⚠ only evaluates LSB of expression	true if expression changed to 0
<code>\$changed/\$stable (expression [,clocking_evnt])</code>	true if expression did/ <u>did not</u> change since last clocking event
<code>\$past (expression1 [, num_of_ticks] [, expression2] [, clocking_event])</code>	sampled value of expression1 <i>num_of_ticks</i> (default 1) in the past
<code>\$sampled (expression [, clocking_event])</code>	sampled value of expression with respect to last occurrence of <i>clocking_event</i>

clocking_event is optional clock for sampling *expression1* in all
In `$past ()` *expression2* is used to gate *clocking_event*

2-17

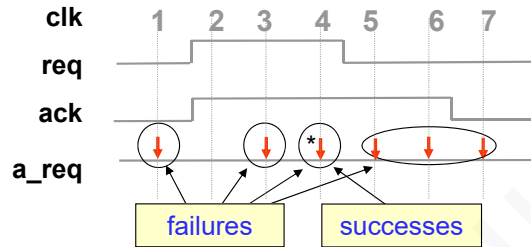


LSB = Least Significant Bit

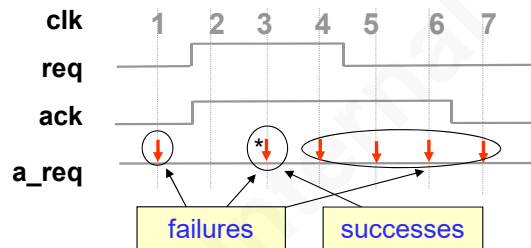
Sampled Value System Function Examples

■ Mismatch causes immediate failure

```
a_req: assert property
  ( @(posedge clk)
    $rose(req) ##2 ack
  );
```



```
a_req: assert property
  ( @(posedge clk)
    $rose(req) ##[1:3] ack
  );
```

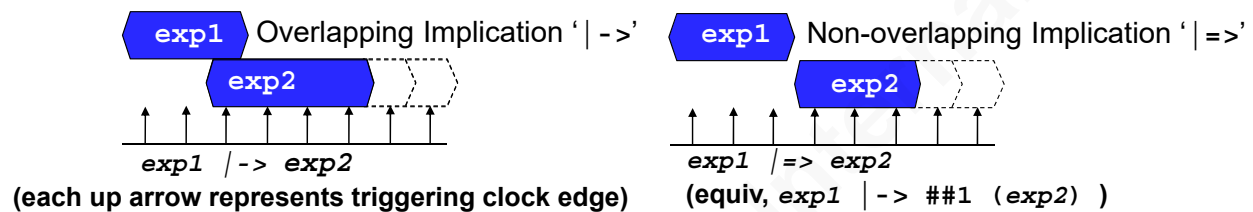


2-18

Implication Operator

■ Implication operators (/ -> or / =>)

- <Antecedent> / -> (or / =>) <Consequent>
 - ◆ Equivalent to if-then structure
- Prevent false failures
 - ◆ Only when Antecedent is **True**, is the Consequent to be **True**
 - ◆ When Antecedent is False, assertion reports **vacuous** success
 - Vacuous successes are filtered out automatically in dve and Verdi³
 - VCS coverage of assertions does include vacuous success



2-19

The implication operator as defined here is usable only in concurrent assertions or property definitions, not in immediate assertions.

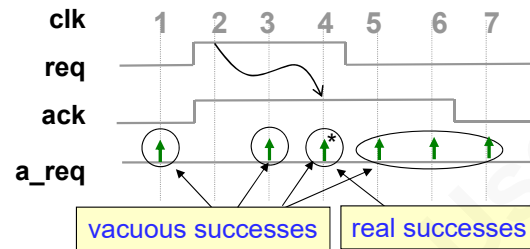
It is recommended that you always use the overlapping implication operator / ->.

dve and Verdi are Synopsys' testbench debugging and visualization tools.

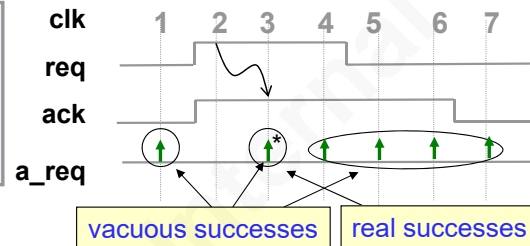
Implication Operator Examples

■ Consequent mismatch causes failure

```
a_req: assert property (
  @(posedge clk)
  $rose(req) |-> ##2 ack
);
```



```
a_req: assert property (
  @(posedge clk)
  $rose(req) |-> ##[1:3] ack
);
```



2-20

It is up to individual tools and applications to filter out vacuous successes. VCS automatically filters out vacuous successes.

Disabling Assertion Threads

- Use `disable iff` to abort property evaluation on a boolean condition (reset expression)
 - If TRUE, terminates the attempt with a vacuous success
- Checks for resets
 - Property aborted as soon as reset expression evaluates to TRUE

```
assert_one_hot: assert property (  
    @(posedge clk)  
    disable iff (reset)  
    $rose(req) |-> ##[1:3] ack  
);
```

2-21

Named Property

■ Property construct

- Reusable, named property with optional formal arguments

```
property <property_name ([<arg_1>,...,<arg_n>])>;  
  @(clocking expression) <property_expression>;  
endproperty
```

- Example

Define Named Property

```
property xmit_protocol(frm_n, vld_n);  
  @(posedge clk) $fell(frm_n) |-> ##[0:3] $fell(vld_n);  
endproperty
```

Assert Named Property

```
assert property (xmit_protocol(frame_n, valid_n)) else  
  $error;
```

2-22

Note: A property expression need not span time. It could be a simple boolean check.

not Operator

- A property is a negation if it is of the form

- `not (property_expression)`

- ◆ Returns true if the expression evaluates to false and vice versa

- With implication, use `not` only in the consequent

e.g. assert that if a occurs, then it is not followed by b

- If the implication is included in the negation then each vacuous success (when a is not true) results in failure of the property expression

```
property p1; @(posedge clk)
    a |-> not (##1 b);
endproperty
```

CORRECT

```
property p1; @(posedge clk)
    not(a |-> ##1 b);
endproperty
```

INCORRECT

2-23

Using `not` over the complete property expression leads to failures since an implication reports vacuous successes when the antecedent is false. This is not the assertion intended. There is no concept of vacuous failure. `not` is most often used over sequences (discussed in another unit).

and & or Operators

■ Two properties can be logically ORed

- `property_exp1 or property_exp2`
 - ◆ Returns true if and only if at least one of `property_exp1` and `property_exp2` evaluates to true.

■ Two properties can be logically ANDed

- `property_exp1 and property_exp2`
 - ◆ Returns true if and only if both `property_exp1` and `property_exp2` evaluate to true.

2-24

Choosing Property Expression

- A discriminant may be used to select a particular property expression

```
if (<expression>) property_exp1 ;  
[else property_exp2;]
```

- If boolean expression *expression* is true then evaluate *property_exp1* else evaluate *property_exp2*
 - ♦ the *else* clause is optional
- If boolean expression *expression* is false and the *else* is missing property evaluation returns a vacuous success

```
if (<expression1>) property_exp1 ;  
[else if (<expression2>) property_exp2;]  
[else property_exp3;]
```

2-25

Embedding Assertions

■ Embed assertions inline with DUT code or test code

■ Embed assertions in a separate file

- Encapsulate assertions in a `module`, connect to target module using a `bind`
 - ◆ Input ports needed for all signals used in the properties
 - ◆ Internal signals of target module can be used as well
- `bind` checker module to:
 - ◆ All instances of a target module
 - ◆ A specific instance of a target module
 - ◆ Connect ports with named or positional style

No changes to
design code needed
with separate file

```
// syntax of bind command
bind <module_name/instance_name> <SVA_module_name>
    #<parameters> <bind_instance_name>
    (<port_list>); //named or positional connect
```

2-26

You may use macros or VCS pragmas to control compile

```
`ifdef ASSERT_FOO //using macros
assert_foo: assert property(foo)
    $display("%m passed");
    else $fatal(1, "%m failed");
`endif
```

Using VCS pragmas:

```
/* sv_pragma */ - multiline assertion code pragma
// sv_pragma - single line assertion pragma
--sva - VHDL assertion pragma

/* sv_pragma
    assert_foo: assert (@(posedge clk) prop_foo)
    $display("%m passed");
    else $fatal(1, "%m failed");
*/
```

Requires compile-time switch

```
Verilog: -sv_pragma
VHDL: -sva
```

Example - Embedding SVA in Separate File

```
// Verilog module DESIGN
module DESIGN (clk,a,b,c,out,...);
input clk, a, b, c;
output out;
reg d_reg;
...
always @( a or b or c)
...
endmodule: DESIGN
```

```
// SV module checker
module checker #(max_lat = 1)
(input clk, aa, bb, cc, dd);
...
c1: assert property ( @ (posedge clk)
( bb ##[1:max_lat] aa ) |-> dd ##max_lat cc );
...
endmodule : checker
```

```
module my_bindings;
// bind to module DESIGN -> connected to all instances
bind DESIGN checker #(.max_lat(1)) my_checker_1 (
.clk(clk) .aa(a), .bb(b), .cc(c), .dd(d_reg));
...
endmodule: my_bindings
```

Separate module recommended for bindings

2-27

Property Operators Review

Available property operators (in order of precedence)

Operators	Description
<code>not</code>	Property negation - <code>not (property_exp)</code>
<code>and</code>	Logical AND of two properties - <code>property_exp1 and property_exp2</code>
<code>or</code>	Logical OR of two properties - <code>property_exp1 or property_exp2</code>
<code>if...else</code>	<code>if (<expression>) property_exp1; [else property_exp2;]</code>
<code> -></code>	Overlapping implication
<code> =></code>	Non-overlapping implication

 Shaded operators can also operate on instances of named properties

2-28

Control of Assertions

■ Compile time

- `-assert disable_assert`

■ Assertion control system tasks

- `$assertoff [args] ;`
 - ◆ Stops and resets assertion monitoring
 - ◆ Active assertions not affected
- `$asserton [args] ;`
 - ◆ Enable assertions previously turned off
- `$assertkill [args] ;`
 - ◆ Abort all active assertion checks
 - ◆ Future assertion monitoring not affected

`args := (levels[,list_of_modules_or_assertions])`

2-29

Assertion control tasks take two optional arguments

- levels (indicates levels of hierarchy)
- list_of_modules_or_assertions (indicates which modules, scopes or assertions to control)

Control of Assertions: \$assertcontrol (1/2)

- **\$assertcontrol** offers a comprehensive and fine-grained control functionality
 - Ability to **enable**, **disable** or **kill** assertions based on their type or directive type
 - Locking feature to assist in overriding control
 - Applications
 - ◆ Fine grained control helps in debugging during development phases of a design project
 - ◆ Applying **lock** to assertions contained in a power domain while controlling other assertions in the system
- **Syntax:**

```
$assertcontrol ( control_type [, [ assertion_type ]  
[, [ directive_type ] [, [ levels ]  
[, list_of_scopes_or_assertions ] ] ] )
```

2-30

\$assertcontrol subsumes functionality provided by all other assertion control tasks.

In many situations \$assertcontrol simplifies manual grouping of assertions either by name or type.

Control of Assertions: \$assertcontrol (2/2)

- Valid values for `control_type`, `assertion_type`, `directive_type` are:

control_type			assertion_type			
Value	Control	Description	Value	Assertion Type	Value	Directive type
1	Lock	Prohibit any control changes	1	Concurrent	1	Assert
2	Unlock	Allow Control changes	2	Simple Immediate	2	Cover
3	On	Enable	4	Observed Deferred	4	Assume
4	Off	Disable	8	Final Deferred		
5	Kill	Kill assertions	16	Expect		
6	PassOn	Enable execution of success action blocks	32	Unique		
7	PassOff	Disable execution of success action blocks	64	Unique0		
8	FailOn	Enable execution of fail action blocks	128	Priority		
9	FailOff	Disable execution of fail action blocks				
10	NonVacuousOn	Enable execution of action blocks on non-vacuous success				
11	VacuousOff	Disable execution of action blocks on vacuous success				

2-31

Current Limitations - as of VCS 2018.09. Later releases may have support.

`control_type Kill` is not supported for `Unique` and `Priority` Assertion Types
`assertion_type 16` (`Expect` Statement) is unsupported (No Effect)

The control type values from 6 to 11 (`PassOn`, `PassOff`, `FailOn`, `FailOff`, `NonvacuousOn`, `VacuousOff`) are not supported

Example - \$assertcontrol

```
// Verilog module DESIGN
module DESIGN (clk,a,b,c,rst_n,out,...);
input clk, a, b, c, rst_n;
output out;
reg d_reg; event rst_event
...
always @( a or b or c)
...
initial begin @(negedge rst_n)
@(posedge rst_n) ->rst_event
endmodule: DESIGN
```

```
// SV module checker
module checker #(max_lat = 1)
(input clk, aa, bb, cc, dd);
...
c1: assert property ( @ (posedge clk)
( bb ##[1:max_lat] aa ) |-> dd ##max_lat cc );
...
endmodule
```

```
module my_bindings;
// bind to module DESIGN -> connected to all instances
bind DESIGN checker #(.max_lat(1)) my_checker_1 (
.clk(clk) .aa(a), .bb(b), .cc(c), .dd(d_reg));
endmodule
```

Use **let** to write cleaner code.
See Notes for definition of **let**

```
module assert_control;
let OFF=4; let ON=3; let CONCRNT=1; let ASSERT=1; let ALL_LVL=0;
initial begin: assertion_control_block
//Turn OFF all assertions in DUT for all levels of hierarchy
$assertcontrol(OFF,CONCRNT,ASSERT,ALL_LVL,TEST_TOP.DUT);
//Turn ON specific checker at relevant event
@(HARNESS.DUT.rst_event); //generated in DESIGN module for example
$assertcontrol(ON,CONCRNT,ASSERT,ALL_LVL,TEST_TOP.DUT.my_checker1.c1);
end: assertion_control_block //can also fork this block
endmodule: assert_control
```

Assume *DESIGN* instance
DUT in *TEST_TOP* module.
Can control specific module
instances and checkers for
various types and levels

2-32

A **let** declaration defines a template expression (a **let** body), customized by its ports. A **let** construct may be instantiated in other expressions.

let declarations can be used for customization and can replace the text macros in many cases. The **let** construct is safer because it has a local scope, while the scope of compiler directives is global within the compilation unit. Including **let** declarations in packages is a natural way to implement a well-structured customization for the design code.

You can also OR assertion types and directive types. For example

```
let CONCRNT = 1;
let SIMPL_IMMEDIATE = 2;
let CSI = (CONCRNT | SIMPL_IMMEDIATE); //Concurrent and Simple Immediate
let KILL = 5; let COVER = 2; let ALL_LVL=0;
$assertcontrol(KILL,CSI,COVER,ALL_LVL,TEST_TOP.DUT);
```

Caution: Control of complete interface instances is not yet supported in VCS (as of vcs2018.09). Specific assertions inside interface instances can be controlled.

Runtime Report for Disabled Assertions

■ Report using runtime switch

`-assert report[=<file>]`

- `.disablelog` appended to the filename
- Default filename `assert.report.disablelog` dumped in the current directory

■ Report lists all assertions disabled via

- System tasks `$asserton/off/kill`
- `-assert hier` at compile and runtime

■ Report will be categorized by

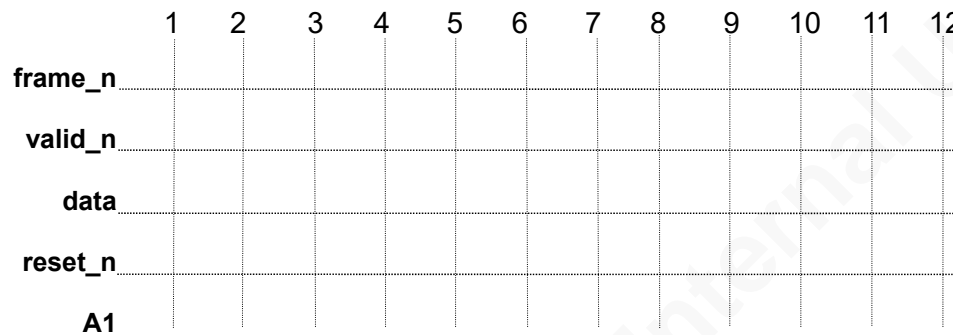
- Disabled module assertions (compile time)
- Assertions disabled via `-assert hier` switch
- Assertions still disabled at end of simulation

2-33

Test for Understanding 1

■ Draw one successful match waveforms for

- A1: `assert property(@(posedge clock) disable iff(!reset_n) $fell(frame_n) |-> ##4 (data && valid_n) ##1 (data && valid_n) ##1 (data && valid_n));`
- Assume `reset_n` is high throughout , `frame_n` falls at clock 1



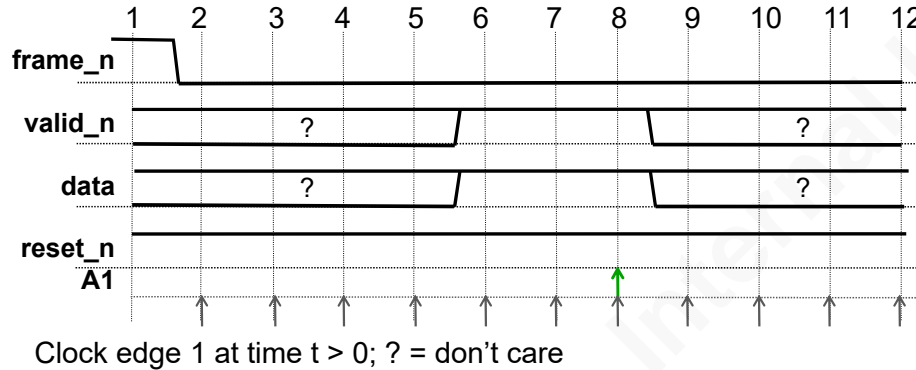
Clock edge 1 at time $t > 0$

2-34

Test for Understanding 1: Solution

■ Draw one successful match waveform for

- A1: `assert property(@(posedge clock) disable iff(!reset_n) $fell(frame_n) |-> ##4 (data && valid_n) ##1 (data && valid_n) ##1 (data && valid_n));`
- Assume `reset_n` high throughout, `frame_n` falls at clock 1

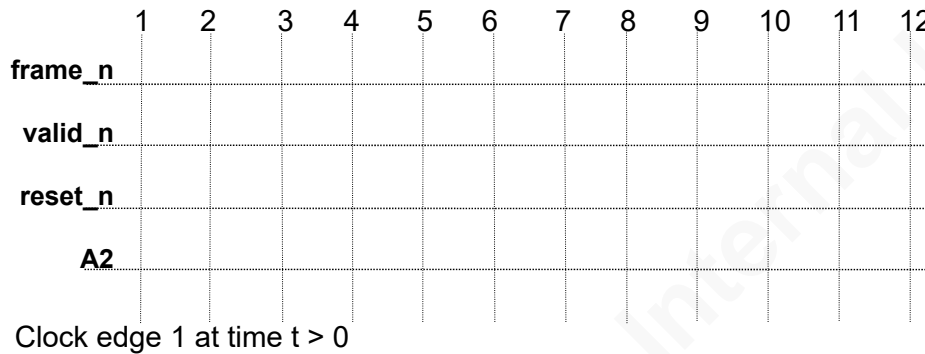


2-35

Test for Understanding 2

■ Draw one successful match waveform for

- A2: `assert property(@(posedge clock)
disable iff(!reset_n) $fell(frame_n) | -> ##[2:9]
($rose(frame_n) && !valid_n));`
- Assume `reset_n` is high throughout and `valid_n` falls one clock after `frame_n` falls (at clock 1)

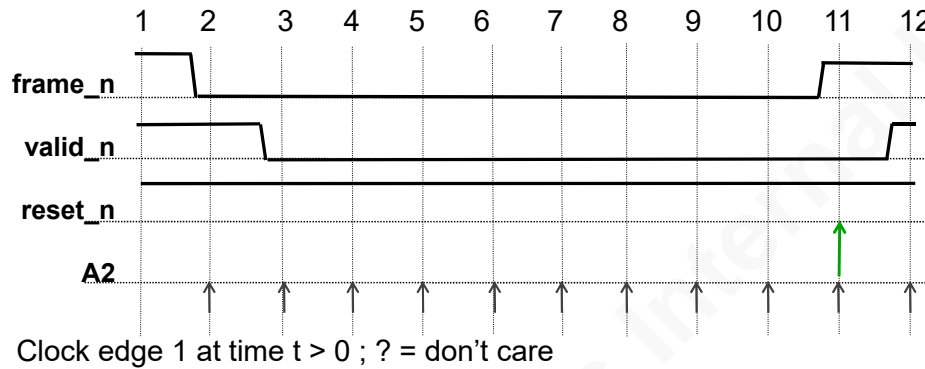


2-36

Test for Understanding 2: Solution

■ Draw one successful match waveform for

- A2: assert property(@(posedge clock)
disable iff(!reset_n) \$fell(frame_n) | -> ##[2:9]
(\$rose(frame_n) && !valid_n));
- Assume reset_n is high throughout and valid_n falls one clock after frame_n falls (at clock 1).

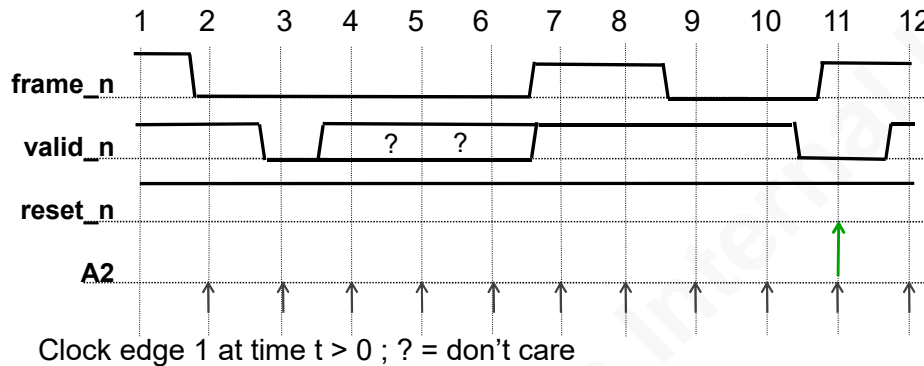


2-37

Test for Understanding 2: Alternate Solution

■ Draw one successful match waveform for

- **A2:** `assert property(@(posedge clock)
disable iff(!reset_n) $fell(frame_n) | -> ##[2:9]
($rose(frame_n) && !valid_n));`
- Assume `reset_n` is high throughout and `valid_n` falls one clock after `frame_n` falls (at clock 1)



2-38

Note that the property only checks for the state of `valid_n` each time `frame_n` rises within 2 to 9 clocks after `frame_n` falls and stops checking after the first success. If the condition is not true at any edge during the 2(clock# 4) to 9(clock# 11) clocks, the assertion fails at the ninth clock edge after `frame_n` falls. The consequent expression check on clock# 7 fails because `valid_n` is not low, but we still have 4 more clocks to check for the condition. Also note that in this solution, another assertion check starts at clock# 9 which also succeeds at clock# 11.

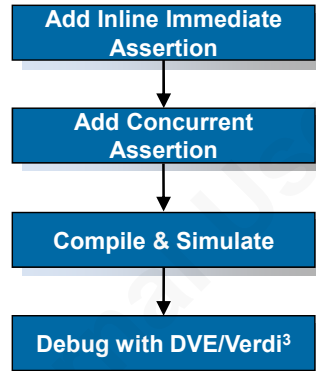
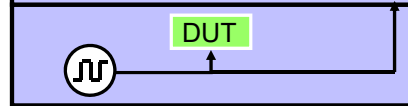
Lab 1 Introduction



45 minutes

Implement Immediate and Concurrent Assertions

```
program automatic test(...);
initial begin
  $vcdpluson;
  $display("Hello");
  reset();
end
task reset();
...
endtask
endprogram
```



2-39

Unit Objectives Review

Having completed this unit, you should be able to:

- Describe immediate and concurrent assertions
- Describe use of action blocks within assertions
- Describe the scheduling of assertions in SystemVerilog
- Describe property construct and property expressions
- Place assertions in the test environment



2-40

Appendix

VCS Compile and Runtime Switches for SVA

Using parameters in SVA

Creating Assertion Arrays

SVA Debugging

Using SVA in VHDL and Mixed-Languages

Improving Assertion Performance

2-41

VCS Compile Switches for SVA

Compile-time Switches	Description
<code>-sverilog</code>	activate SV(A) compilation
<code>-sv_pragma</code>	enable in-line SVA (VCS only)
important <code>-assert</code> options	
<code>filter_past</code>	ignore <code>\$past</code> when buffer is empty
<code>enable_diag</code>	control result reporting at runtime
<code>dve</code>	enables viewing assertions in dve
<code>disable</code>	turns off assertion/cover/assume
<code>disable_assert</code>	turns off assertion/assume
<code>disable_cover</code>	turns off cover properties
<code>disable_file = filelst</code>	disable SVA checkers listed in filelst
<code>dumpoff</code>	disable dumping SVA info into VPD file
<code>hier = file</code>	file to enable and disable assertions (See Note)
<code>svaext</code>	Enable IEEE 1800-2009 and later features

2-42

`-assert hier = file` is also required at runtime.
It also needs the `-enable_hier` option at compile time.

VCS Runtime Switches for SVA

important <code>-assert</code> runtime options	
<code>maxfail=N</code>	stop reporting after N failures
<code>maxsuccess=N</code>	stop reporting after N successes
<code>maxcover=N</code>	stop cover after N matches - cover properties
<code>report [=file]</code>	generates assertion report in filename
<code>quiet</code>	disable display of assertion failure messages
<code>success</code>	display assertion success messages also
<code>dumpoff</code>	disable dumping SVA info into VPD file
<code>nopostproc</code>	disable detailed coverage messages
<code>finish_maxfail=N</code>	stop after N failures for any assertion
<code>svaext</code>	Enable IEEE-2009 and later features
<code>global_finish_maxfail=N</code>	stop after N total failures for all assertions

2-43

Report generated by the runtime time switch `-assert report=file` will be categorized based on

- Module assertions disabled at compile
- Assertions disabled via `-assert hier` switch
- Assertions disabled at end-of-simulation

assertion options that are enabled when the `-assert enable_diag` elaboration option is used:

`-assert success`, `-assert summary`, `-assert maxcover=N`,
`-assert maxsuccess=N`

assertion options that are enabled when `-assert enable_hier` elaboration option is used:

`-assert hier`, `-assert maxfail=N`, `-assert finish_maxfail=N`

assertion options that do not require the `-assert enable_diag` or `-assert enable_hier` option:

`-assert dumpoff`, `-assert nocovdb`, `-assert nopostproc`,
`-assert quiet`, `-assert quiet1`, `-assert no_fatal_action`,
`-assert report`, `-assert vacuous`,
`-assert global_finish_maxfail=N`

Check VCS User-guide on solvnet for more details on these options and switches.

Appendix

VCS Compile and Runtime Switches for SVA

Using parameters in SVA

Creating Assertion Arrays

SVA Debugging

Using SVA in VHDL and Mixed-Languages

Improving Assertion Performance

2-44

Use of Parameter

- Since SVA is part of SystemVerilog language, design parameters can be passed to SVA

```
module check_bus (req, ack, clk);  
  
    input clk, req, ack;  
    parameter DELAY = 10;  
  
    property p1;  
        @(posedge clk) req |-> ##DELAY ack;  
    endproperty  
  
    a1: assert property(p1);  
  
endmodule
```

2-45

For this example, a parameter is defined that will allow the user to specify a custom message when the assertion fires. A default message is provided, but can be passed as a user defined parameter during the binding step. If multiple parameters are to be passed during binding, they are separated by commas.

```
module check_par #(parameter string s="default_msg 1",  
    parameter z=0, parameter string y="default_msg 2") (...  
  
bind data_bus check_par #("Msg1", 5, "Msg2") u1 (...  
  
bind data_bus check_par #("Msg1", , "Msg2") u2 (...
```

Appendix

VCS Compile and Runtime Switches for SVA

Using parameters in SVA

Creating Assertion Arrays

SVA Debugging

Using SVA in VHDL and Mixed-Languages

Improving Assertion Performance

2-46

Creating Assertion Arrays

■ Use **generate** statement to create arrays of properties

```
module assert_dma_in (clk, dma_req_i, dma_ack_o);
parameter ch_count = 1; ————— Default creates 1 assertion
input clk;
input [ch_count-1:0] dma_req_i, dma_ack_o;
// Define assertions for each channel
for (genvar k = 0; k < ch_count; k = k+1) begin: DMA_Assumptions
    genvar assert_dma_req:
    (special variable) assert property (@(posedge clk) $fell(dma_req_i[k]) | ->
                                $past(dma_ack_o[k]));
                                required label
end: DMA_Asumptions //for loop
endmodule: assert_dma_in
                                override default parameter
                                instance name
```

Creates 31 separate assertions

```
bind top assert_dma_in #(31) block1 (clk_i, dma_req_i, dma_ack_o);
```

2-47

Appendix

VCS Compile and Runtime Switches for SVA

Using parameters in SVA

Creating Assertion Arrays

SVA Debugging

Using SVA in VHDL and Mixed-Languages

Improving Assertion Performance

2-48

Assertion Failures

■ Assertion failures occur when:

- Assertion code is wrong
- Assertion description itself is incomplete
- Input sequence may not be constrained properly
- Bug in the design

■ Debugging failures poses challenges

- Multiple attempts and threads of assertions
- Correlate large amounts of related data
- Isolating the cause or contributing logic

2-49

SVA Debug

■ Textual Reporting

- Embedded `$display()`;
- `-assert report` and `-assert success`
 - ◆ Report which assertions failed or succeeded through text report

■ Visual Debugging

- DVE (Discovery Visual Environment) or Verdi³
 - ◆ Assertions waveform with intermediate evaluations
 - ◆ GUI supports debugging facilities like source debug, list, trace attempt and assertions summary

2-50

Embedded \$display Statements

```
$display with sequences
A1:assert property @(negedge clk)
    x |-> ##1 (~TRUE, $display ("1: ", $time))
        ##[3:5] (y, $display ("2: ", $time))
        ##[4:7] (z, $display ("3: ", $time))
Bounding true expression
endproperty: A1
```

Add a \$time

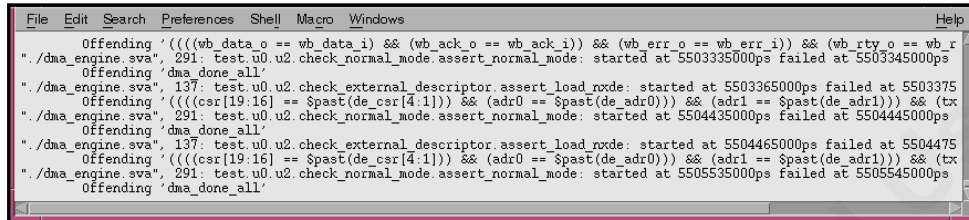
Sample output

```
1: 25
1: 35
1: 45
1: 55
2: 55
2: 65
2: 65
1: 75
1: 85
2: 85
2: 85
2: 85
2: 95
2: 95
3: 95
```

- **\$display and \$time statements can be added between sequences**
 - Use local variables and assign \$time as an attempt ID

Assertions Report

- The runtime option `-assert report` creates a textual report on all the failures



```
File Edit Search Preferences Shell Macro Windows Help
Offending '(((wb_data_o == wb_data_i) && (wb_ack_o == wb_ack_i)) && (wb_err_o == wb_err_i)) && (wb_rty_o == wb_rty_i))'
"/dma_engine.sva", 291: test.u0.u2.check_normal_mode.assert_normal_mode: started at 5503335000ps failed at 5503345000ps
Offending 'dma_done_all'
"/dma_engine.sva", 137: test.u0.u2.check_external_descriptor.assert_load_rvde: started at 5503365000ps failed at 5503375000ps
Offending '(((csr[19:16] == $past(de_csr[4:1])) && (adr0 == $past(de_adr0))) && (adr1 == $past(de_adr1))) && (tx == $past(de_tx))'
"/dma_engine.sva", 291: test.u0.u2.check_normal_mode.assert_normal_mode: started at 5504435000ps failed at 5504445000ps
Offending 'dma_done_all'
"/dma_engine.sva", 137: test.u0.u2.check_external_descriptor.assert_load_rvde: started at 5504465000ps failed at 5504475000ps
Offending '(((csr[19:16] == $past(de_csr[4:1])) && (adr0 == $past(de_adr0))) && (adr1 == $past(de_adr1))) && (tx == $past(de_tx))'
"/dma_engine.sva", 291: test.u0.u2.check_normal_mode.assert_normal_mode: started at 5505535000ps failed at 5505545000ps
Offending 'dma_done_all'
```

Assertion report file (failures)

- `-assert success`
 - Will print all the successes
 - Will not print vacuous successes

Assertion Debug With DVE

- **Integrated assertions, testbench and RTL debugger**
 - Supports both simulation (VCS) and formal (Magellan)
- **Assertions and results are shown in design hierarchy**
 - Sort and filter assertion results
 - View success, failure and timing for assertion evaluations
- **Unified environment for tracing assertion code**
- **Visual debug aids enhanced waveform debug**
 - Guidance symbols
 - View contributing signals
 - Trace active drivers/loads

2-53

DVE Debug Flow

■ Steps to debug assertion failures through DVE

- On loading the waveform database (.vpd file), DVE displays the list of failures in the assertion pane
 - ◆ Need to setup assertion window to show by default
- Each failure in the assertion pane reports
 - ◆ Start time and end time of that attempt
 - ◆ Name and instance of the SVA
 - ◆ Reason of failure
- Double-clicking on the assertions of interest in the assertion pane, will bring up the corresponding waveform window and source windows
 - ◆ Waveform window shows all signals, events, and Booleans associated with the attempt

2-54

Invoking DVE: Interactive Mode

■ Compile design with support for assertion debug with DVE

```
%vcs -sverilog source.v -R -debug_access+all \
-assert dve -assert enable_diag
```

-sverilog	enables the use of SystemVerilog code
-assert enable_diag	control result reporting at runtime (optional)
-assert dve	enables you to see assertion attempts
-debug_access+all	enables debug including line tracing

■ Starts DVE from existing simulation executable

```
%simv -gui -assert maxfail=10 -assert report=t1_assert_rep.txt
```

2-55

In addition to loading VPD files for post-processing, you can also setup and run a simulation interactively in real-time using a compiled Verilog, VHDL, or mixed design.

Compile-Time Options:

-debug_access+<opt> Enables command line debugging with option <opt>.

Common options are

- pp Creates a VPD file (when used with the VCS system task \$vcdpluson) and enables DVE for post-processing a design. Using -debug_access+pp can save compilation time by eliminating the overhead of compiling with -debug_access+all.
- all This turns on all debugging capabilities except inside cells and encrypted modules. This affects simulation performance depending on the size of your design. It creates a VPD file.

Please refer to VCS User Guide on solvnet for other useful options

Run-time Options:

- ucli Forces runtime to go into UCLI mode by default Also see the following section, Runtime Options, for more information.
- gui starts DVE at runtime in interactive mode.


Invoking DVE: Post-Processing Mode

■ Launch DVE

```
% simv -assert maxfail=10 -assert report=t1_asrt_rep.txt  
% dve &
```

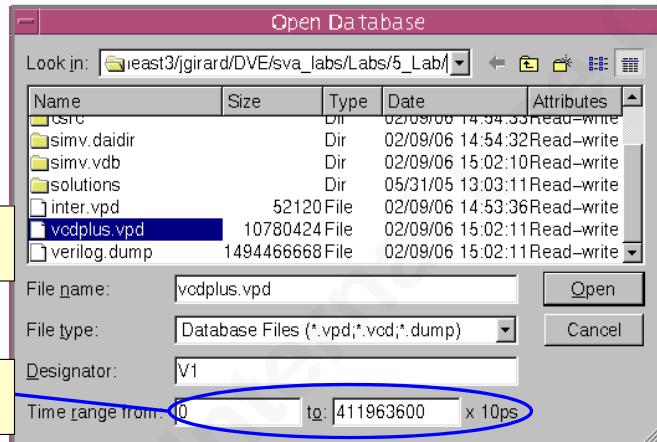
Testbench must have `$vcdpluson()` ;

■ Open Database (vcd,vpd)

- Click the Open Database icon  to open dialog box

Tip: `dve -vpd <filename>`
Starts DVE and loads file

Optional:
Select time range



2-56

“vpd” files are usually created during VCS batch simulation, and used in post processing using DVE.

vpd files are compressed binary files and VCD files are ASCII files. The compressed binary format of vpd files mean that they take far less disk space and load much faster than VCD files.

In post-processing DVE can go both forwards and backwards in time and analyze the complete set of data.

Opening a Database - Do either of the following:

- From the Menu bar, select File>Open Database
- From the Toolbar, click the Open Database icon

The Open Database dialog box appears.

In the Open Database dialog box,  browse and select name of the VPD file you want to load.

Enter or accept a Designator for your design.

Enter a time range to load. The default is start of simulation to the end.

Click OK.

DVE loads the selected VPD file.

DVE Assertion Debug Overview

The screenshot displays the DVE Assertion Debug interface with several key components labeled:

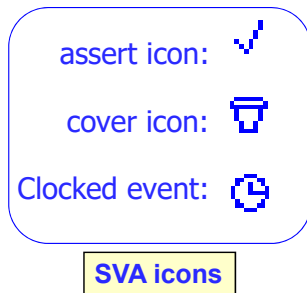
- Hierarchy Browser:** Shows the project hierarchy with components like `u0`, `u1`, and `u2`.
- Data pane:** A table showing assertion results. The `assert_chunk_dec` assertion is highlighted as a failure.
- Assertion pane:** Lists all assertions, including `assert_chunk_dec`, `assert_tot_dec`, `assert_ARS`, and `assert_HW_handshake`.
- Source Window:** Displays the Verilog source code for `assert_chunk_dec`, showing the assertion property and the `St0->St1` transition.
- Waveform Window:** Shows a timing diagram with signals like `READ[10:0]` and `state[10:0]`. A red arrow points from the failure in the Assertion pane to the corresponding signal in the waveform.

A yellow box with the text "Double click assertion failure to populate waveform and source windows" has arrows pointing to the `assert_chunk_dec` failure in the Assertion pane and the `assert_chunk_dec` entry in the Source Window.

2-57

Complied Assertions

- Properties become objects which are visible within DVE windows and panes



Tooltip over assertion to view hierarchal label

Variable	Value	Type
!assert_ARS	Success	Assertion
!assert_HW_handshake	Success	Assertion
!assert_STOP	Success	Assertion
!cover_STOP	Mismatch	Cover property
test.u0.u2.check_HW_handshake.cover_STOP		
!ck_start	'h0	Wire (Port In)
!ck	'h1	Wire (Port In)
!csr [31:0]	'h00000000	Wire (Port In)
!dma_abort	'h0	Wire (Port In)

DVE Data pane

Assertion Pane

- The DVE assertion pane displays a flat list of all assert and cover statements defined in the design

Expand to see all failures

Selection Filters

default: automatically display assertion failures

Assertions	all	at time	current	to end	failures	incompletes	successes	all
Name	Instance	Start	End	Delta	Reason	Failures	Successes	
assert_pass_through	test.u...	150038500	150038500	0	((((wb_data_o == wb_data_i) && (wb_a...	57	1500	
assert_pass_through	test.u...	76053500	76053500	0	((((wb_data_o == wb_data_i) && (wb_a...	15360	1347	
assert_chunk_dec	test.u...	150057500	150057500	0	(chunk_dec == ((\$past(state) != READ...	6	1501	
assert_tot_dec	test.u...	150057500	150057500	0	(tsz_dec == ((\$past(state) != READ) ...	6	1501	
Failure1		150057500	150057500	0	(tsz_dec == ((\$past(state) != READ) ...			
((tsz_dec == ((\$past(state) != READ) && (state == READ)))								
Failure3		150092500	150092500	0	(tsz_dec == ((\$past(state) != READ) ...			
Failure4		150093500	150093500	0	(tsz_dec == ((\$past(state) != READ) ...			
Failure5		150127500	150127500	0	(tsz_dec == ((\$past(state) != READ) ...			
Failure6		150128500	150128500	0	(tsz_dec == ((\$past(state) != READ) ...			

Reason: offending string that caused assertion failure (tooltip)

DVE Assertion pane

2-59

Attempted Assertions

- All assertion attempts can be viewed individually

Assertion pane

Display box shows all attempts for a single assertion or cover property

Assertion window CSM

Default: show the first 10 successes or failures

Assertion Attempt Display

The current assertion has 192 failures, 281128 successes and 0 incompletes.

Filter: Show from to

#Attempts - failures: successes: incompletes:

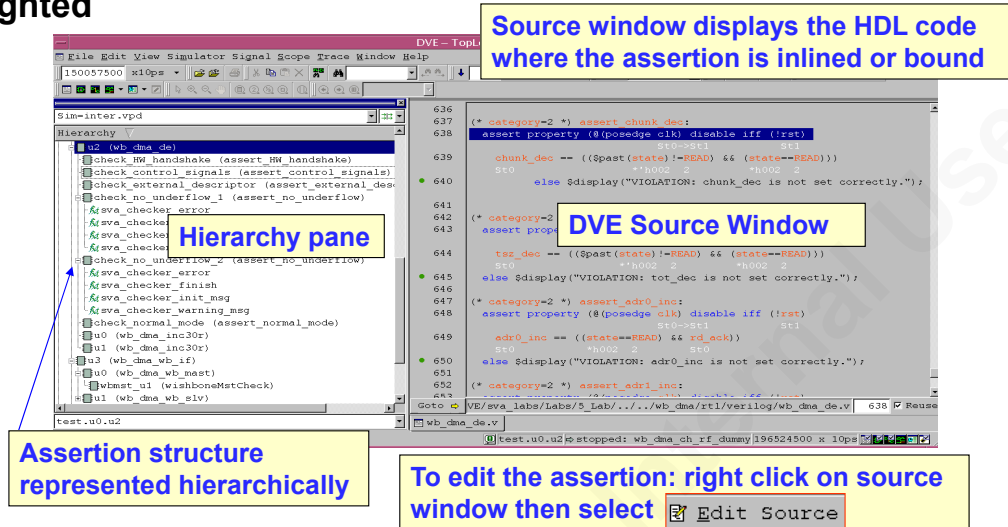
Name	Start	End	Delta	Reason
Failure1	195723500	195724500	1000	(tot_start && (pre_tot_sz == txsz[11:0]))
Failure2	195793500	195794500	1000	(tot_start && (pre_tot_sz == txsz[11:0]))
Failure3	195863500	195864500	1000	(tot_start && (pre_tot_sz == txsz[11:0]))
Failure4	195933500	195934500	1000	(tot_start && (pre_tot_sz == txsz[11:0]))
Failure5	196086500	196087500	1000	(tot_start && (pre_tot_sz == txsz[11:0]))
Failure6	196221500	196222500	1000	(tot_start && (pre_tot_sz == txsz[11:0]))
Failure7	196356500	196357500	1000	(tot_start && (pre_tot_sz == txsz[11:0]))
Failure8	196491500	196492500	1000	(tot_start && (pre_tot_sz == txsz[11:0]))
Failure9	196709500	196710500	1000	(tot_start && (pre_tot_sz == txsz[11:0]))
Failure10	196909500	196910500	1000	(tot_start && (pre_tot_sz == txsz[11:0]))

Trace Attempt Close Tips >>

2-60

SVA Source Code

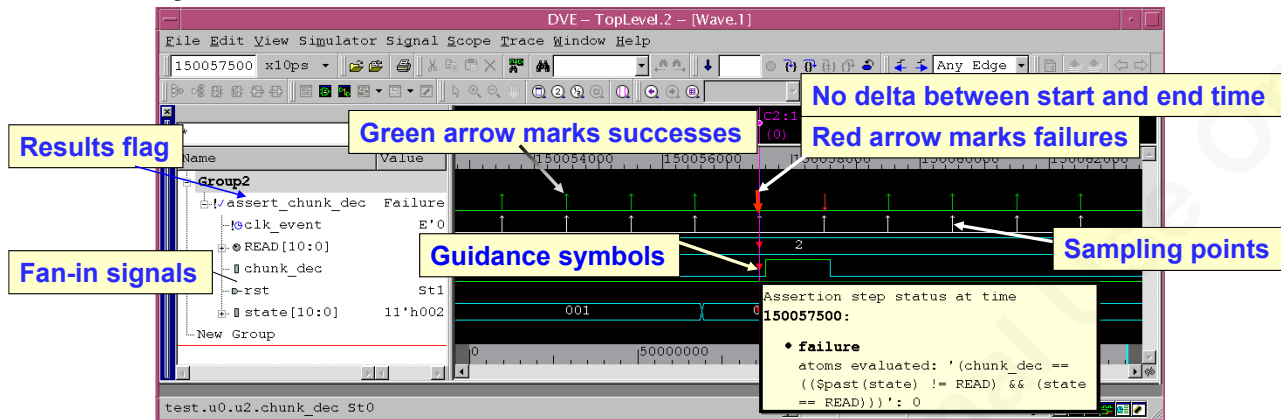
- Source window displays the assertion code with the assertion highlighted



2-61

SVA Waveforms

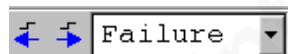
■ Identify the cause of an assertion failure:



DVE Waveform Window



Zoom controls



Search controls

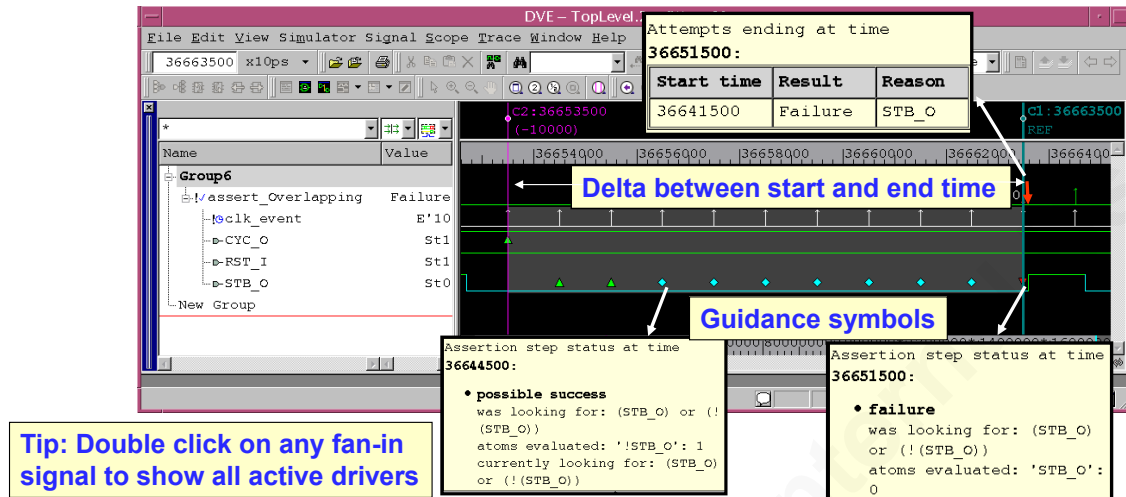
Tip: Use search controls to quickly search for assert or cover statements

2-62

SVA Failure in the Waveform Window

Tip: Double click on assertion result arrows to expand time window

The C1 and C2 cursors are automatically placed at the start and end of the assertion

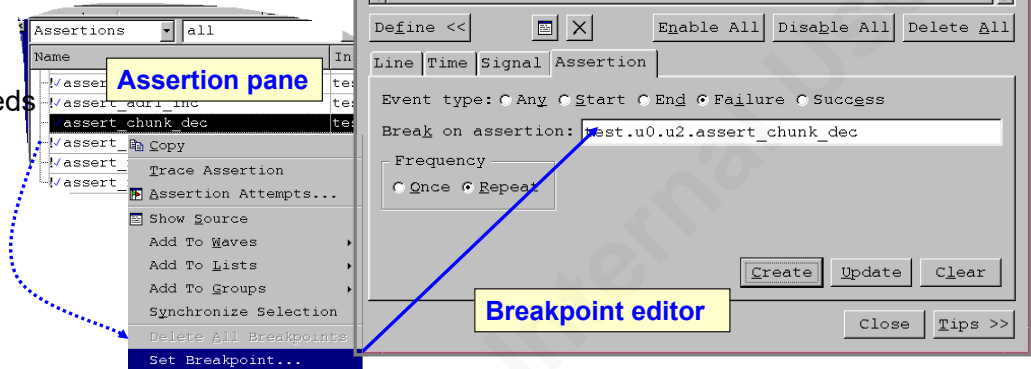


2-63

Assertion Breakpoints – DVE Interactive Mode

■ Assertion breakpoints can be set either in the data and assertion panes or Source window

- Breakpoint can be triggered when assertion:
 - ◆ Starts
 - ◆ Ends
 - ◆ Fails
 - ◆ Succeeds



2-64

Debug Summary

- **Text Messages**
- **DVE graphical debug aids for assertions**
 - Assertion pane, Data Pane, Hierarchy Pane
 - Waveform Window & Source Window
 - Guidance symbols
 - Assertion breakpoints
 - Driver/Load Pane
- **Methods available for controlling assertions**
 - Assertion control system tasks
 - Compile time exclusions (VCS)
 - Categories (tool specific)

2-65

Appendix

VCS Compile and Runtime Switches for SVA

Using parameters in SVA

Creating Assertion Arrays

SVA Debugging

Using SVA in VHDL and Mixed-Languages

Improving Assertion Performance

2-66

In-lining SVA in VHDL via pragmas

- VCS only
- Embed SVA code via `--sva` pragmas

```
architecture RTL of FF is
begin
    ...
    --sva property p1;
    --    @(posedge clk) a && b | -> ##DELAY !c;
    -- endproperty
    --sva a1 : assert property(p1);
end architecture
```

2-67

SVA Inlined in VHDL Design

```
entity counter is
  port (
    count_out : out std_logic_vector(7 downto 0);
    ld_enb, clk : in std_logic;
    count_enb : in std_logic);
end entity counter;

architecture rtl of counter is
  signal cnt : unsigned(7 downto 0); -- count register
  signal tc : std_logic; -- terminal count
  ...
begin -- architecture rtl
  cnt <= cnt;
  -- inline checker library instantiation:

  Chk2:assert_even_parity generic map (0,8,0,"PARITY") port map (clk,1'b1,cnt);
  ...
end architecture rtl
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.sva_lib.all;
```

2-68

SVA Bind with VHDL

■ Binding File can contain

- Binding to Entity
 - ◆ `bind vhdl_ent checker c1 (.clk(clk), .a(a), .b(b));`
- Binding to Entity and Specific Architecture
 - ◆ `bind \vhdl_ent(rtl) checker c1 (.clk(clk), .a(a), .b(b));`
- Binding to Entity and Architecture pair from specific library
 - ◆ `bind \wrk1.vhdl_ent(rtl) checker c1 (.clk(clk), .a(a), .b(b));`
- Binding to specific instance
 - ◆ `bind top.m1 checker c2 (.clk(clk), .a(a), .b(b));`

■ All bind statements above can also go inside a module

```
module bind_a;  
    bind vhdl_ent checker c1 (.clk(clk), .a(a), .b(b));  
endmodule
```

2-69

SVA in VHDL Compilation Flow

- **Analyze the Verilog file containing the assertions**

- `% vlogan -sverilog sva.v`

- **Analyze the VHDL source**

- `% vhdlan dut_E.vhd dut_A.vhd`

- **Analyze top level testbench, if top is**

- Verilog: `% vlogan tb.v`
- VHDL: `% vhdlan tb_E.vhd tb_A.vhd`

- **Elaborate design (pass `-sva_bind_enable/-sva_bind` options)**

`% vcs -sva_bind_enable <module_with_binds> -sverilog work.tb`

`% vcs -sva_bind <file_with_binds_outside_module> -sverilog work.tb`

- For Verilog-only designs do not pass `-sva_bind/-sva_bind_enable`

2-70

Appendix

VCS Compile and Runtime Switches for SVA

Using parameters in SVA

Creating Assertion Arrays

SVA Debugging

Using SVA in VHDL and Mixed-Languages

Improving Assertion Performance

2-71

Assertion Performance Mode

- **Enhanced assertion performance targeting regression type simulations or fast detection of assertion failures**
 - Enabled by compile time switch `-assert failonly`
 - Assertion debug compile options should not be used
 - ◆ `-assert dve`
 - ◆ `-assert enable_diag`
 - ◆ Some forms of the `-debug_access` option
 - Limits the amount of failure reporting and thus the information for debugging such as
 - ◆ No attempt start time
 - ◆ No offending expression

2-72

Agenda

DAY

1

1 Introduction to SVA

2 Immediate and Concurrent Assertions



3 Concurrent Assertions: Sequences



4 SVA Coverage



5 SVA Libraries



Unit Objectives



After completing this unit, you should be able to:

- Describe what a SVA sequence is
- Describe the sequence construct in SVA
- Understand the use of a sequence in properties
- Describe how to write sequences
- Describe how to repeat sequences
- Describe how to combine sequences

3-2

Sequences

■ A sequence is

- A mechanism for defining sequential behaviors
- A temporal expression over zero, finite or infinite clock cycles
 - ♦ `req ##1 ack` is a sequence expression that spans one clock cycle
 - ♦ Sequence expression `^{data, parity} == 1'b0` spans zero clocks
- A building block for properties
- Used in a concurrent assertion to check for correct behavior
 - ♦ Combinations of sequential behaviors are checked

```
assert property @(posedge clk)
  (req ##1 ack) | -> ##1 (^{data, parity} == 1'b0);
```

3-3

Sequence Expressions

■ Sequence expressions consist of

- Simple Sequence expression: Boolean expressions
 - ◆ Covered in last unit - Boolean operators: `&&`, `|`, `!`, etc.
- Complex Sequence expression: temporal expressions
 - ◆ Covered in last unit:
Delay operators: `##n`, `##[m:n]` etc.
System Functions: `$rose`, `$fell` etc.
 - ◆ Boolean repetition (`[*m:n]` , `[->m:n]` , `[=m:n]`)
 - ◆ Sequence repetition(`[*m:n]`)
 - ◆ Sequence operators `and`, `or`, `within`, etc.
- Sequence can not contain property implication operators:
 - ◆ `| ->`, `| =>`

■ Return True for a match, False for mismatch

3-4

Note: A Sequence can have multiple matches. For example

```
req ##[1:2] ack
```

may return True over two clocks if `ack` is true for both clocks following `req`. Later we will see how multiple matches of a sequence affects it's use in properties and other sequence operations.

Named Sequences

- Use Named Sequence to encapsulate sequence expressions

```
sequence <sequence_name( [<port_1>, ..., <port_n>] )>;  
    @(clocking expression) <sequence_expression>  
endsequence: [<sequence_name>]
```

- Define named sequences that are reusable and use in property

```
sequence s1(req,ack);  
    req ##1 ack;  
endsequence: s1
```

```
sequence s2;  
    ^{data,parity} == 1'b0;  
endsequence: s2
```

- Clocking can be declared in sequence or property, but not in both

```
assert property(@(posedge ck) s1(r1, a1) | -> ##1 s2);
```

3-5

Local Variables in Sequences and Properties

■ Local Variables

- Are declared inside a named property or sequence
- Allow data to be captured at one point and referenced later in a thread (local to the thread)
 - ◆ New automatic variable allocated for each assertion thread

■ Use of variable is in sequence_match_item attached to the sequence boolean expression

- Sampled value of RHS assigned

(boolean_expr, sequence_match_item)

Variable declaration

```
property p;
  reg [31:0] data; //Same size as data_in
  @(posedge clk)
  (ack_in, data = data_in)
  | -> ##5 (req_out && (data_out == data));
endproperty
```

Local variable initialization in sequence match item

Compare with saved value

3-6

A sequence expression can be followed by a sequence_match_item. The sequence_match_item can be one of

operator_assignment

inc_or_dec_expression

subroutine_call – functions and void functions only

Note: References to any variables in the sequence_match_item expression uses the sampled value. The value of data_in in the example above is the sampled value (from the Preponed region).

To use local variables you must use the vcs compile switch -assert svaext

Boolean Repetition Operator

■ Match multiple occurrences of a Boolean expression

- **Boolean expression** [**<n|min:max>*]
- There is an implicit ##1 between each Boolean evaluation
- Repetition over a number or range of clocks

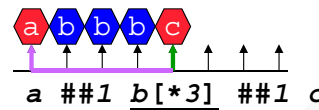
Example with repetition range

`b[*1:3]`

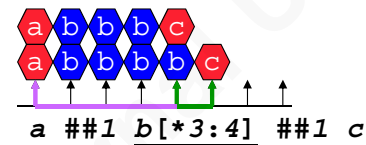
is equivalent to

`(b) or`
`(b ##1 b) or`
`(b ##1 b ##1 b)`

Consecutive Repetition



Range Repetition



3-7

For repeating boolean expression, the repetition operator can be used to loop on a specific boolean. Note, there is an implicit ##1 between each repetition of the boolean expression.

For example

```
sequence s_ready_3;
  ready [*3];
endsequence: s_ready_3
```

is equivalent to

```
sequence s_ready_3;
  ready ##1 ready ##1 ready;
endsequence: s_ready_3
```

Special Case: Repetition with Zero Range

■ Match zero or more occurrences of a **Boolean** expression

■ **Boolean expression** [$*0:max$]

- Matches even if Boolean does not occur for zero-repetition case
- If zero-repetition case occurs

One **##1** delay cycle is removed from **n**

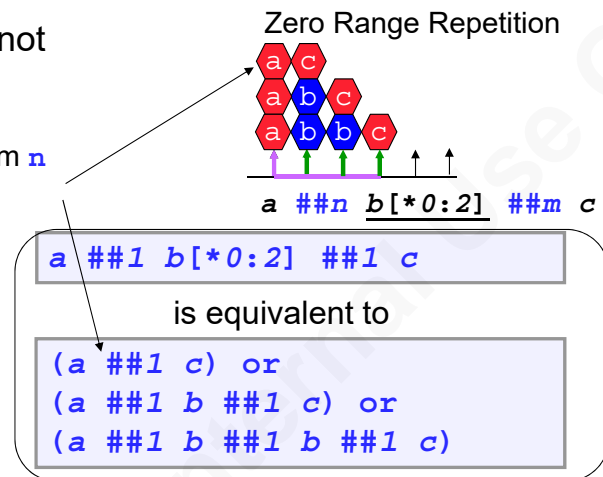
a ##n b[*0:2] ##m c

Will be modified to

a ##(n - 1 + m) c

If $(n - 1 + m)$ resolves to < 0

Zero-repetition case will result
in no match



3-8

From IEEE 1800 Spec:

Using 0 as the repetition number, an empty sequence results, as follows: **a [*0]**

An empty sequence is one that does not match over any positive number of clock ticks. The following rules apply for concatenating sequences with empty sequences. An empty sequence is denoted as **empty**, and a sequence is denoted as **seq**.

- **(empty ##0 seq)** does not result in a match.
- **(seq ##0 empty)** does not result in a match.
- **(empty ##n seq)**, where **n** is greater than 0, is equivalent to **(##(n-1) seq)**.
- **(seq ##n empty)**, where **n** is greater than 0, is equivalent to **(seq ##(n-1) 'true)**.

For example:

b ##1 (a[*0] ##0 c)

produces no match of the sequence.

b ##1 a[*0:1] ##2 c

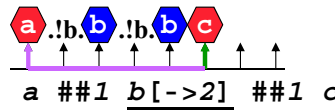
is equivalent to

(b ##2 c) or (b ##1 a ##2 c)

Boolean Goto Repetition

- If there is one or more delay cycles between repeated **Boolean expressions**, use goto operator **[->]**
 - **Boolean expression [-> <count | min:max>]**
 - Boolean expression is repeated with one or more cycle delays between the repetitions
 - Match must end on last occurrence of Boolean expression

“Goto” Repetition



3-9

b [->2]

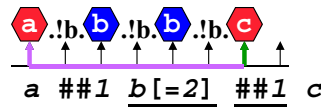
Is equivalent to:

(!b [*0:\$] ##1 b) [*2]

Boolean Non-Consecutive Repetition

- For delay cycles after repeated **Boolean** expression match, use non-consecutive repetition operator [=]
- **Boolean expression [= <count | min:max>]**
- Boolean expression is repeated with one or more cycle delays between the repetitions
- Match does not have to end on last occurrence of Boolean expression

"Non-Consecutive" Repetition



3-10

a ##1 b [=2] ##1 c

Is equivalent to:

a ##1 b [->2] ##1 !b [*0:\$] ##1 c

Sequence Repetition Operator

■ Match zero or more occurrences of a [sequence](#) expression

- **sequence** [**count|*min:max**]
- There is an implicit **##1** between each instance

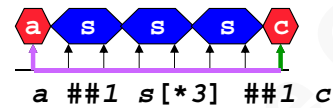
Example with repetition range

```
(a ##2 b) [*1:3]
```

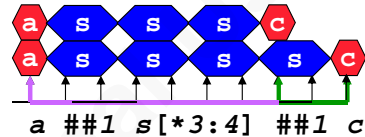
is equivalent to

```
(a ##2 b) or
((a ##2 b) ##1 (a ##2 b)) or
((a ##2 b) ##1 (a ##2 b) ##1 (a ##2 b))
```

Consecutive Repetition



Range Repetition



3-11

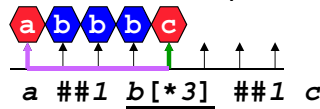
For repeating sequences, the repetition operator can be used to loop on a specific sequence.

Note, there is an implicit ##1 between each repetition of the sequence.

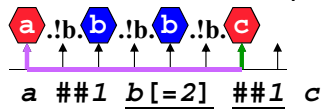
Repetition Summary

■ Boolean Repetition

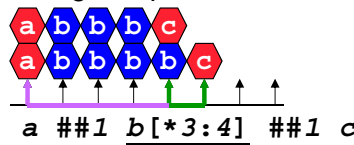
Consecutive Repetition



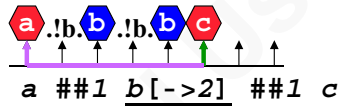
Non-Consecutive Repetition



Range Repetition

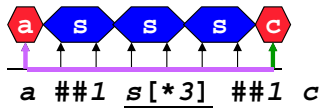


Goto Repetition

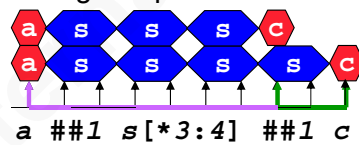


■ Sequence Repetition

Consecutive Repetition



Range Repetition



3-12

For Consecutive repetition the boolean expressions may be replaced by sequences.
For Goto and Non-consecutive Repetition only boolean expressions are valid.

Sequence Operators: and, or

■ Logical and/or of Sequences

- Sequences must start at same time
- End times can be different

■ and of sequences

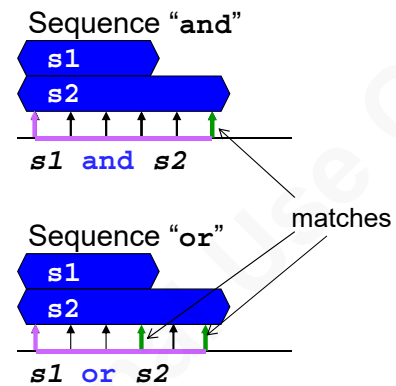
- matches after both sequences match

Use **and** when both expressions are expected to succeed, but end times are different

■ or of sequences

- matches whenever either sequence matches

Use **or** when at least one expression is expected to succeed



3-13

Sequence logical operators are held until the last sequence completes

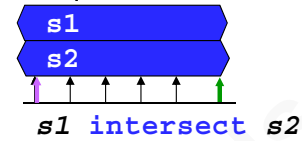
Sequence Operators: intersect, within

- **Sequence expression may be logically connected together**

- **intersect of sequences**

- Sequences must start at the same time
- Matches only if both sequences match at same time

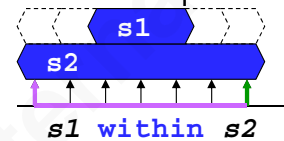
Sequence "intersect"



- **within a sequence**

- Sequence fully contained within another sequence
 - ◆ Sequences must start and match within another
 - ◆ Ok to start and match at the same time (see intersect above)

Sequence "within"
another Sequence

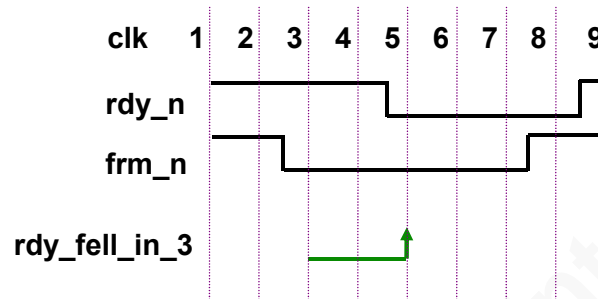


3-14

Sequence logical operators are held until the last sequence completes

intersect Example

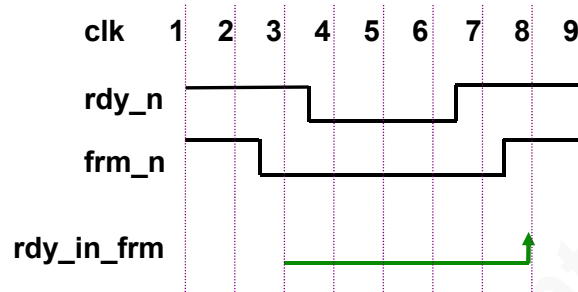
```
//check for first falling rdy 3 clocks after falling frm_n  
sequence rdy_fell_in_3;  
  1[*3] intersect  
  ($fell(frm_n) ##1 $fell(rdy_n) [->1]);  
endsequence: rdy_fell_in_3
```



3-15

within Example

```
sequence rdy_in_frm;  
  ($fell(rdy_n) ##1 $rose(rdy_n)[->1]) within  
    ($fell(frm_n) ##1 $rose(frm_n)[->1]);  
endsequence: rdy_in_frm
```



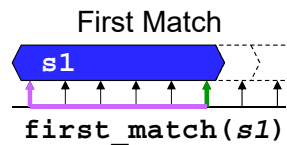
3-16

Note that the signal waveform shown is only one of many successful waveforms possible.

Sequence Operators: `first_match`

- Sequences can have multiple matches
 - To prevent multiple thread matches use `first_match`
- `first_match` returns match upon the first match of a sequence
 - Other threads started at the same time are terminated

```
first_match (stb ##[1:3] ack) | -> rdy;
```



3-17

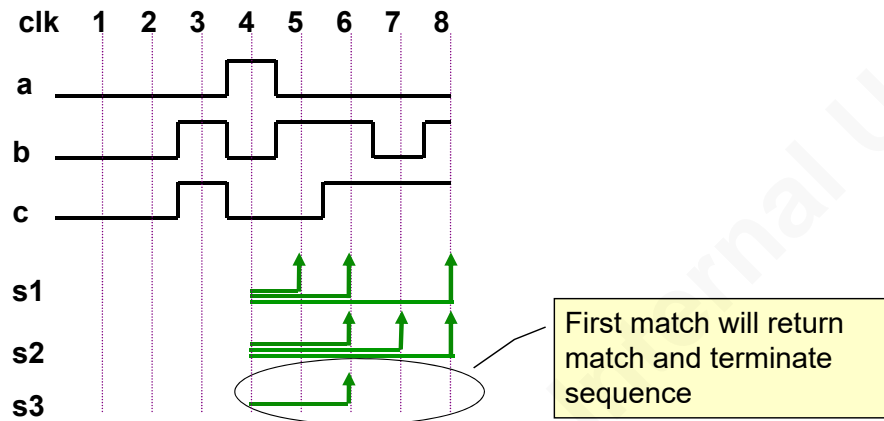
An implication operator has the structure

```
sequence_expr | -> (property_expr | sequence_expr)
```

For each of multiple matches of the antecedent the consequent is checked. Use the `first_match` operator to limit the check for only the first match of the antecedent.

first_match Example

```
sequence s1; a ##[1:4] b; endsequence  
sequence s2; !b ##[2:4] c; endsequence  
sequence s3; first_match(s1 intersect s2);  
endsequence
```



3-18

Sequence Operators: throughout

■ Ensures a Boolean condition holds for an entire sequence

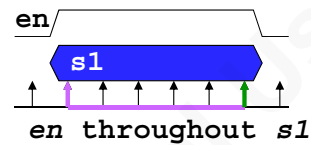
- Checks that constant Boolean condition holds for the duration of a sequence
- Sequence fails if Boolean condition evaluates false anytime during the sequence
- Use **throughout** operator

Example of **throughout** assertions

"*enable* must be asserted over the entire bus cycle"

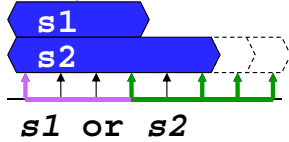
"*ready* must be sampled low 8 clocks while *transmit* is active"

Expression "throughout"
a Sequence

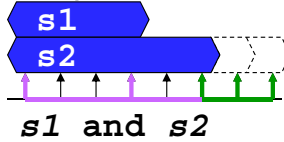


Sequence Operators Summary

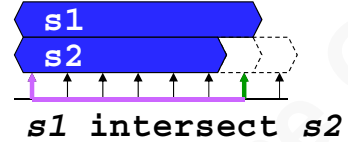
Sequence "or"



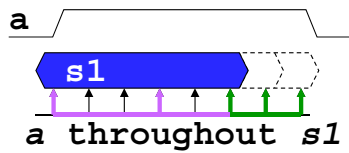
Sequence "and"



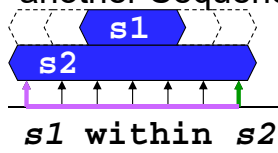
Sequence "intersect"



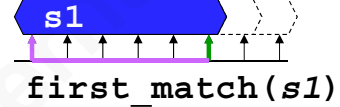
Expression "throughout"
a Sequence



Sequence "within"
another Sequence



First Match



3-20

Operator Precedence

Operators (high to low precedence)	Description
<code>[*n] , [*m:n]</code>	consecutive repetition, consecutive range repetition
<code>[->n]</code>	goto boolean repetition (non-consecutive, exact)
<code>[=n]</code>	non-consecutive boolean repetition
<code>and</code>	all sequences expected to match, end times may differ
<code>intersect</code>	all sequences expected match, same start & end times
<code>or</code>	one or more expected sequences to match
<code>throughout</code>	expression expected to match throughout a sequence
<code>within</code>	containment of a sequence expression
<code>##n, ##[m:n]</code>	sequence delay, sequence delay window

3-21

Sequences in Assertion Properties

■ Using sequences in assertion properties:

```
sequence wb_access;  
    stb ##0 ack ##[1:16] !stb;  
endsequence: wb_access  
property p_BackToBack_Accesses;  
    @(posedge clk) disable iff (!reset_n)  
        $rose(stb) |-> wb_access ##1 wb_access;  
endproperty
```

1. Literal Sequence
2. Property Operator
3. Named Sequence
4. Sequence Operator
5. Named Sequence

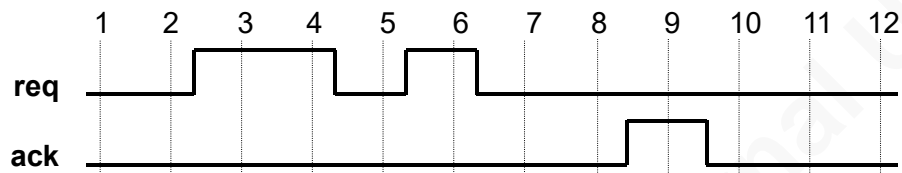
A property may contain sequences
A sequence may contain sequences
A sequence can not contain properties

3-22

Exercise 1 – Arrival of Requests

- English description:

"check for ack high within 3 cycles after 3 requests"

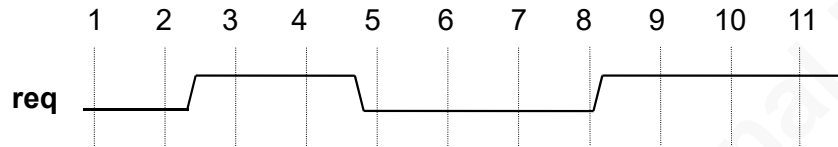


3-23

Exercise 2 – Min Arrival Rate of Requests

- English description:

"There should be a minimum of 2 requests within 10 cycles"



3-24

Unit Objectives

Having completed this unit, you should be able to:

- Describe the sequence construct in SVA
- Understand the use of a sequence in properties
- Describe how to write sequences
- Describe how to repeat sequences
- Describe how to combine sequences



3-25

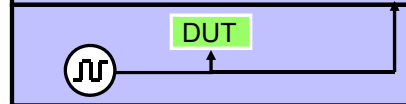
Lab 2 Introduction



45 minutes

Implement Sequences & Properties in Assertions

```
program automatic test(...);  
  initial begin  
    $vcdpluson;  
    $display("Hello");  
    reset();  
  end  
  task reset();  
    ...  
  endtask  
endprogram
```



Implement Sequences and Properties

Define Assertions and Bindings

Compile & Simulate

Check with DVE/Verdi³

3-26

Appendix

Solutions to Exercises

Sequence method triggered

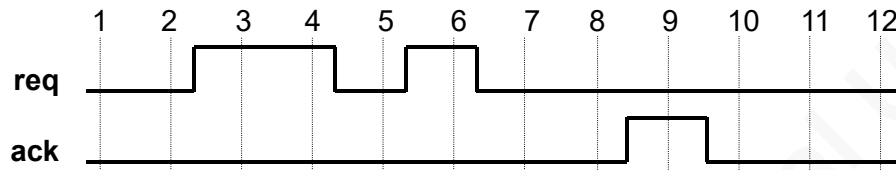
Sequences with multiple clocks

3-27

Solution 1 – Arrival of Requests

- English description:

"check for ack high within 3 cycles after 3 requests"



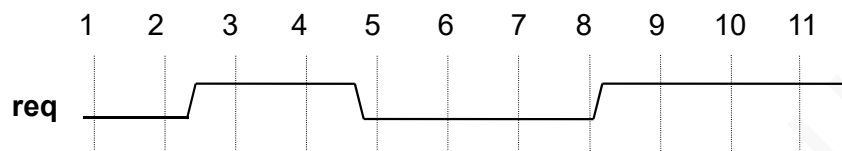
```
sequence ack_after_3_requests(clk, req, ack);  
    @(posedge clk) req[->3] ##[0:3] ack;  
endsequence
```

Clock is optional in a sequence

Solution 2 – Min Arrival Rate of Requests

■ English description:

"There should be a minimum of 2 requests within 10 cycles"



```
sequence min2req;  
    req[->2:10] within 1[*10];  
endsequence: min2req
```

3-29

Appendix

Solutions to Exercises

Sequence method triggered

Sequences with multiple clocks

3-30

Sequence Method `triggered`

- How can you test if a sequence has ended?
 - Method `triggered` samples a sequence and
 - ◆ Returns true iff the current clock tick is the end time of a match of some evaluation attempt of the sequence
 - ◆ Returns false otherwise
- Occurrence of end point of a sequence can be tested as boolean expression in another sequence

```
sequence s_prologue;  
  @(posedge clk) a ##1 b;  
endsequence
```

If no sampling event specified in sequence:
Error-[USUMOE] Unclocked sequence used in matched
or triggered sequence 's_prologue' should be clocked

```
property p_follow; @(posedge clk)  
  s_prologue.triggered | -> ##[0:4] req;  
endproperty
```

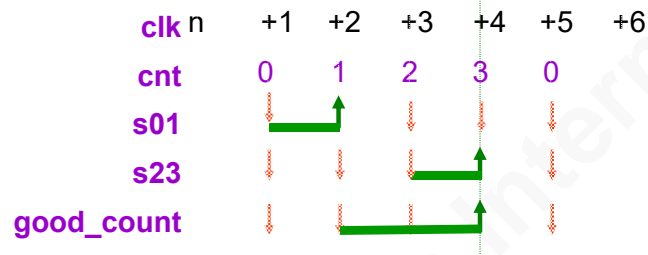
3-31

triggered Statement Example

■ Example

- Two-bit counter

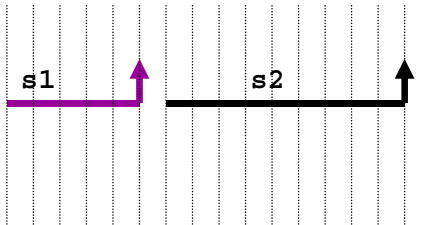
```
sequence s01;  
  @(posedge clk) (cnt == 0) ##1 (cnt == 1);  
endsequence  
sequence s23;  
  @(posedge clk) (cnt == 2) ##1 (cnt == 3);  
endsequence  
sequence good_count;  
  s01.triggered ##2 s23.triggered;  
endsequence
```



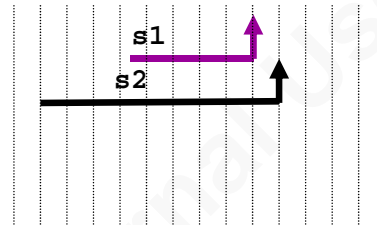
3-32

Using triggered to Join Parallel Sequences

```
sequence foo;  
    s1 ##1 s2;  
endsequence
```



```
sequence foo;  
    s1.triggered ##1 s2.triggered;  
endsequence
```



3-33

Appendix

Solutions to Exercises

Sequence method triggered

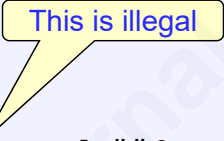
Sequences with multiple clocks

3-34

Sequences with Multiple Clocks

- What if you want to join the end of two sequences that use two clocks or different clock edges?

```
sequence s_slow;  
  @(posedge slow_clk) a ##1 b;  
endsequence  
  
sequence s_fast;  
  @(posedge fast_clk) c ##1 d;  
endsequence  
  
sequence good_count;  
  @(posedge fast_clk) s_slow.triggered ##2 s_fast;  
endsequence
```



- For this kind of requirement use method `matched`

3-35

Sequences with Multiple Clocks

■ **matched** method provides synchronization between clocks

- It stores the result of sampling the sequence until the arrival of the next tick of the destination clock

```
sequence s_slow;
  @(posedge slow_clk) a ##1 b;
endsequence

sequence s_fast;
  @(posedge fast_clk) c ##1 d;
endsequence

sequence good_count;
  @(posedge fast_clk) s_slow.matched ##2 s_fast;
endsequence
```

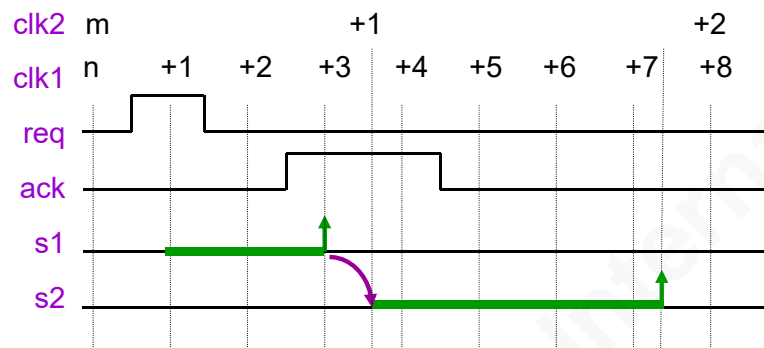
3-36

What if you have multiple clock ticks of `fast_clk` in one period of `slow_clk`? It must say something about memorizing if triggered was true in the cycle of the destination sequence. latch and hold if triggered was true till the nearest future sampling clock edge of the destination clock

matched Example in Multi-Clock

```
sequence s1;  
  @(posedge clk1) req ##[1:2] ack;  
endsequence
```

```
property p;  
  @(posedge clk2) s1.matched |-> ##1 !ack ;  
endproperty
```



3-37

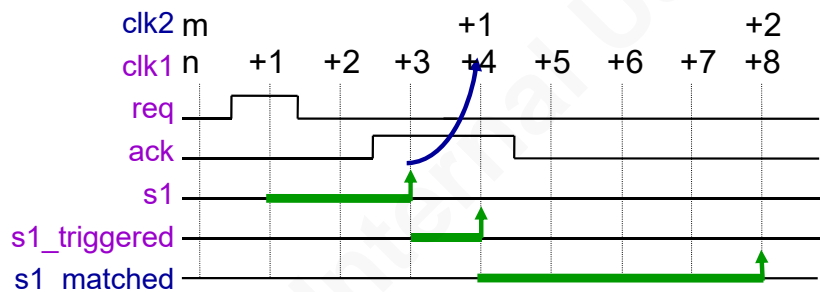
Difference Between matched and triggered

- **Use triggered to join sequences in a single clock domain**
 - Return value of `matched` will be available only in the next closest tick of the destination clock
- **Use matched to join differently- or multiply-clocked sequences**
 - Return value of `triggered` is available only in the current clock.

```
sequence s1;
  @(posedge clk1)
  req ##[1:2] ack;
endsequence

sequence s1_triggered;
  @(posedge clk1)
  s1.triggered ##1 ack;
endsequence

sequence s1_matched;
  @(posedge clk2)
  s1.matched ##1 !ack;
endsequence
```



3-38

If two clocks coincide when `matched` is returned, for example `matched(t(n+1))` happens at clock `n+4` of `clk1` which coincides clock `m+1` of `clk2`, the return value of `matched(t(n+1))` will be only available at clock `m+2` of `clk2`.

Agenda

DAY

1

1 Introduction to SVA

2 Immediate and Concurrent Assertions



3 Concurrent Assertions: Sequences



4 SVA Coverage



5 SVA Libraries



Unit Objectives



After completing this unit, you should be able to:

- Describe types of coverage using SVA
- Add coverage properties
- Describe compile and runtime options for Assertion coverage
- Process coverage results after simulation

4-2

Assertion Coverage: `assert` Statement

- To generate coverage information for `assert` statements
 - use `-cm assert` switch at compile
 - ◆ Generates the number of times for pass/fail/incomplete data
 - ◆ Generates both, vacuous and non-vacuous coverage
- Important to know if assertion is passing only vacuously because its triggering conditions were never set up
 - For example, the following will only pass vacuously if `req` is never asserted

```
assert property @(posedge clk)
    disable iff (reset) req ==> ##1 ack );
```

4-3

Default Assertion Coverage

■ By default the Assertion Coverage reports

- Did the test suite properly exercise each assertion ?
 - ◆ Attempt succeeded or failed
- Were there any vacuous-only successes or proofs ?
 - ◆ Antecedents that never evaluated true

`A | -> B;`

■ Assertion Coverage currently does not report

- Did every path or range value in the assertion get exercised in assertions with complex sequencing and/or multiple branches?

`A | -> ##[0:10] (B || C) ;`

No indication of which values in the range were asserted

completes if either event B or C is true, no indication of which branch taken

4-4

A critical aspect of verification methodology is measuring progress. In order to answer the question "What should I do next?" it is necessary to know what has already been done and point out what has yet to be done. In ABV (Assertion Based Verification), the important metrics address the following questions:

Are there enough assertions in the design?

Is the verification plan for simulation complete?

How thorough is the formal verification analysis?

One of the first questions that design teams ask when adopting ABV is "How do I know enough assertions have been placed in a design?" Obviously, a bug can only be found if an assertion is placed that would point to the bug when the assertion is violated.

Unfortunately, a metric to address the sufficiency of assertions is complex: it must address not only coverage of the HDL code, but also the function of the code.

Functional Coverage: `cover property`

- The `cover property` statement identifies sequences or properties to be tracked for functional coverage
 - It is like `assert property` except:
 - ♦ It generates only number of times for success data
 - ♦ It has no `else` clause
 - i.e. it does not enforce assertion
 - ♦ Failures are never reported for cover properties
- By default coverage is always on for cover properties
 - Turn coverage off using `-assert disable_cover`
- Cover properties produce external reports

```
sequence s8;  
    @(posedge clk) a ##[1:5] b;  
endsequence  
c7: cover property (s8);
```

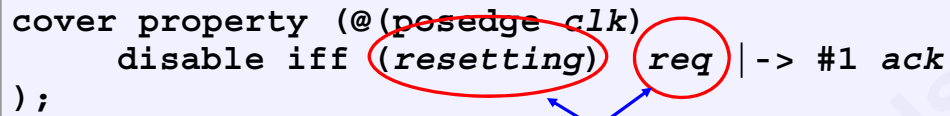
4-5

Coding Guidelines: cover property

■ Avoid using `disable iff` and implication operator

- Coverage for vacuous successes reported have to be filtered by simulator engine or reporting tool

```
cover property (@(posedge clk)
  disable iff (resetting) req | -> #1 ack
);
```



Vacuous match when *resetting* or *req* == 0

■ Instead use `throughout` to disable property and no implication in property expression

- All successes are non-vacuous and will be covered

```
cover property (@(posedge clk)
  (!resetting) throughout req ##1 ack );
```

4-6

If you use `disable iff` and implication operators in a cover statement, you will have lots of vacuous successes. It's up to each simulation engine to distinguish and filter vacuous successes.

Adding Coverage Example (1/2)

“What are the min/max latencies of my protocol?”

```
`ifdef COVER_ON
genvar i;
generate begin
  for (i = 0; i <= 7; i = i+1) begin : latency
    cov: cover property @(posedge clk)
      !(resetting) throughout req ##i ack );
  end : latency
end
endgenerate
`endif
```

Note: potential performance hit if max-min large

4-7

Adding Coverage Example (2/2)

“which combinations of arbiter requests (8 channels) occurred?”

```
// req[7:0] is a 8-bit vector
`ifdef COVER_ON
genvar i;
generate
for (i = 0; i <= 255; i = i+1) begin : channel
    cov_req: cover property (@(posedge clk) req == i );
end: channel
endgenerate
`endif
```

Note: This can be done better using covergroups

4-8

The combinations can be covered better and more easily using SystemVerilog covergroups
covergroup cov_arbiter with sample(bit[7:0] req);

```
coverpoint req {
    arbiter_requests[] = { [0:255] };
}
endgroup: cov_arbiter
```

```
cov_arbiter cov_arb = new();
...
cov_arb.sample(req);
```

or you could sample the covergroup in the success block of an assertion
assert property (@(posedge clk)
\$fell(frame_n[port_no]) |-> ##3 (req[port_no]))
cov_arb.sample(req);

Compile-Time Options

```
% vcs -f filelist -sverilog -cm assert -cm_assert_hier file_1
```

Switches	Description	Required?
-f <i>filelist</i>	Normal files or switches	optional
-sverilog	activate SV(A) compilation	required
-cm assert	turns on coverage for asserts	optional
-assert disable_cover	turns off coverage for cover statements	optional
-cm_assert_hier <i>file_1</i>	limits coverage to module hierarchy specified in file <i>file_1</i>	optional

By default, VCS compilation creates a functional coverage database called simv.vdb

4-9

Run-Time Options

```
%simv -cm assert -cm_assert_name blockA
```

Switches	Description	Required?
<runtime options>	non coverage related options	optional
-cm assert	enables collection of “assert” coverage	optional
-cm_assert_name <i>name</i>	name functional coverage database file	optional*

* If you do not specify `-cm_assert_name`, the default name of “results” is used for naming the functional coverage file in the simv.vdb/snps/fcov database

4-10

Unified Report Generator (URG)

```
%ourg -dir simv.vdb -metric assert -grade
```

Switch	Description (<i>default</i>)
-dir	Indicates the locations of the coverage databases (<i>current directory</i>)
-metric	Indicates the types of coverage for which you need the report to be generated (<i>all types</i>)
-grade	Indicates auto grading of tests with optional first argument of % and second argument of time (<i>none</i>)

- % urg -help
 - To view all URG options

4-11

Unit Objectives Review

Having completed this unit, you should be able to:

- Describe types of coverage using SVA
- Add coverage properties
- Describe compile and runtime options for Assertion coverage
- Process coverage results after simulation



4-12

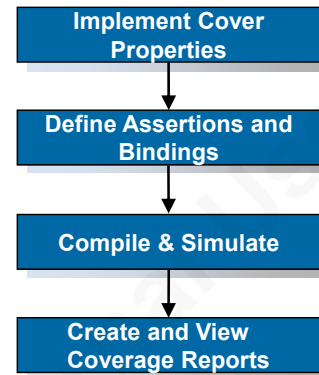
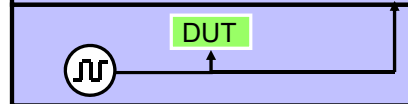
Lab 3 Introduction



30 minutes

Implement Assertion Coverage

```
program automatic test(...);  
  initial begin  
    $vcdpluson;  
    $display("Hello");  
    reset();  
  end  
  task reset();  
    ...  
  endtask  
endprogram
```



4-13

Appendix

Coverage Methodology Recommendations

Coverage Reporter

4-14

Methodology Recommendations (1/3)

- **Store all coverage under one root directory**

- `% mkdir ./coverage`

- **Specify compile and runtime argument**

```
% vcs ... -cm_dir coverage/cov.cm -cm assert+...  
% simv ... -cm assert+...
```

- `coverage/cov.cm.vdb`
 - ◆ Directory created for Testbench coverage (functional)

4-15

Methodology Recommendations (2/3)

- **Use a common base name for the database files**
 - `-cm_name MyTest1` // TB and Code
 - `-cm_assert_name MyTest1` // Assertion
- **Store URG generated reports together**
 - `% mkdir ./coverage/cov_reports`
 - `% urg -report ./coverage/cov_reports <...>`
- **Save the merged coverage database for later viewing or debugging**
 - `% urg <...> -dbname merged_coverage`

4-16

Methodology Recommendations (3/3)

- **Keep code coverage logs in a common directory**
 - `% mkdir ./logs`
 - `% simv <...> -cm_log ./logs/MyTest1.cm.log`
- **Controlling the hierarchy for coverage gathering**
 - `% vcs <...> -cm_assert_hier cov.cfg`
 - ◆ `cov.cfg` example entries
 - `+tree test_top.dut`
 - `-module fifo`
- **Do not collect code coverage information on SVA checker library modules**
 - If using compile option `-cm_libs yv`, add `-module assert*` to the `cov.cfg` file

4-17

SVA checker library is covered in the next unit.

Appendix

Coverage Methodology Recommendations

Coverage Reporter

4-18

Coverage Report: Dashboard View

% firefox ./coverage/cov_reports/dashboard.html

SYNOPSYS® Dashboard

dashboard | hierarchy | modlist | groups | tests | asserts

Date: Thu Aug 4 16:50:32 2016
User: XXXXXXXXXX
Version: K-2015.09
Command line: urg -dir simv.vdb
Number of tests: 1

Total Hierarchical Coverage Summary

SCORE	LINE	TOGGLE	FSM	ASSERT
87.43	92.03	95.07		75.19

Hierarchical coverage data for top-level instances

SCORE	LINE	TOGGLE	FSM	ASSERT	NAME
89.93	96.65	97.94		75.19	router_test_top
0.00	0.00	0.00			rtslice1

Total Module Definition Coverage Summary

SCORE	LINE	TOGGLE	FSM	ASSERT
75.32	62.69	85.50		77.78

Total Groups Coverage Summary

SCORE	WEIGHT
30.00	1

4-19

Coverage Report: Assertions View



Assertions

[dashboard](#) | [hierarchy](#) | [modlist](#) | [groups](#) | [tests](#) | [asserts](#)

Assertions by Category

	ASSERT	PROPERTIES	SEQUENCES
Total	49	80	0
Category 0	49	80	0

Assertions by Severity

	ASSERT	PROPERTIES	SEQUENCES
Total	49	80	0
Severity 0	49	80	0

Summary for Assertions

	NUMBER	PERCENT
Total Number	49	100.00
Uncovered	0	0.00
Success	49	100.00
Failure	0	0.00
Incomplete	0	0.00
Without Attempts	0	0.00

Summary for Cover Properties

	NUMBER	PERCENT
Total Number	80	100.00
Uncovered	32	40.00
Matches	48	60.00

[illegible]

4- 20

Agenda

DAY

1

1 Introduction to SVA

2 Immediate and Concurrent Assertions



3 Concurrent Assertions: Sequences



4 SVA Coverage



5 SVA Libraries



Unit Objectives



After completing this unit, you should be able to:

- Describe assertion libraries
- Describe the benefits of assertion libraries
- Describe the global macros used in standard SVA libraries
- Describe the common parameters passed to library assertions
- Describe the coverage parameters used by the library assertions

5-2

What are Assertion Libraries?

- **Collection of monitoring modules to verify that:**
 - A DUT performs a specific behavior correctly(check)
 - A certain set of behaviors occurred(coverage)
- **A Library may consist of:**
 - Basic checker library
 - ◆ Typical behaviors, independent of a design or protocol
 - ◆ Relatively simple containing few assertions
 - Reusable protocol checkers
 - ◆ Standard protocols like PCI Express, ARM AMBA
 - ◆ Complete and often complex set of properties
 - Assertions usable as assertions or assumptions

5-3

Synopsys Checker Library Benefits

■ Fast start up for typical behaviors

- No need to know assertion language or how to write monitors
- 53 basic pre-verified checkers (common RTL structures)
 - ◆ 31 are OVL 2.3 equivalent
 - ◆ 22 provide more complex functionality

■ Easily modifiable

- Generic, fully parameterized library
- Assertions implemented as SV modules and interfaces
- All assertions have consistent structure
- Source code can be customized to create custom libraries

👉 Coverage points included use cover property and covergroup constructs

5-4

OVL stands for Open Verification Library. OVL is a vendor- and language-independent template interface for design validation. It is governed by the Accellera consortium which sets standards for Design Verification technology. OVL libraries may be (and are) written in any language. OVL also specifies coverage points and switches for different types of coverage when using an OVL compliant assertion library.

Checker Library vs. Custom Assertions

- **Design engineers generally use assertions for RTL block-level verification**
 - In-line SVA assertions tied closely with implementation
 - Checker Library has many common assertions pre-coded for ease of adoption
- **Verification engineers create more complex assertions**
 - Verify system level assumptions across blocks
 - Custom assertions provide capabilities beyond the scope of the checker library
 - These are typically kept in a separate file from design HDL.

5-5

Synopsys SVA Checker Library

Found in: \$VCS_HOME/packages/sva_cg

Basic	State Integrity	Protocol
<code>assert_always</code> <code>assert_always_on_edge</code> <code>assert_change</code> <code>assert_cycle_sequence</code> <code>assert_decrement</code> <code>assert_delta</code> <code>assert_even_parity</code> <code>assert_fifo_index</code> <code>assert_frame</code> <code>assert_handshake</code> <code>assert_implication</code> <code>assert_increment</code> <code>assert_never*</code> <code>assert_next</code> <code>assert_no_overflow</code> <code>assert_no_transition</code>	<code>assert_no_underflow</code> <code>assert_odd_parity</code> <code>assert_one_cold</code> <code>assert_one_hot</code> <code>assert_proposition</code> <code>assert_quiescent_state</code> <code>assert_range</code> <code>assert_time</code> <code>assert_transition</code> <code>assert_unchange</code> <code>assert_width</code> <code>assert_win_change*</code> <code>assert_win_unchange</code> <code>assert_window</code> <code>assert_zero_one_hot</code>	<code>assert_arbiter*</code> <code>assert_data_used</code> <code>assert_dual_clk_fifo</code> <code>assert_fifo*</code> <code>assert_memory_sync</code> <code>assert_memory_async</code> <code>assert_no_contention</code> <code>assert_req_ack_unique</code> <code>assert_req_requires</code> <code>assert_stack</code> <code>assert_valid_id</code> <code>assert_multiport_fifo</code>
	Value Integrity	
	<code>assert_bits</code> <code>assert_value</code>	
	Temporal Sequence	
	<code>assert_hold_value</code> <code>assert_reg_loaded</code>	

* See coming examples 

5-6

SVA Checker library is from Synopsys, it is backward compatible with OVL but introduces additional sophisticated checks.

Documentation is available on solvnet.

General library of common functionality checks

Both combinational and sequential checks

Superset of Accellera Open Verification Library (OVL)

Each checker

Can be in-lined directly in HDL code

Can be bound to HDL in separate file through binding

Checker Library Flow

■ Pick behaviors to be asserted

- Use the Checker Library Guide
\$VCS_HOME/doc/UserGuide/pdf/sva_checkerlib.pdf

■ Each checker is a module

- Bind checker to appropriate design module(s)
 - ◆ Bind may be in separate binding module or outside of modules

■ Compile using vcs

```
% vcs -f filelist -sverilog +define+ASSERT_ON \
+incdir+$VCS_HOME/packages/sva_cg \
+libext+.sv -y $VCS_HOME/packages/sva_cg \
dut.v
```

5-7

Checker Binding

■ Use a separate module for the bindings

- Easier to maintain and debug
- Compatible with other tools

```
module my_bindings;  
  //Disable checking when (!rst || de_start)  
  bind wb_dma_de assert_no_underflow  
    #(0,9,0,-1,0,"VIOLATION: chunk_cnt has underflow.")  
    check_no_underflow_1  
    (clk, rst &&! de_start , chunk_cnt);  
endmodule: my_bindings
```

checker name

parameter list

instance name

test expression

reset expression

5-8

While this is not a requirement, placing binds inside a module ensures compatibility with other tools like VC-Formal, VCSMX etc. The code maintenance is also cleaner since one knows where to find all the binds.

VCS Compilation Switches

```
vcs -f filelist -sverilog  
+define+ASSERT_ON \  
+incdir+$VCS_HOME/packages/sva_cg \  
+libext+.sv -y $VCS_HOME/packages/sva_cg
```

Switches	Description
+define+ASSERT_ON	Macro activates assertions
+incdir+\$VCS_HOME/packages/sva_cg	SVA checker include directory
-y \$VCS_HOME/packages/sva_cg	Physical location of Checker Library
+libext+.sv	Library file extension (.sv files)

All switches listed are required when using the checker library

5-9

Global Macros

- Compile-time global macros apply to all instances

```
+define+ASSERT_ON +define+ASSERT_GLOBAL_RESET+top.pwr_reset
```

Macro Name	Description (default behavior)
ASSERT_ON	Activates assertions (not defined)
COVER_ON	Activates coverage (not defined)
ASSERT_INIT_MSG	Reports configuration for each checker at start of simulation
ASSERT_GLOBAL_RESET	Assign a global reset signal (reset_n port)
SVA_CHECKER_NO_MESSAGE	Turns off reporting custom messages(ON)
SVA_CHECKER_INTERFACE	Activates assertions for use in a interface(OFF)

5-10

Checker Parameters

- Each checker has its own set of parameters described in the documentation
 - Plus a set of parameters common to all checkers

Common Parameter	Description(default)
<code>severity_level</code>	Currently not supported (0)
<code>property_type</code>	<code>property_type=1</code> implies <code>assume property</code> for formal verification (<code>property_type = 0</code> implies <code>assert property</code>) Other types reserved for future use
<code>msg</code>	Custom error message (“VIOLATION”)
<code>category</code>	Specifies the category for possible filtering during simulation(0)
<code>coverage_level_<i>i</i></code>	Specifies coverage to be enabled (none)

5-11

OVL-Like Example 1

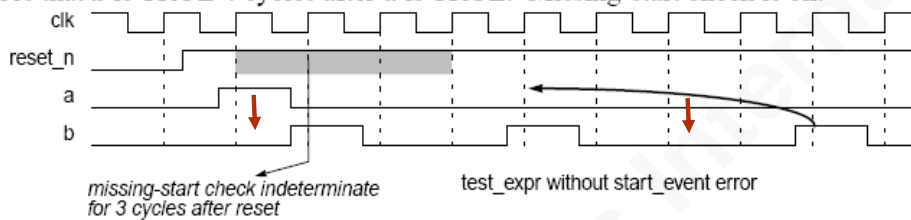
assert_next: Ensures that the value *test_expr* evaluates TRUE exactly *num_cks* number of clock cycles later after a *start_event*

severity_level=0
num_cks=4 (# of clocks)
check_overlapping=0 (off)
only_if =1 (check missing start)

Example:

```
assert_next #(0,4,0,1) vld_a_b (clk, reset_n, a, b );
```

Ensures that b is TRUE 4 cycles after a is TRUE. Missing-start check is on.



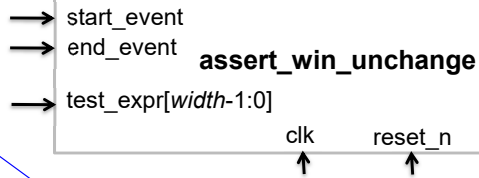
Parameters:
severity_level
num_cks
check_overlapping
only_if
property_type
msg
category
coverage_level

5-12

OVL-Like Example 2

assert_win_unchange: Ensures that *test_expr* does not change in the window between a *start_event* and an *end_event*

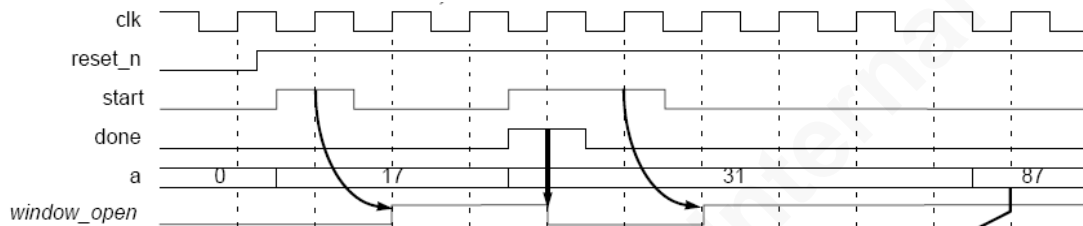
severity_level=0
width=8



Parameters:
severity_level
width
property_type
msg
category
coverage_level

Example:

```
assert_win_unchange #(0,8) w1 (clk,reset_n,start,a,done);
```



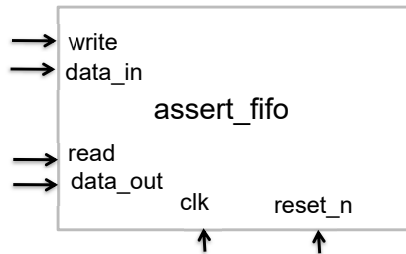
ASSERT_WIN_UNCHANGE Error: a changed during divide

5-13

Protocol Example 1

assert_fifo: Implements a checker for a single-clock, single in- and single out-port fifo

severity_level=0
 depth=10 (# of elements in fifo)
 elem_sz=16 (data is 16 bits)
 hi_water_mark=0 (chk disabled)
 enq_lat=0 (enq latency)
 deq_lat=0 (deq latency)
 overflow_chk=1 (default: enable)
 value_chk=1 (default: enable)
 pass_thur=0 (underflow)
 edge_expr=0 (posedge)
 msg=default
 category=0



Parameters:

severity_level	depth
elem_sz	hi_water_mark
enq_lat	deq_lat
overflow_chk	value_chk
pass_thur	edge_expr
msg	category
coverage_level	

Example:

data_in is pushed on FIFO when *write* is 1, *data_out* must be equal to that value at the head of the FIFO when *read* is 1

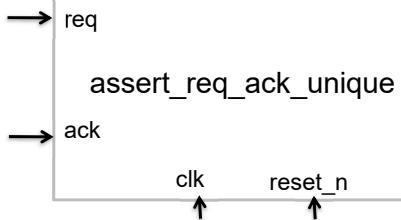
```
assert_fifo #(.depth(10), .elem_sz(16))
SVA_FIFO_inst (clk,reset_n,write,data_in,read,data_out);
```

5-14

Protocol Example 2

assert_req_ack_unique: Verify each req receives an ack within specified interval

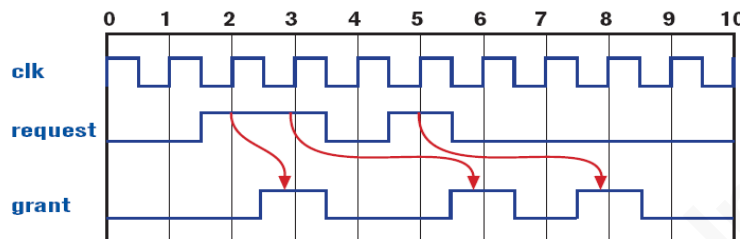
severity_level=0
min_time=1
max_time=10
max_time_log_2=4
version=0 (max_time less than 15)
edge_expr=0 (posedge)
msg="req_ack_failed"



Parameters:

severity_level min_time
max_time_log_2 version
edge_expr msg
category
coverage_level

```
assert_req_ack_unique #(0,1,10, 4, 0, 0,"req_ack failed")
  req_grant_1_1 (clk, reset_n, request, grant);
```



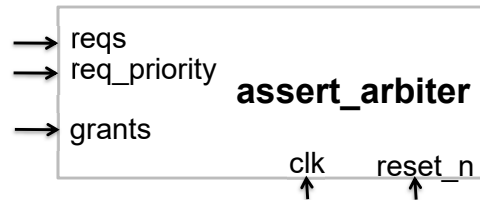
Verifies that for each request there is a grant received within 1 and 10 clock cycles.

Use version 1 (time stamp)
if max_time > 15 (time stamp)

5-15

Protocol Example 3 (1/2)

assert_arbiter: Ensures arbiter provides grants to corresponding requests within min_lat and max_lat cycles



- Provides basic functionality
 - ◆ Priority and Round-robin, FIFO or LRU
- Verifies mutual exclusion, latency, and selection rules

Parameters:
severity_level
no_chnl
bw_prio
grant_one_chk
req_priority_chk
arbitration_rule
min_lat
max_lat
edge_expr
msg
category
coverage_level

5-16

Protocol Example 3 (2/2)

■ Checking round-robin arbiter logic for 32 inputs

- No channel will be issued more than one grant while other channels have requests pending (1024 checkers)
- No grant will be issued without a request (32 checkers)
- Only one grant is issued per clock cycle (1 checker)
- Grant must de-assert within one clock cycle (32 checkers)

```
assert_arbiter #(
    .no_chnl(32),           // number of request/grant pairs
    .bw_prio(3),           // # bits used to encode priority
    .grant_one_chk(1),     // on
    .req_priority_chk(0),   // off
    .arbitration_rule(1)    // round-robin
    .min_lat(1),.max_lat(0), // min/max latency
    .coverage_level_2(3),   // coverage level 2 enabled
    .coverage_level_3(3))) // coverage level 3 enabled
    arb_u1 (CLK, RST, REQ, req_priority, GNT);
```

5-17

Adding Coverage: VCS Checker Library

■ VCS SVA checker library coverage

- In addition to performing function checks, each checker can collect and report coverage
- Detailed statistics and transaction information
 - ◆ Has a FIFO been filled and emptied
 - ◆ Did an arbiter receive a grant for each channel
 - ◆ Have all variations of legal transactions occurred
 - ◆ Did maximum and minimum latencies occur

■ Enabled through compiler directive

- `+define+COVER_ON`

■ Controlled through parameters

- `coverage_level_1, _2, _3` parameters

5-18

Adding Coverage: Checker Library Levels

■ Coverage parameter `coverage_level_L`, L = 1, 2, 3

- Bits control individual cover points
 - ◆ L=1 - Basic coverage (default)
 - cover property - Example: No. of Enqueues and Dequeues
 - ◆ L=2 - Ranges of data / delay values
 - covergroup and cover property - Example: latency values
 - ◆ L=3 - Corner coverage
 - cover property - Example: Specified Min latency
- Levels and cover points enabled and disabled independently

5-19

See SVA Checkers User Guide on solvnet for more information on Coverage Levels.

Elaboration checks on SVA Checker Library

■ Compile-time switches to check for correct parameters

- Enable or disable elaboration checks on the parameters passed to checkers
 - ◆ Use the IEEE1800-2009 elaboration system tasks
 - ◆ Report errors without generating the runtime executable
- Enable parameter checks on all checkers

```
+define+SVACG_ENABLE_ALL_ELAB_PARAM_CHECKS
```

- Enable the parameter check on an individual checker

```
+define+SVACG_ENABLE_ELAB_PARAM_CHECKS_<checker_name>
```

- Disable the parameter check on an individual checker

```
+define+SVACG_DISABLE_ELAB_PARAM_CHECKS_<checker_name>
```

5-20

Unit Objectives

Having completed this unit, you should be able to:

- Describe assertion libraries
- Describe the benefits of assertion libraries
- Describe the global macros used in standard SVA libraries
- Describe the common parameters passed to library assertions
- Describe the coverage parameters used by the library assertions



5-21

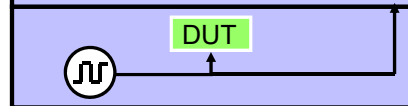
Lab 4 Introduction



30 minutes

Implement Assertions Using SVA Library

```
program automatic test(...);  
  initial begin  
    $vcdpluson;  
    $display("Hello");  
    reset();  
  end  
  task reset();  
    ...  
  endtask  
endprogram
```



Implement Assertion
using SVA Library

Compile & Simulate

Debug Results using
DVE/Verdi³

5-22

That's all Folks!



5-23

This page was intentionally left blank

Registered For Synopsys Internal Use Only

Customer Support

Synopsys Customer Education Services
© 2019 Synopsys, Inc. All Rights Reserved

20181001

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](#) >

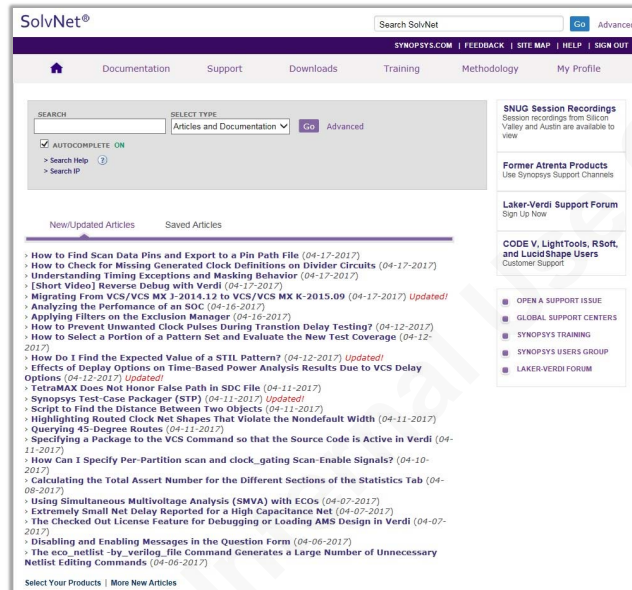
[DFM](#) >

<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)



<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 image displays three screenshots of the Synopsys SolvNet registration process. The top-left screenshot shows the 'New User Registration' page with a 'SIGN UP FOR AN ACCOUNT' button. The top-right screenshot shows the registration form with fields for Corporate Email, Username, First Name, Last Name, Password, and Re-enter Password. The bottom screenshot shows the 'New User Registration' page with a 'Synopsys Site ID' field and an 'Add Another Site' button. Red arrows point from the instructions to these specific elements: from step 2 to the 'SIGN UP FOR AN ACCOUNT' button, from step 3 to the Password field, and from step 4 to the 'Synopsys Site ID' field.

<https://solvnet.synopsys.com/ProcessRegistration> CS - 4

Support Center

■ Industry seasoned Application Engineers:

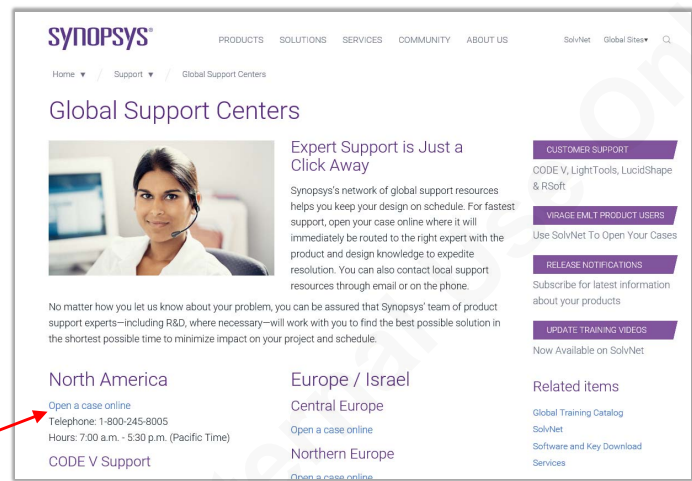
- 50% of the support staff has >5 years applied experience
- Many tool specialist AEs with >12 years industry experience
- Engineers located worldwide

■ Great wealth of applied knowledge:

- Service >2000 issues per month

■ Remote access, and interactive debug, available via WebEx

Contact us:
Open a support case



<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.

Registered For Synopsys Internal Use Only