cadence®

Assertion Checking in Simulation

Product Version 15.2

February 2016

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

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

Contents

1

Introduction to ABV

Assertion Based Verification provides an effective way to improve quality of verification by providing better controllability and observability of design errors. ABV is enabled with specification of assertions in the design. Assertions are executable specification of the design and are mostly written as 'assert' properties used to check the design functionality and 'cover' properties primarily used for functional coverage. Assertion can be verified using different ABV technologies: Formal, Simulation and Simulation Acceleration. This book focuses primarily on checking of assertion in Simulation to help catch most design errors early in the design and verification cycle.
This chapter contains the following sections:

Why Use ABV

ABV is a form of whitebox testing. That is, properties--asserted behaviors--can monitor behavior deep within the design and not just at the interface level. This feature lets you identify errors sooner and closer to the source, and also lets you specify the functional coverage points deep within the design.

The following are the primary technologies that use assertions:

- Formal Analysis--Formal property checking tools are used to prove that a property that is asserted will hold true for all input conditions that do not violate an assumed behavior.
- Simulation--Dynamic checking of monitors in simulation
- Acceleration-based assertions (ABA)--Another form of dynamic ABV that uses a simulation accelerator.

Some verification goals are well-suited for ABV technologies, and other goals might be handled better by other tools. The key is to use ABV where you get the best return on investment.

Advantages of ABV

ABV technologies can isolate functional errors close to their source. The advantages of using ABV in different areas is explained below:

Using ABV for Error Detection

Simulation Stimulus Errors

Errors often occur due to differences between what a designer assumes is valid for an interface, and how the interface is actually stimulated. If the stimulus violates the assumption, an assertion error points to this discrepancy. There is no need to wait for the error to propagate. This technique helps to detect test errors, interconnect errors, and errors in the assumptions that are made by the designer. This technique can be especially useful in emulation, where it can be difficult to determine the root cause of a failure.

Protocol Errors

Protocol errors at the physical interface can be difficult to debug, as these errors are not propagated to high-level response checkers for many simulation cycles. Also, depending on the tolerance of the devices connected to the interface, the error might not propagate at all. When a device that violates the protocol, is reused, it can cause a problem in its new environment. Assertions are of great value here, because they can detect immediately when the signaling protocol is violated. These assertions can be reused for all devices that have a common interface.

Design Errors

Assertions that describe design requirements are continually tested to ensure that these requirements hold true. This is especially useful when a design is enhanced, to ensure that existing functionality is not broken. Assertions identify the problem close to the source, without the need for it to propagate to the outputs.

Performance Requirements

Properties can describe such requirements as minimum or maximum latency, minimal preamble, or minimum interframe gap. Formal tools are especially valuable for checking that these requirements are never violated, as a violation can cause interoperability issues that might even result in product recall.

Hard-to-Find Corner Cases

Hard-to-find corner cases are often missed because:

- The test does not run long enough to reach the condition that causes the failure to propagate to the output.
- The tester did not think to simulate the error.

In simulation, an assertion close to the error reports it immediately, so the error does not have to propagate to the output. A formal verification tool might be able to detect those conditions the tester does not think to simulate, assuming it has sufficient capacity for the analysis. If getting to the condition is the issue, acceleration or emulation is useful.

Errors Caused by State Machine Interactions

These errors are often very complex. A formal analysis tool might be able to identify the problems, depending on the complexity of the design.

Using Assertions as Coverage Points

Coverage points that are expressed using assertion languages can help ensure that the device is well tested. These coverage points:

- Identify holes in the tests--Assertions that describe good behavior can be used for functional coverage analysis, including FSM analysis and corner-case coverage. Coverage points can be automatically generated for assertions that are conditionally tested, to indicate that the behavior required to trigger the evaluation did not occur.
- Eliminate inefficient or redundant tests--Functional coverage and code coverage are good ways to identify redundant tests.

Using Assertions for Transaction Viewing

Transactions recorded from an assertion simulation are displayed in the SimVision waveform tool in the same way as other transactions. You can graphically view assertion failures in the context of the other activity in your system.

ABV for Block Verification

The technique used here is Formal analysis, also called *property checking*, refers to verification without user-provided stimulus. Formal analysis tools use mathematical techniques to prove the functionality of the device, where the intended functional behavior is specified by assertions. These tools can be applied at a designer's desktop, months before the testbench environment is ready, helping to find bugs earlier and faster. Additionally, formal analysis is good at exposing hard-to-find corner-case bugs that are not covered by the testbench.

How Formal Analysis Works

Formal analysis tools use assumptions that are expressed by you, using an assertion language or library, to constrain the inputs. It then tests all valid combinations and sequential combinations of inputs, and reports an error when an assertion is not true. Since no testbench is needed, verification can start earlier and bugs can be found sooner.

What Kind of Assertions to Write for Formal

Assertions for property checkers must be control-oriented, so you must create properties that describe the intended behavior of the outputs. It is also important to describe the behavior of the inputs that can affect the output behavior.

You can write data-oriented assertions if the state space is effectively managed by parameterizing storage elements and bus widths. You can also restrict data values to just a few, or you can, for example, check that the first data value written is the first data value read. Checking all data value combinations will generally result in capacity issues.

Who Writes them

Ideally, the designer writes the assertions that will be used in formal analysis. The designer must ensure that input assumptions are described. An unnecessary constraint might mask an error, while not enough constraints can result in false failures.

Constraining Inputs with Assertions

The biggest problem in formal analysis is overconstraining the inputs. When this happens, valid input configurations or sequences are not verified. Overconstrained inputs can mask bugs. Most tools can catch vacuity conditions--conditions that conflict--although not all conditions can be detected.

Making Constraints for Formal Analysis Complete and Accurate

If your input constraints are not complete, the tool will tell you by issuing a false failure. A false failure is when the waveform shows an input sequence that cannot happen in real life. This is an indicator that more constraints are needed. A false failure or vacuity might be reported that points to a condition where a constraint is not accurate.

Simulation is another way of detecting when a constraint is not accurate. The simulator will fail if the behavior that is expressed by a constraint is violated by the simulation stimulus. It is important to determine if the test provided incorrect stimulus, or whether the assertion was inaccurate.

ABV for Cluster Verification

The technique used here is simulation. Assertions describe expected behavior in a design. When you simulate assertions, the assertion statements act as monitors that check for this expected behavior during the simulation run. These monitors can report failures if and when they occur. Monitors can also record transactions that represent complex sequential signal relationships when probed.

In general, dynamic assertion simulation is driven by a testbench that tests a particular aspect of the design's functionality. Testbenches can be either traditional, signal-based testbenches, or transaction-based testbenches. There is no need to write special testbenches to simulate assertions.

If you turn on assertion code generation in the parser, the parser automatically compiles assertions into a simulation snapshot as part of the Incisive simulation front end.

During code generation, the parser creates executable code for assertions along with the rest of the design.

Tcl commands in the Incisive simulators let you activate and deactivate reporting of selected assertion state changes. You can get a plain text listing of the assertions in your design; their current state; and their active, finished, and failed counts.

During simulation, you can use run-time Tcl commands to:

- Query the presence, or current state, of assertions.

- Control the textual output generated by assertions at run time.
- Probe state changes, failure events, and transactions that are implied by assertion behavior.
- Stop the simulation or trigger evaluation based on the state transitions of assertions.

When an assertion check fails during simulation, the simulator generates an error. You can view the results in the SimVision waveform window.

Static ABV vs Dynamic ABV

Dynamic ABV--simulation--provides a familiar use model and can monitor any behavior in any design. A static tool, on the other hand, sometimes runs out of capacity. Also, both the design under test and the assertions associated with the property being checked must be synthesizeable. Formal analysis requires the ability to abstract a cycle-based model from the design, such that the cycle-based model will behave the same as the event-based model.

The challenge of formal analysis is to express assertions that can be proved in a reasonable amount of time, and to provide sufficient assumptions about the accepted input behavior.

Static ABV excels in verifying designs early in the design cycle, particularly modules and blocks written by designers. When used in the context of the ABV flow and methodology, formal analysis runs smoothly, and enables design and verification teams to improve productivity and quality with a minimal learning curve. As such, static ABV is fully complementary to dynamic ABV.

ABV for System Verification

The technique used here is acceleration or emulation. With acceleration, a non-synthesizable testbench is run in the simulator, and the design and associated assertions are synthesized and run in hardware. The performance is driven by how fast the test simulates and the amount of synchronization that is required between the test and the design. Assertions are run in hardware to avoid frequent synchronization with the simulator--communication between the simulator and hardware is transaction-oriented rather than signal-oriented.

In emulation, the test and design are synthesized into hardware. Emulation can also involve the use of dynamic targets, where the design is synthesized and subjected to the stimulus of a real environment.

System-level runs, or runs where the time it takes to bring up the software can be very long, use acceleration or emulation to complete within a reasonable time. Assertions are especially useful in long runs, because they can track errors close to their source, reducing potentially lengthy debug times.

When emulating a design in its real environment, it can be difficult to assess what is being tested.

Moreover, reproducing a bug can be very difficult. Assertions can isolate the problem so that it can be reproduced more easily. Coverage points help to determine what was tested or, more importantly, what was *not* tested.

Assertions for Acceleration

In general, any assertion that is appropriate for the simulator can also be used in acceleration. However, assertions must be synthesizeable to be used in acceleration/emulation.

Abbreviations Used in ABV

ABA	Acceleration-based assertions
ABV	Assertion-based verification
IAL	Incisive Assertion Library
ICE	In-circuit emulation
IFV	Incisive Formal Verification
IES	Incisive Enterprise Simulator
OVL	Open Verification Library
PSL	Property specification language
SVA	SystemVerilog assertions
TBA	Transaction-based acceleration
TLM	Transaction-level model

2

Introduction to Assertion Simulation

Assertions let you define the assumptions you made when coding your design. The simulator tests those assumptions, and generates an error if an assertion fails.

Using Assertions to Verify a Design

When you embed assertions in the design source code or place them in a separate property file, they are actively monitored during simulation.

You use an option to enable assertion checking when you compile the design. The elaborator then adds the assertions to the design snapshot.

Assertions behave like other design objects. You can set breakpoints on them or probe them to a simulation database. You can display them in the browser, waveform, and other simulation debugging windows.

The Incisive debugging environment, SimVision, also provides an Assertion Browser, which displays the current state of your assertions during simulation, as well as coverage and failure counts. You can use this information to focus on the areas of your design that need further testing.

The Incisive™ Enterprise Simulator-XL supports a subset of the IEEE 1850 property specification language standard, PSL. PSL comes in four flavors-Verilog, SystemVerilog, VHDL, and SystemC®. Incisive also supports a subset of the System Verilog Assertion (SVA) as defined in the IEEE 1800-2009 LRM.

The Incisive Debugging Environment for ABV

The *Assertion Writing Guide* introduced the PSL and SVA languages for writing assertions. This manual will describe how to use these assertions in your simulation environment.

Assertions are an integral part of the Cadence Incisive simulator. You can run your simulations with assertion checking enabled, either in batch mode or when using the debugging environment.

When working in the SimVision graphical debugging environment, you can:

- Easily see your assertion definitions

- Set breakpoints on assertion failures, to help debug design problems
- Probe assertions for viewing in the waveform window, so you can graphically view assertion failures in the context of the other activity in your system

ABV Documentation

For information that will help you quickly get started with assertion checking, see the following Cadence manuals:

- *Assertion Writing Guide*
  Describes how to write PSL assertions for Verilog, VHDL, SystemVerilog, and SystemC designs, and how to write SVA assertions for SystemVerilog designs.
- *Assertion Checking in Simulation* (This manual)
  Describes how to enable assertion checking, how to control it, and how to interpret the results.
- *SystemC Simulation Reference*
  Chapter 6, "Using SystemC PSL," describes how to use the SystemC flavor of PSL.
- *Incisive Assertion Library Reference*
  Describes the contents and use of the Incisive Assertion Library (IAL), a set of predefined verification modules provided to facilitate adoption of assertion-based verification.
- *ICC User Guide*
  Describes how to use PSL and SVA constructs for control-oriented functional coverage analysis.

For a quick-reference guide to Cadence assertion support, see the following Cadence manual:

- *Assertion Writing Quick Start*
  Describes the process of writing and simulating assertions in the Cadence Incisive simulator. Also includes printable quick reference pages for:
  - PSL, for use with Verilog, SystemVerilog, VHDL, and SystemC
  - SVA (SystemVerilog assertions)
  - ABV Tool Commands (Incisive simulator Tcl and GUI commands for assertion simulation)

3

Assertion Languages and Libraries

Assertion Languages

Cadence currently supports PSL and SVA languages.

The Property Specification Language

The property specification language, PSL, is the formal specification language selected by the Accellera Formal Verification Technical Committee as the basis for a standard property language. Using PSL, you can document constraints and specifications related to a design. The information is easily available to verification engineers, both for the initial design, and for later designs in which this design is embedded as IP.

To ensure consistency between PSL assertion statements and the design to which they apply, the PSL language uses the expression syntax and semantics of the design in which it is used; for example, Verilog expression syntax in a Verilog design. This feature ensures that the assertions can cover the full range of behavior that can be described in the HDL, and that there is no chance of semantic mismatch between the HDL description of a behavior and the PSL description of the same behavior. It also reduces the learning curve for being able to express a behavior.

Cadence supports the Verilog, SystemVerilog, VHDL, and SystemC flavors of PSL.

The PSL assertion language provides facilities that enable you to:

- Describe behavior patterns in terms of conditions that hold true in successive clock cycles
- Chain together simple behavior patterns to represent sequential activity
- Compose behavioral constraints to capture complex behavior specifications

Assertions can be embedded in the HDL design file, or can be provided in an external file.

You can use PSL to write assertions and coverage monitors. Both are descriptions of specific behavioral patterns, but assertions are expected to evaluate to true, and generate an error if they evaluate to false. If you probe coverage monitors, they record transactions in the database when they evaluate to true.

For more information about the PSL syntax, see the *Assertion Writing Quick Start* .

SystemVerilog Assertions

An assertion in SystemVerilog is a statement that says that the behavior that the assertion describes, must be true. That is, you cannot specify that a behavior will never occur. Assertions in SystemVerilog are composed of sequences of Boolean expressions with time steps between them.

SystemVerilog Assertions (SVA) differ from PSL by offering two different types of assertions:

- Immediate assertions
  An immediate assertion tests an expression when the statement is executed in the procedural code.
- Concurrent assertions
  Concurrent assertions describe behavior that occurs over time. They are evaluated only at the occurrence of a clock tick.

For more information about the SVA syntax, see the *Assertion Writing Quick Start* .

Assertion Libraries

Libraries provide a familiar use model that eliminates the need to understand the underlying assertion language. They can help you quickly adopt ABV by providing a familiar use model and real examples to learn from and adapt.

Incisive Assertion Library (IAL)

IAL provides many same library elements as OVL does, and also more complex elements. IAL implements the elements in both PSL and SVA. Each element contains assertion checks and coverage points.

You can obtain the Incisive Assertion Library from your Cadence installation at:

install_dir/tools/ial/

Note: This location also contains the OVL components.

Open Verification Library (OVL)

The Open Verification Library is an open-source library of Verilog modules and VHDL entities for common assertion constructs. OVL offers HDL and PSL implementations. OVL works in Verilog, SystemVerilog, and VHDL. Cadence provides OVL inside its Incisive Assertion Library. For more information, see "OVL Users" under "Instantiating the IAL Component" in "Assertion-Based Verification Using the IAL," in the *Incisive Assertion Library Reference*.

4

Compiling and Elaborating a Design with Assertions

When you use assertions in a design, assertion checking must be enabled for compilation and elaboration. During compilation, you use specific options to enable/disable assertion checking, which include:

- options to disable assertion checking for low power simulation
- disable assertion checking during elaboration and compilation
- enable synthesis pragma checking.

The following are examples of simple compilation and elaboration commands:

```
irun -assert -sv *.v *.vhd -top memtest2:rtl
```

For more information about compiling and elaborating a design, see the Incisive simulator documentation.

Enabling Assertion Simulation

If you want to enable simulation-based assertion checking, and view assertions in the SimVision waveform window, you must compile your design with the appropriate option, as follows:

	PSL	SystemVerilog
ncvlog	-assert	-sv, or .sv file extension
ncvhdl	-assert	N/A
irun	-assert	-sv, or .sv file extension

Enabling/Disabling Assertion Checking at Compile Time

The following command-line option given to the Incisive simulator lets you pass to the compiler, an assertion control file that enables/disables specified assertions:

`ncvlog`	`-controlassert` *`filename`*
`ncvhdl`	`-controlassert` *`filename`*
`irun`	`-controlassert` *`filename`*

This control file specifies which assertions to include in, or exclude from, the compilation. You can use more than one `-controlassert` option to specify multiple assertion control files.

The syntax of the assertion control file is as follows:

```
assertion -off | -on
     [-directive directive_type]
     [module_name.]assert_name
```

- *`directive_type`* is `assert`, `assume`, `cover`, or `restrict`.
- *`assert_name`* is the name of an assertion or a wildcard specification (`*` and `?` wildcards are supported). Unlike Tcl, the names cannot be hierarchical--the specified names and wildcards are matched during parsing, when the hierarchy and instance names are not known. However, names of the forms `assert_name` and `module_name.assert_name` are permitted. For example, either `in1` or `top.in1`, or a wildcard pattern for either kind of name, is accepted. Ordinary names, such as `in1`, are treated as assertion names, not module names.
- Tcl comments are supported.

For example:

```
# Disable all assertions except assertions in module1:
assertion -off *
assertion -on module1.*
# Disable prop1 wherever it occurs:
assertion -off prop1
# Disable all cover directives in module1:
assertion -off -directive cover module1.*
```

Enable/disable lines are executed in the order in which they appear in the file ("last one takes

precedence").

> The `-on/-off` options must be specified first on the command line. If an `assert_name` is specified, it must appear last on the command line.

The `controlassert` option provides a placeholder in the database so that the aggregate coverage is possible across runs with different assertions turned on/off.

Notes about using the `controlassert` option:

- The `-controlassert` command does not apply to immediate assertions.
- Disabled assertions are shown with an Off status in the Design Browser, Source Browser, and Assertion Browser windows.
- The assertion summary does not show assertions that are turned off at compile time.
- Elaboration hierarchy data shows the disabled assertions in the design.
- Tcl commands return an "assertion not found" message for disabled assertions.
- Disabled assertions are reported with 0 coverage in the coverage database for that test run.
- Assertions specified in the assertion control file that are not found in the design are silently ignored.
  This feature lets you use one master assertion control file when compiling files separately.
- Syntax errors in the assertion control file are reported in the compile log file, but compilation continues.

Disabling Assertion Checking at Elaboration Time

The following option lets you turn off assertion checking in the elaborator, without needing a full recompilation. It disables all run-time assertion commands:

`ncelab`	`-noassert`

Disabling Assertion Checking for Low Power Simulation

For low power simulation, the Cadence Common Power Format supports the `create_assertion_control` command, which turns off assertion checking in a power domain that is shut down.

For more information about low power simulation, see the *Low Power Simulation User Guide*.

Enabling Synthesis Pragma Checking for ABV

Synthesis pragmas are comments that make claims about some aspects of the functionality of an HDL design. Synthesis tools assume that these pragmas are correct and optimize the synthesized implementation accordingly.

To ensure that synthesis pragmas are correct, you can specify that the Incisive parsers convert synthesis pragmas to implicit PSL assertions.

You can convert certain synthesis pragmas to assertions for simulation (for details, see "How Synthesis Pragmas Convert to PSL " in the *Assertion Writing Guide*).

Note: Synthesis pragmas are not permitted in verification units.

Supported pragmas are `one_hot` and `one_cold` in `cadence`, `synopsys`, and `ambit` forms for Verilog ports; and `full_case` and `parallel_case` in `cadence`, `synopsys`, and `ambit` forms for Verilog case statements. Pragmas do not work for internal signals. Examples of synthesis pragmas are the following, where the entry in double quotes contains the names of the signals to be checked:

```
// cadence one_hot "set, reset"
// ambit synthesis one_hot "set, reset"
// synopsys one_hot "set, reset"
```

If you want to turn on conversion of synthesis pragmas, you must compile your design with the following option:

`ncvlog`	`-genassert_synth_pragma`
`irun`	`-genassert_synth_pragma`

Note: The `-noassert_synth_pragma` option is still supported for backwards compatibility, but it is no longer needed--by default, synthesis pragmas are not converted to assertions.

In SystemVerilog, using `-genassert_synth_pragma` translates synthesis pragmas to PSL and checks them in simulation. The clocking of these internally-generated assertions is based on the default PSL clock. If you do not have a clock defined, the pragmas are checked every simulation cycle, and can be subject to false failures caused by transient signals. To ensure proper clocking of the translated pragmas, you can define a default clock in PSL, as described in "Declaring Default Clocks in PSL" in the *Assertion Writing Guide*.

Setting Debugging Access to ABV Design Objects

Besides enabling/disabling assertion checking during compilation and elaboration, you can set debug access levels for ABV design objects. Generally, the elaborator marks all simulation objects in the design as having no read, write, or connectivity access by default. When you run in regression mode, this access setting improves simulator performance. However, if you interactively debug your design, you might need read, write, and connectivity access to all of the signals. You specify access to design objects as shown in the table that follows.

`ncelab`	`-access rwc`
`irun`	`-access rwc`

Read access is enabled by default for assertions.

Specifying an Assertion Property File

When compiling a design, you can also create and use assertions that are contained in an external file. There are two approaches for attaching assertions to a design:

- PSL defines a property file, which can contain SVA and PSL.
- SystemVerilog defines a `bind` file, which can only contain SVA.

This section discusses the semantics of property files, and the next section discusses binding files.

For more information, see these topics in the *Assertion Writing Guide*:

- "Putting PSL Assertions in Verification Units ," in "Using PSL"

- "Binds in an External Text File ," in "Using SVA"

For important information about using property and `bind` files, see "Using an Assertion Property or Binding File".

Using any of the PSL-specific property file options--`-propfile`, `propdir`, and `propext`--as well as the `-assert` option, sets the following compiler directives, which might affect OVL simulation:

- `-DEFINE ASSERT_AL_INUSE`
- `-DEFINE ASSERT_OVL_PSL`

Specifying the Names of Property Files

If you plan to supply assertions in a property file, you need to specify the filename as follows:

ncvlog	-propfile *filename1* [-propfile *filename2* ...]
ncvhdl	-propfile *filename* [-propfile *filename2* ...]
irun	-propfile_vlog *filename1* [-propfile_vlog *filename2* ...]
	-propfile_vhdl *filename1* [-propfile_vhdl *filename2* ...]
	-propfile_sc *filename1* [-propfile_sc *filename2* ...]

- `-propfile_sc` is not supported with the `-noedg` option.
- The error messages logged during the compilation of `-propfile` continues to be available in `stdout` and is also saved in the current log file, such as `ncelab.log`, `ncsim.log`, or `irun.log`. `vuibvlogparse.log` file is no longer created.

When you specify the property file name on the irun command line, irun uses the file extension to identify the type of file. By default, the recognized files extensions are `.pslvlog` for a Verilog vunit file, `.pslvhdl` for a VHDL vunit file, and `.pslsc` for a SystemC vunit file. For example, the given Verilog vunit file has a valid file extension:

```
irun *.v memctl_vunit.pslvlog ...
```

See the *irun User Guide* for information about how to change or add to the list of valid recognized file extensions.

Options `-propfile_vlog`, `-propfile_vhdl` and `-propfile_sc`, given to irun outside a `-makelib/-endlib` pair, would not be applicable to the files being compiled inside `-makelib/-endlib` pair.

To bind a vunit to a module inside `-makelib/-endlib`, specify the `-propfile_vlog/-propfile_vhdl/-propfile_sc` inside same `-makelib/-endlib`.

Example:

```
  irun -assert -makelib lib lib.v -propfile_vlog lib.psl -endlib test.v -propfile_vlog test.psl -input sim.tcl
```

Here, vunit(s) in `lib.psl` can only be bound to module(s) in `lib.v`, while vunit(s) in `test.psl` can only be bound to module(s) in `test.v`.

Note: If your PSL property file contains a `vunit` that binds to an instance, you must specify the property file when you compile the topmost module referenced by that instance. This permits the compiler to locate the instance specified in the binding. For example, if a `vunit` in `prop.psl` binds to instance `top.ex1`, the property file should be specified in the topmost module as:

```
ncvlog -propfile prop.psl top.v
```

Specifying Multiple Assertion Property Files

If you are working with a design that uses a large number of property files, the following parser options make specifying specific property files easier:

- `propdir`--Search the specified directory for property files. The directory name can be a

relative or absolute path. There is no limit to the number of `propdir` options you can specify.

- `propext`--Search for property files that have the specified file extension. The file extension you specify is used to match the names. A leading "." is inferred if you do not include it. The default extension of the property file is `".psl"`. There is no limit to the number of `propext` options you can specify.

ncvlog	-propdir *propfile_dir1_path* [-propdir *propfile_dir2_path* ...] -propext *file_extension*
ncvhdl	-propdir *propfile_dir1_path* [-propdir *propfile_dir2_path* ...] -propext *file_extension*
irun	-propdir *propfile_dir1_path* [-propdir *propfile_dir2_path* ...] -propvlog_ext *file_extension* -propvhdl_ext *file_extension*

Specifying an External Binding File

SystemVerilog specifies a `bind` construct that enables you to instantiate a module, interface, or program into a target module or interface without changing the code. Bind statements can be placed in the code, or they can be provided in an external file that is specified at elaboration time.

If you are using an external file to specify `bind` statements, you must pass the name of the external file using the `-extbind` *`filename`* option:

ncelab	-extbind *filename*
ncvhdl	-extbind *filename*
irun	+extbind+*filename*

For more information, see "Binds in an External Text File ," in Chapter 7, "Using SVA," of the *Assertion Writing Guide*.

For important information about using `bind` in a external file, see "Using an Assertion Property or Binding File".

Using an Assertion Property or Binding File

The PSL `vunit` implementation and the SVA `bind` implementation are tightly coupled in the Incisive simulator. The SVA `bind` construct automatically turns on the `+assert` compile option that enables `vunit` processing. Likewise, PSL `vunit` instance binding automatically turns on the SystemVerilog `+sv` compile option when you specify an instance as the `design_unit`.

You need to consider the following when using an external property or binding file:

- When binding a PSL `vunit` to an instance, the design must be compatible with SystemVerilog.
  When you use instance binding, signal names must not use SystemVerilog keywords. This limitation does not apply to module binding.
- Using the SVA bind construct enables
  - Any PSL pragmas in the design--that is, PSL embedded in the design with `// psl`
  - The PSL-based synthesis pragma processing

Allowing Unused Property Checking

For performance reasons, when your design contains PSL or SVA properties that are not used or asserted, you cannot use `assertion -on` later to assert the property at run time.

If you want to use these properties later, you must specify the following elaboration option:

`ncelab`	`+allow_unused_properties`

Note: You can list unused properties in the Assertion Browser by using the Tcl `assertion -on` *`property_name`* command. Unused properties are listed as Disabled.

Incremental Elaboration of Designs Containing Assertions

To reduce the elaboration time when re-elaborating a design, IES provides an incremental elaboration flow feature. This feature enables multi-snapshot incremental elaboration, which provides the ability to break a design into two sections that are elaborated separately and combined at simulation time. These two sections are elaborated as two snapshots, a primary snapshot and an incremental snapshot. In this, the primary snapshot contains the portion of the design that is stable and unchanging, and the incremental snapshot contains the portion of the design that is less stable.

For more information about incremental elaboration, refer to the Multi-Snapshot Incremental Elaboration d ocumentation.

Limitation of Design Containing Assertions

If the assertion completely resides in either of the two snapshots, the incremental elaboration works as expected. However, if the binding specified in the incremental snapshot relates to the module/instance in the primary snapshot, then incremental elaboration will not work as expected and will require re-elaboration of the primary as well. This is because in case of SV bind a second compilation is done from `ncelab`.

5

Assertions Simulation Semantics

To prevent race conditions, the Incisive simulator divides a time slot into a set of ordered regions. This division is intended to provide predictable interactions between design and testbench code.

Supported Scheduling Regions for ABV

The simulator assertion facility supports the following regions that are used by SystemVerilog assertions:

- Preponed region
  The values of variables used in a concurrent assertion are sampled in the Preponed region, before signal assignments.
- Processing region
  Processing in the Active, Inactive, and NBA regions continues in a loop until all value changes are processed.
- Observed region

  When property expressions in assertions are triggered, they are evaluated in the Observed region. To emulate hardware processing, the signal values that were sampled in the Preponed region are used, which prevents race conditions that might otherwise occur when clock and data change at the same time. Current values are used for local variables. For example:

  ```
  @(posedge clk) clk == 0
  ```

will always be true, because `clk` is sampled before the `posedge clk` event is detected.

During evaluation in the Observed region, pass/fail code and subroutines/tasks on sequence match are scheduled in the Reactive region of the current time slot.

- Reactive region

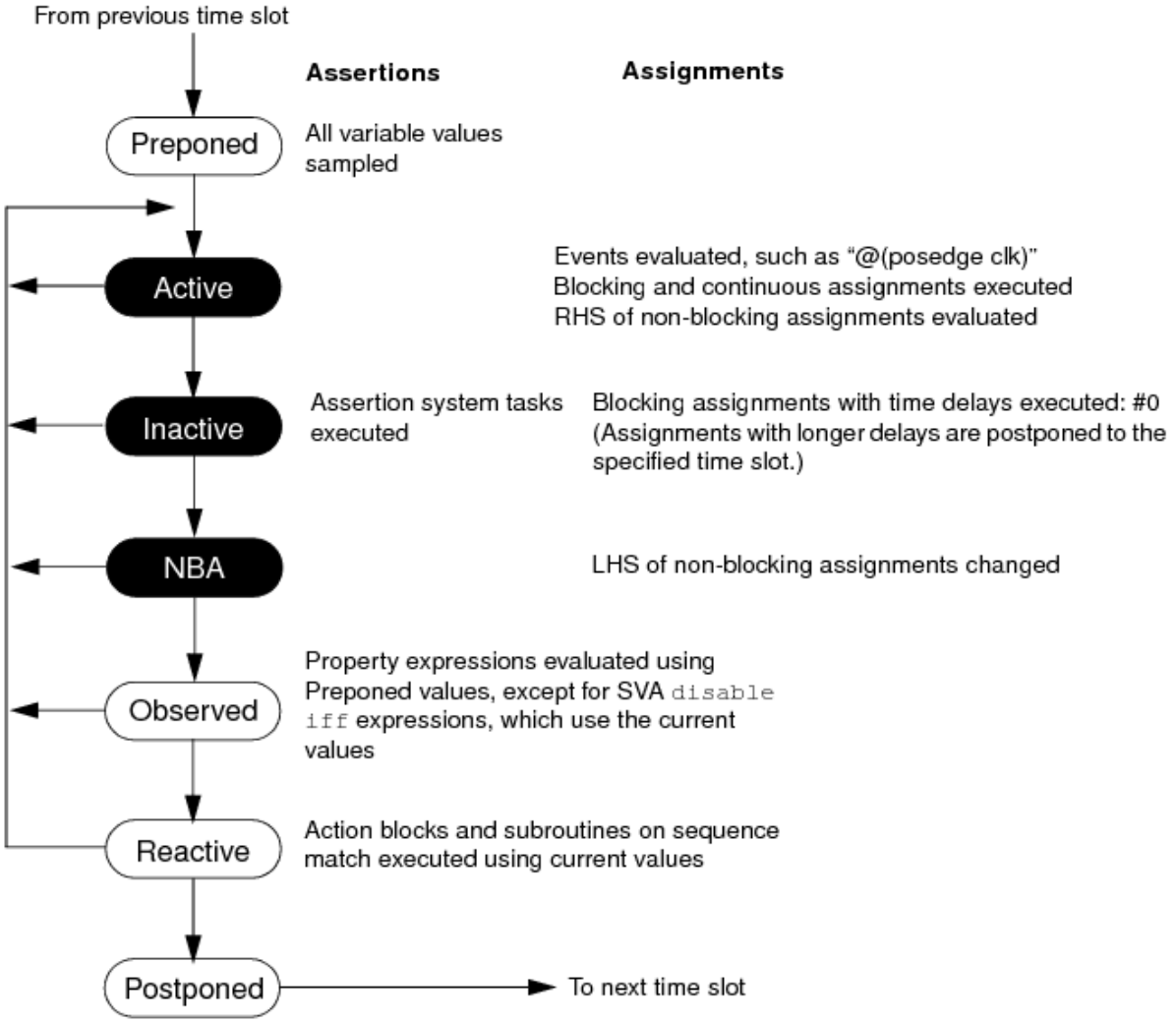
  The Reactive region uses the results obtained in the Active, Inactive, and NBA regions (Figure 5-1).

  The following are executed in the Reactive region:

- Action blocks (reference current values)
- Subroutines on sequence match (reference sampled values of signals and current values of local variables)

The Preponed values are used to evaluate clocked assertions, but action blocks are executed using the changed values. If you print the values of assertion signals in the action block, you get the changed value, not necessarily the value used in the assertion evaluation.

Figure 5-1 Scheduling Regions



Assertion Scheduling Example

The following example demonstrates how values are evaluated in the scheduling regions.

For this example, the following processing occurs during time slot 5:

- Line 09--The clk signal goes to 1, which produces a posedge clk event.
- Lines 10-11--Responding to the posedge clk, a=1 and clk1=1, which produces a posedge clk1 event.
- Lines 17-18--Responding to the posedge clk1 event, clk2=1 and b=1.
- Lines 21-22--This assertion will always be true, because clk2 and b are the same initially, and they toggle together. When clk2 rises, the Preponed value of clk2 and b is 0.

```
01 module test
02     reg a=0;
03     reg b=0;
04     reg c=0;
05    reg clk=0;
06    reg clk1=0;
07    reg clk2=0;
08
09     always #5 clk = ~clk;
10
11 always @(posedge clk) begin
12     a = ~a
13     clk1 = ~clk1
14 end
15
16 always @(posedge clk1) begin
17     clk2 = ~clk2;
18     b = ~b;
19 end
20
21 simult_clk: assert property
22 @(posedge clk2) (b==0);
```

Assertion Reset Expressions in Scheduling Regions

A property might include a `disable iff` or `async_abort` reset expression. When the reset expression is tested during property evaluation, the values of the variables in that reset expression are the values they have in the current simulation cycle. They are not sampled values.

Assertion Variables as Clock Expressions

If you use a variable to clock an assertion, and use the same variable within that assertion, the variable can have two different values during an evaluation. For example:

```
onehot0_sel: assert property (@(select) $onehot0(select));
```

The property is evaluated in the Active region when `select` changes from `` `b10 `` to `` `b11 ``. This evaluation uses the sampled value of `select` from the Preponed region, which is `` `b10 ``, so the assertion evaluates to True. Similarly, the assertion will evaluate to False when `select` changes from `` `b11 `` to `` `b10 ``.

Scheduling Regions Summary

To summarize:

- Assertions are evaluated in the Observed region using the data values from the Preponed region of the current time slot.
- Action blocks and sequence match items execute using the values in the Reactive region.
- Assertion system tasks--`$assertoff`, `$asserton`, `$assertkill`--are evaluated before assertions are evaluated.
  - If `$asserton` occurs at time 50, the first attempt of the assertion is at time 50.
  - if `$assertkill` occurs at time 50, assertion results at time 50 are disabled at time 50.
- The `disable iff` and `async_abort` expressions are evaluated in the Observed region using current values.

More Information about Scheduling Regions for ABV

For more information about scheduling regions, see the following sections in the IEEE 1800 SystemVerilog standard:

- 4.4 "Stratified event scheduler"
- 16.5 "Concurrent assertions overview"

6

Setting Up an ABV Simulation Run

There are several different ways to control and query assertion checking in the Incisive simulator before starting a simulation run. Then when you enable assertion processing and begin the simulation, you can use SimVision to interactively debug your design.

The SimVision analysis environment supports assertions in the Source Browser, Register window, Design Browser, and Trace Signals sidebar. It adds assertions to the Set Breakpoint, Probe, Force, and Deposit facilities, and it makes an Assertion Browser window available.

Note: The Cycle View and Schematic View windows do not support assertions.

Using Assertion Names in Tcl Commands

During simulation, you will need to refer to your assertions by name when you use the Cadence simulator Tcl commands to query your assertions.

When referring to assertions in the simulator Tcl commands, you can use the following forms:

- An assertion name--An absolute or relative path to the property
- A scope--An absolute or relative path to the design scoping construct
- Wildcard names--Standard Tcl wildcards

  Note: You cannot use wildcards for the hierarchy.

Setting Up Assertion Probes

It is not necessary to always probe assertions, because failures will be reported regardless of whether or not the assertion is probed. However, if you want to view assertions in the SimVision waveform window for debugging, you need to probe them. You must also probe them to see assertion counts in the Assertion Browser at times other than at the end of simulation, or in post-processing mode (Chapter 10, "Post-Processing Assertions").

When you probe an assertion, a database is automatically created to store the information.

If you view assertion probes in the SimVision waveform window, you might also want to probe the signals that are referenced by the assertions. If the assertion signals are not probed, the waveform window will not show any data during simulation. For more information about probing the signals that contribute to an assertion, see "Probing Signals Referenced by Assertions".

Tcl Command to Create Assertion Probes

```
probe -create [-transaction] -assertions [type]
            [-signals] [options] assertion_locator
```

You can probe an assertion as a state change, transaction, or failure event so the simulator will record its state changes, transactions, or failure events in the simulation database.

To probe an assertion, use the `probe -create` Tcl command with the `-assertions` option in the command-line interface or a Tcl file. This option specifies that the object name that you supply refers to an assertion rather than a signal. If you supply a scope name, all assertions within the scope are probed.

- `-transaction`

  Creates a transaction probe; see "Creating Assertion Transaction Probes using Tcl".

- The *`type`* specifies how to record the probed assertion--as state changes or failures. The type of assertion probe to create can be

  - `-state`

    Creates a state probe. This option is the default if no other options are given. This is the *`type`* you need to specify if you want to see the statistic counters at all times, during the simulation or in PPE mode.

It is an error to use `-state` with `-transaction` or `-failure` .

  - `-failure`

    Creates a failure probe; see "Creating Assertion Failure Probes using Tcl".

  - No `-failure` or `-transaction` option

    If you do not include the `-failure` or `-transaction` option with `-assertions`, the default is to create a state probe. If you use the `-signals` option alone without including a

type specification, a state probe is created. See "Creating Assertion State Probes using Tcl". For example, the following creates a state probe for the CHECK_X assertion in the waveform window, and also displays all signals referenced by CHECK_X:

```
probe -create -assertions -waveform -signals memtest2.mctl.mem8x256.CHECK_X
```

- `-signals`

  Probes the signals that are referenced by the specified assertion; see "Probing Signals Referenced by Assertions".

- The following *options* have been modified for assertion checking to apply to both failure probes and transaction probes:
  - `-database name`

    The database must be opened with the `-shm` option, or the `$shm_open()` system task, as described in "Using Tcl to Add Assertions to a Database Opened with $shm_open()".
  - `-depth levels | to_cells | all`

    If a scope is given, specifies how many scope levels to descend when searching for assertions to process. The recursion depth is the same as for the `probe` command for signals.
    - The *levels* argument is an integer that specifies the number of scope levels to descend. The number 1 includes only the given scope, 2 includes the given scope and its subscopes, and so on. The default is 1.
    - The `all` keyword includes all scopes in the hierarchy below the given scope. Using this option creates state probes for all PSL and SVA assertions in the design. For example, the following records all state probes to the database without also saving the associated signals:

      ```
      probe -create -shm -assertions -depth all
      ```

This command is useful when you want to see the counter values in the waveform viewer, or at any time in the Assertion Browser. It also enables the counts to be visible in post-processing mode. Using the `-depth all` option can result in thousands of assertions being probed to the database. To reduce the volume of data that is displayed, the waveform window displays only the assertions of the scope in which the `probe` command was issued.

- The `to_cells` keyword is the same as `all` except that the descent stops at cells.

- `-name` *`name`*

  The *`name`* is a name of your choice for this probe. This name is created in the scope where the `probe` command was executed, and it can be displayed in the SimVision waveform window.

  - If you do not supply a name for a failure probe into multiple assertions, a default name of `fail_probe_`*`n`* is assigned, where *`n`* is an integer number that is incremented for each new probe.
  - If you do not supply a name for a failure probe into a single assertion, a default name of `assertionname_fail_probe_`*`n`* is assigned, where *`n`* is an integer number that is incremented for each new probe.
  - If you do not supply a name for a transaction probe, the default name is `tr_`*`assertionname[n]`* , where *`[n]`* is a substream number.

For example, the following Tcl command sets a failure probe on an assertion called `ASSERT_WIDTH_MAX_CHECK` in the `top.soc.duv0.core.mii_inport0` scope and gives it an explicit name--the trace will be called `Inport0_Mii_Clk_High_Max` in the waveform window.

```
ncsim> scope top.soc.duv0.core.mii_inport0

ncsim> probe -create -assertions -failure -name Inport0_Mii_Clk_High_Max \

       -shm mii_clk_high.ASSERT_WIDTH_MAX_CHECK
```

Because the `-assertions` option restricts the probe to acting on assertion objects and scopes, the following probe `-create` options are illegal when you use the `-assertions`, `-failure`, or `-transaction` modifiers:

`-all`	`-inputs`	`-outputs`	`-ports`
`-screen`	`-variables`	`-vcd`	`-evcd`

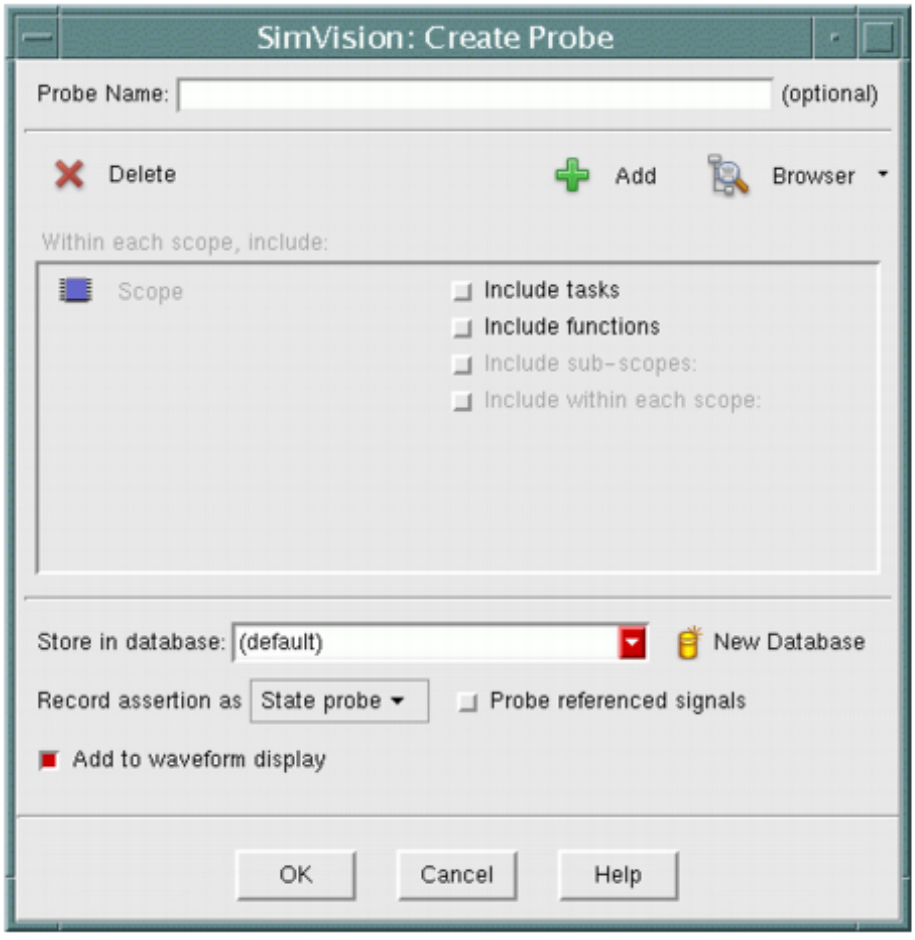
SimVision Commands to Create Assertion Probes

You can create a probe from the Source Browser or Assertion Browser, as follows:

1. Select the assertion(s) you want to probe.

2. Press the right mouse button on the selected assertion and choose *Create Probe* from the pop-up menu, or choose *Simulation - Create Probe* from the menu bar.
   The Create Probe form is displayed (Figure 6-1).
3. Optionally, in the Create Probe form, type a name for the probe in the *Probe Name* field.
4. Set the *Record assertion as* option to *State probe* or *Failure event* or *Transactions*.
5. To view the probe in the waveform window, enable the *Add to waveform display* button, if it is not already enabled.
6. To probe the signals that contribute to an assertion failure event or transaction probe, enable the *Probe referenced signals* button.
7. Click *OK*.

Figure 6-1 Create Probe Form



Creating Assertion State Probes using Tcl

```
probe -create -assertions assertion_locator
```

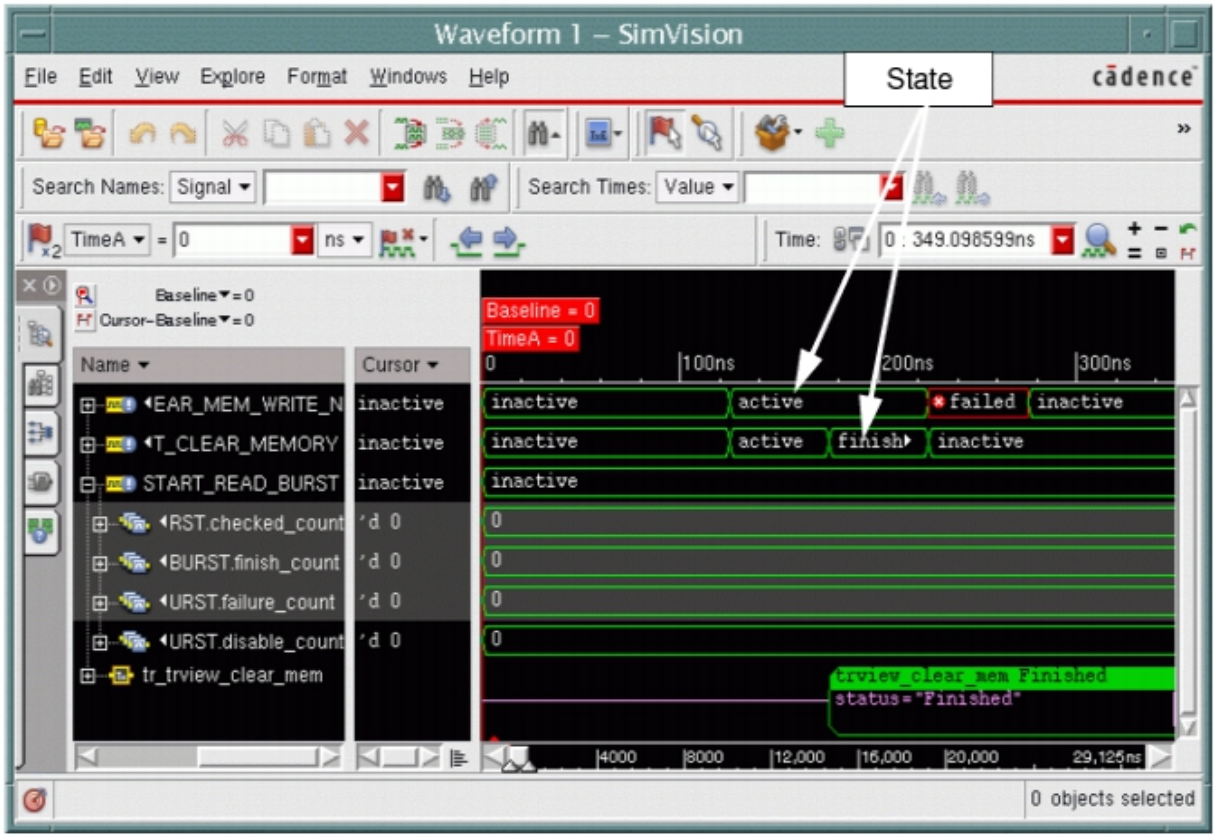
A state probe is similar to a signal probe: It records changes in the assertion state ("Understanding Assertion States") for the probed assertion. This type of probe is the default. The following example creates a single waveform trace on the `triggerRW` assertion within the `ctr` module:

```
probe -create top.ctr.triggerRW
```

Creating Assertion State Probes using SimVision

You can use SimVision to create state probes, as described in "SimVision Commands to Create Assertion Probes". If you do not specify a name for the probe, the assertion name is used. The traces created by assertion state probes are shown in Figure 6-2.

Figure 6-2 Assertion State Probes



Creating Assertion Failure Probes using Tcl

```
probe -create -assertions -failure assertion_locator
```

The `-failure` option specifies that you want to record an event in the simulation database whenever the fulfilling condition of the assertion fails. When you record assertions as failure events, the waveform in the SimVision waveform window shows a failure event marker and a counter that increments every time the assertion fails (Figure 8-10). This feature lets you see an assertion failure relative to other signals, as well as how many times it has failed during the simulation.

To reduce the amount of data that you have to analyze, you can use one assertion failure probe to record all of the failures from multiple assertions into one error counter. You can probe all failures into one counter, or identify the main blocks of your design and group the counters by the structure and hierarchy.

The following example records all assertion failures in the `mii_inport0` scope into one counter:

```
probe -create -assertions -failure scope top.soc.duv0.core.mii_inport0
```

A single probe is created that can be viewed in the waveform window or Design Browser, but does not appear in the Assertion Browser.

To create a single waveform trace to probe all assertions in the design, use the following command:

```
probe -create -assertions -failure -depth all
```

For each `probe` command, a trace is added to the waveform database, which creates a single trace on the waveform display. All assertions referenced by this probe are recorded on the same trace using sequence time, which enables multiple events to be recorded in the database at any given time or delta cycle. The assertions are listed in the order in which they were added to the database.

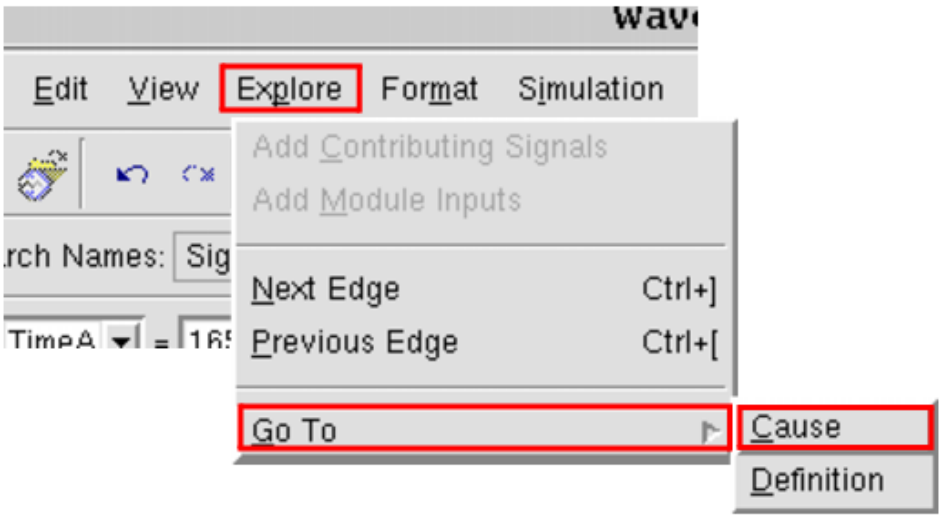
You can select the assertion failure probe in the waveform window, then use the *Next Edge* button to go to the first failure. When you choose *Explore - Go To - Cause*, SimVision pops up the Source Browser with an arrow pointing to the failing assertion.

Using sequence time, you can debug delta cycle delay assertion failures if you also probe the signals referenced in the assertion using sequence time. When you select a failure event, the corresponding sequence time values of all probed signals are also selected.

Probing All Assertions to a Single Failure Probe

1. Choose *Edit - Select All* in the Assertion Browser.
2. Press the right mouse button on one of the selected assertions.
3. Choose *Create Probe* from the pop-up menu.
4. Type a name for the probe in the *Probe Name* area.
5. Set the *Record assertion as* option to *Failure event*.
6. Run the simulation.
7. Select the failure probe in the waveform window.
8. Click the blue *Move primary cursor to next edge* arrow to move the cursor to the failure of interest.
9. Choose *Explore - Go To - Cause* (Figure 6-3).

Figure 6-3 Explore - Go To - Cause



For more information about tracing a failing assertion back to its source, see "Viewing Assertions in the Trace Signals Sidebar".

Creating Assertion Transaction Probes using Tcl

```
probe -create -transaction -assertions assertion_locator
```

When you specify the `probe -transaction` command with the `-assertions` option, assertion objects are recorded as transactions.

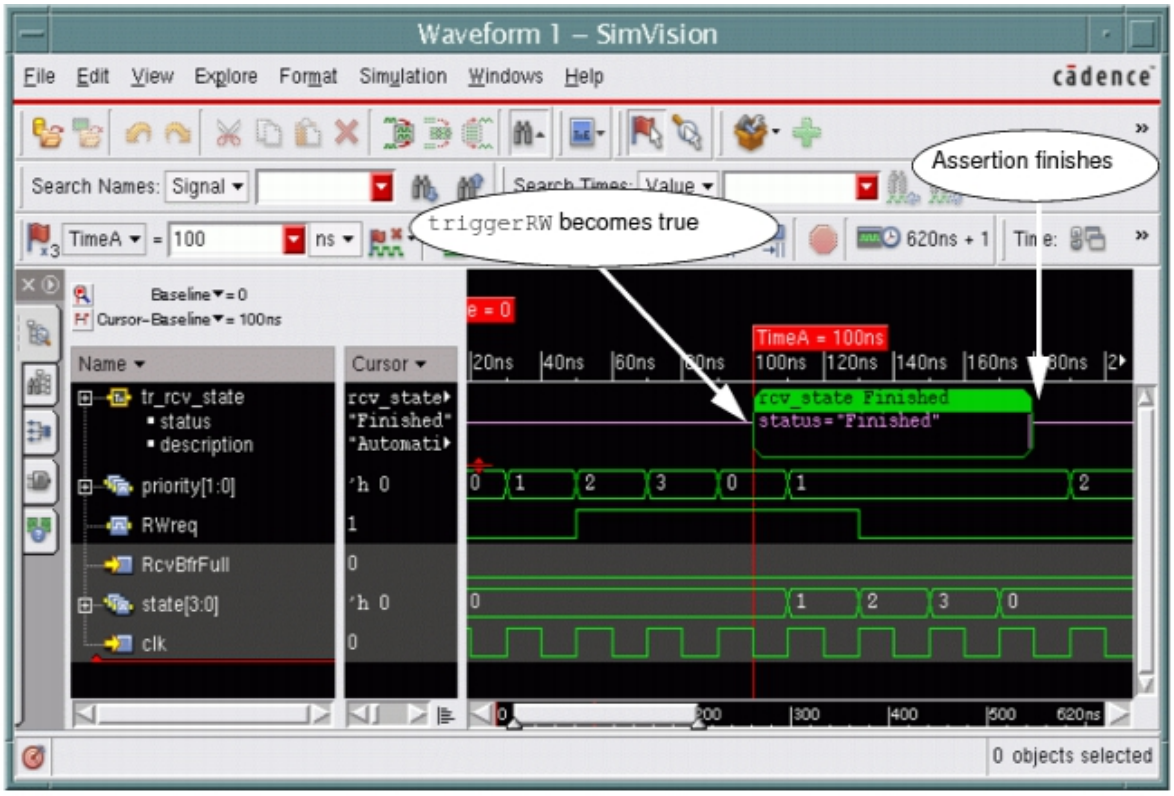
A transaction is a specific sequence of control and data signal activity. A transaction typically has start and end time attributes, and an attribute that specifies the kind of operation involved. You can record sequential assertions as transactions in the simulation database using this Tcl command.

When you record assertions as transactions, the waveform window shows the assertion as a stream with transactions. The default name of the stream is `tr_`*`assertionname`*. When a transaction is recorded, the beginning of the transaction corresponds to the first cycle in which the enabling condition becomes true. The end of the transaction shows when the assertion finished or failed.

For example, Figure 6-4 shows how this assertion transaction is displayed in the SimVision waveform window.

```
// psl default clock = (negedge clk);
// psl sequence triggerRW = { priority == 0 & RWreq & !RcvBfrFull };
// psl property rcv_state = always { triggerRW } |->
//   { state == 0; state == 1; state == 2; state == 3; state == 0 };
```

Figure 6-4 Assertion Transaction in SimVision Waveform Window



The following transaction attributes are displayed:

- Its status--Either `Finished` or `Failed`; only assertion transactions that have ended are displayed
  For example, the start of the following assertion transaction is at cycle 1, when `P` goes high:

```
// psl assert always {P; Q; R} |=> {V; W};
```

This assertion can end in cycle 4 if `V` is not high, or in cycle 5 if `W` is not high, which is an assertion failure, or in cycle 5 when `W` goes high.

- Its description--"Automatically generated from assertion"

Probing and recording assertions as transactions lets you see when the assertion was active relative to other signals, which you can use to visually debug the design in the SimVision waveform window. In addition, you can search for specific transaction data by using the Transaction Explorer.

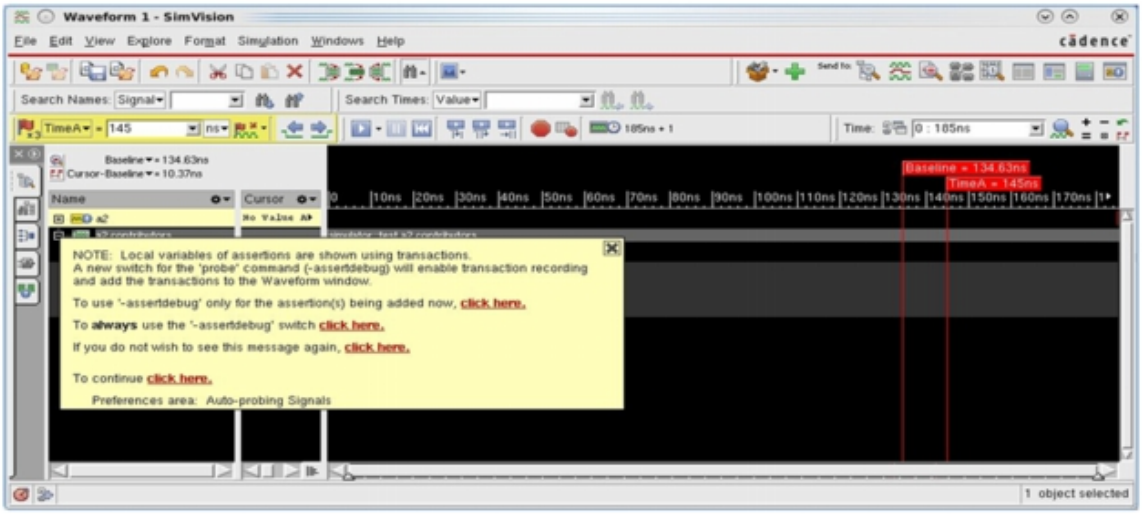
Note: For more information about transaction display in the waveform window, see "Viewing Transaction Probes versus State Probes for Assertions" and "Overlapping Assertion Transactions".

Local Variables recorded as Transactions

To enable local variables debug, the following sub command is added to the probe command 13.2 release onwards:

```
probe -create -assertions -assertdebug test.a1
```

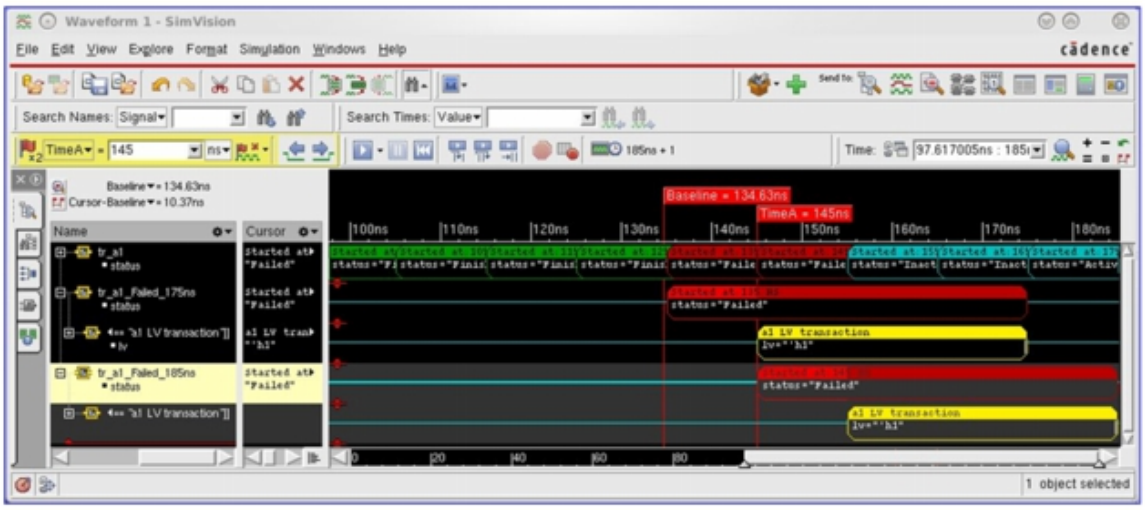
In case the option is not used, a pop up comes up in the GUI window informing you about the same.



The assertion failure hyperlink will have a preference available that enables the assertion that failed

to be opened in the debugger section of the Assertion Browser. Clicking on the assertion name in the failure message will perform the action.

The assertion in question will also be able to be added to the debugger directly using a Right-Mouse-Button (RMB) menu selection from the other windows where assertions can be displayed. The menu entry will be "Debug Assertion". With `probe -create -assertions -assertdebug test.a1' assertions and local variables are recorded and shown as transactions.



Probing Signals Referenced by Assertions

When you probe an assertion, you can use the `-signals` option to probe the signals referenced by that assertion. For example, for the following assertion:

```
// psl sequence triggerRW = { priority == 0 & RWreq & !RcvBfrFull };
// psl rcv_AdrsSel: assert always { triggerRW } |-> { true; AdrsSel == 0[*4] };
```

You can probe its referenced signals by using this Tcl command:

```
probe -create -assertions -failure -signals top.ctr.rcv_AdrsSel -database waves
-waveform
```

If you are using SimVision, you can use the *Probe referenced signals* option on the Set Probe form to probe these signals.

Understanding Assertion States

At any time during simulation, an assertion has one of the following states:

`active`	An assertion attempt is in progress. The first expression of the enabling condition is satisfied, and the assertion has not finished or failed.
`disabled`	The property is disabled by an SVA `disable iff` expression or a PSL `abort` expression.
`finished`	The fulfilling condition for the assertion has evaluated to true.
`inactive`	The assertion is being monitored, but there are currently no partial matches of the sequence of conditions described by the assertion. **Note:** This state applies to sequential assertions only. Boolean and unclocked assertions are either finished or failed.
`off`	The assertion is not being monitored due to external controls: a Tcl disable command, a compile-time disable (`-controlassert`, `-noassert`), a VPI control, or by `$assertoff` or `$assertkill`.
`suspended`	The assertion is located in a power domain that is shut off. The assertion retains its state at shutoff, unless a low-power reset resets its state.
`flushed`	The assertion is flushed after either a process that was earlier suspended resumed execution on reaching an event control or wait statement or a process that was declared by an `always_comb` or `always_latch` resumed execution due to a transition on one of its dependent signals.

You can use this information to focus on the areas of your design that need further testing. You can find out how many of your assertions finished or failed by using the Tcl `assertion -summary` command, as described in "assertion -summary".

The simulator cannot report the state of a given assertion in the first cycle in which its specified behavior begins. Instead, the assertion state is reported at the end of the behavior, in the cycle in which the state is determined.

For attempt-based counting, the simulator reports and displays one state per cycle for an assertion that is simultaneously in more than one state. The state precedence is, from highest to lowest, `failed`, `finished`, and `active`. The state goes to `disabled` when all attempts go to `disabled`; similarly for `off` and `suspended`.

When a property can fail or finish in the same cycle for two or more reasons, the Incisive simulator reports only the longer, earlier-starting reason. This is consistently reflected in the log file, the Assertion Browser, and the waveform database.

For trace-based counting, the failure is reported if a property can both finish and fail at the same time, because assertion-based verification is primarily about finding failures.

For definitions of enabling, fulfilling, and discharging conditions, see Enabling, Fulfilling, and Discharging Clauses in "Basic Assertion Concepts," of the *Assertion Writing Guide*.

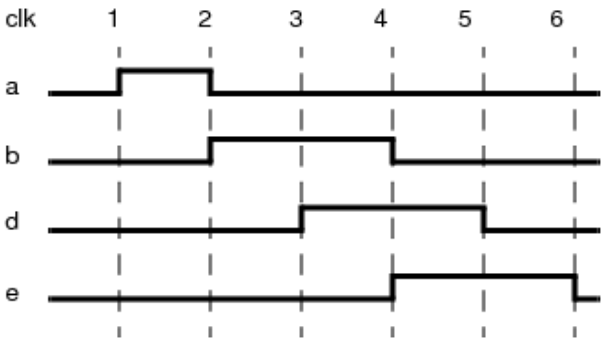
Setting Up the Assertion Statistics Counters

By default, the Incisive simulator provides counters that tell you the number of times an assertion finished, failed, or was disabled in some way. Counters apply to both PSL and SVA concurrent and immediate assertions. Although additional SVA counters are also provided with the `-strict` option--attempts, passes (finishes + vacuous passes), vacuous passes, and disabled assertions--The `-counter` option is the preferred method to print these statistics.There are two approaches that can be used for reporting statistics. One is trace-based counting and the other is attempt-based counting.

Attempt-Based versus Trace-Based Counting

For an assertion such as the following, the Incisive simulator lets you specify how to report the Finished and Failed states:

```
@(posedge clk) (a ##1 b[*1:2]) |=> d ##1 e
```



- Attempt-based--This is the default. From the start of any one clock, this assertion can reach the Finished state a maximum of one time. The implementation of this assertion is equivalent to the following:

```
@(posedge clk) ((a ##1 b) |=> d ##1 e) and ((a ##1 b ##1 b) |=> d ##1 e)
```

For this example, the assertion finishes at cycle 6.
The Finished state indicates that the assertion attempt has reached a state from which it cannot fail.
Multiple finishes or failures from a single attempt are not counted or reported.

- Trace-based--From the start of any one clock, this assertion can reach the Finished state a maximum of two times:

```
@(posedge clk) (a ##1 b) |=> d ##1 e
@(posedge clk) (a ##1 b ##1 b) |=> d ##1 e
```

For this example, the assertion finishes at cycles 5 and 6.
The Finished state indicates the successful completion of a trace from the start to the end of the assertion.

For an example with a range on the right side of the assertion:

```
$rose(x) |=> [0:5]y
```

- Trace-based--When x rises twice before y is asserted, the finished count will increment once.
- Attempt-based--When x rises twice before y is asserted, the finished count will increment twice.

The Failed counter is treated similarly.

The default is attempt-based counting. You can select trace-based counting by using one of the following:

Tcl variable		`set assert_count_attempts 0`
Command-line option	`irun` `ncsim`	`-assert_count_traces`

Note: The `assert_count_attempts` Tcl variable has no effect on SystemC PSL assertions. You must use the `-assert_count_traces` option to enable trace-based counting.

Additional Counters for the -strict Option

When you use the `-strict on` option, the following counts are displayed in addition to the Finished and Failed counts:

- Pass--The number of finishes + vacuous passes

- Vacuous--The number of times the left operand of an implication is not met
- Attempts--The number of times attempted. Essentially, the number of attempts is the number of clock cycles.

Setting Assertions finished/failed limits

By default, for assert properties, IES reports all failures and all finishes. For cover properties, IES reports one finish per cover property.

You can control the reporting of the failed and finished assertions using different run-time options. These options are explained below:

- Disabling an assert/assume statement after a specific number of failures

An assert/assume statement can be disabled completely after the number of FAIL count reaches a specific number as specified by you. The state of the assertion will be shown as OFF at the next evaluation cycle of the assertion.

Tcl command	`assertion -failure_limit <Number>`
Command-line option	`-abvfailurelimit <Number>`

Run time option to limit failure counts for "assert" and "assume" properties. The <Number> specified applies to individual assertions.

This option is only applicable for SV Assertions.

- Disabling assert/assume properties after N number of total failures

All assert/assume properties in the design/testbench can be disabled completely after cumulative failure count reaches the a specific number as specified by you. For global failure limit, the assert/assume statements will be turned OFF in the next evaluation cycle after the limit is hit. This can lead to the number of failures reported being greater than the failure limit. The next evaluation cycle in case of global failure limit is the next evaluation cycle of any of the assertions.

Tcl command	`assertion -global_failure_limit <Number>`
Command-line option	`-abvglobalfailurelimit <Number>`

Run time option to limit global failure counts for "assert" and "assume" properties. The <Number> specified applies to the cumulative total of all assertions in the design.

Tcl Command to limit global failure counts for "assert" and "assume" directives in database: `assertion -global_failure_limit <Number>`. The `<Number>` specified applies to the cumulative

total of all assertions in the snapshot.

This option is only applicable for SV Assertions.

- Disabling a cover statement after a specific number of finished counts

By default IES reports one finish count for each cover property instance. You can change this default behavior and specify a different finish count limit using the `-abvfinishlimit` command.

A cover statement will be disabled completely after the number of FINISH count reaches the specific number as specified by you. The state of the assertion will be shown as OFF on the subsequent triggering of the assertion. The subsequent trigger might come at same time step, in which case, the assertion will be shown as OFF at that time step only.

Tcl command	`assertion -finish_limit <Number>`
Command-line option	`-abvfinishlimit <Number>`

Run time option to limit finish counts for "cover" properties.

This option is only applicable for SV Assertions.

- Enabling reporting of all finish counts for cover properties

By default IES reports one finish for each cover property instance. You can enable the reporting of all finish evaluations for each cover property using a compile time command-line option.

Tcl command	`assertion -record coverall`
Command-line option	`-abvrecordcoverall`

Run time option to enable recording of all finish counts for "cover" properties.

Note: When the "`-abvfinishlimit`" is provided along with "`-abvrecordcoverall`", the finish limit will be honored by the simulator. Thus, all evaluations of "cover" directives beyond the argument provided with "`-abvfinishlimit`" will be disabled.

This option is only applicable for SV Assertions.

Enabling Cover Properties

By default cover properties are turned off during simulation and are not evaluated with assertions. These properties can be enabled at elaboration time or simulation time.
You can enable cover properties at simulation time using the following options:

- `-abvcoveron`

A simulation time reverse switch, `-abvcoveron`, can be used to enable cover properties at simulation time. When you use this switch cover properties get enabled. The finish count reported within case is one for each cover property.

- `-abvrecordcoverall`

  The simulatio time switch, `-abvrecordcoverall`, enables all the cover properties at the simulation time. Using this switch reports all the finishes of the cover properties.

- `-abvfinishlimit`

  The `-abvfinishlimit` switch enables all the cover properties at the simulation time and reports the number of finishes as specified in the argument of the option.

If all the three switches: `-abvcoveron`, `-abvrecordcoverall`, and `-abvfinishlimit` are specified, the `-abvfinishlimit` takes precedence and the number of finishes specified in the argument of the option are reported.

Enabling Cover Properties at Elaboration Time

An elab time switch, `-coverage u`/ `-coverage all`, enables cover properties at elaboration time. When you use this switch, besides its current functionality it enables cover properties during simulation.

Setting Breakpoints on Assertion States

When you set a breakpoint on an assertion, you specify what state changes will cause a break in the simulation and put the simulation into interactive debugging mode.

Note: The default for assertions in interactive mode is to break on failures, but no breaks are generated in command-line mode.

Tcl Commands for Assertion Breakpoints

Note: The `stop` command, the `assert_stop_level` variable, and the `assert_output_stop_level` variable are cumulative. For example, you cannot override the `assert_output_stop_level` variable with the `stop` command. Also, if the `assert_stop_level` is `error`, and the `assert_output_stop_level` is `{finished failed}`, the simulation will stop on any `finished` or `failed` transition that is also on an property for which the `assert_stop_level` is `error`.

Breakpoints on Individual Assertions

```
stop -create -object assertion_locator
stop -create -condition {condition_spec}
```

You can use the `stop` Tcl command with `-object` or `-condition` options that refer to an assertion. Expressions must use the Tcl extensions for the language in which the assertion is embedded.

In the following examples, `START_CLEAR_MEMORY` is an assertion. The `m_task` signal is a data object referenced by this assertion.

Break on any state change of START_CLEAR_MEMORY:

```
ncsim> stop -create -object memtest2.mctl.START_CLEAR_MEMORY
```

Break when START_CLEAR_MEMORY fails:

```
ncsim> stop -create -condition {#memtest2.mctl.START_CLEAR_MEMORY == "failed"}
```

Break when the m_task signal is 1 and START_CLEAR_MEMORY begins:

```
ncsim> stop -create -condition {#m_task == 1 &&
#memtest2.mctl.START_CLEAR_MEMORY == "active"}
```

Note: For the `-condition` *condition_spec* option, you must

- Prefix each assertion and signal name with a pound sign, to distinguish it from an enumeration
- Enclose the condition expression in braces
- Enclose the assertion states in quotes
  For details, see "Tcl Expressions as Arguments" in "Using the Tcl Command-Line Interface," of the *Verilog Simulation User Guide*.

Breakpoints on Multiple Assertions

```
stop -assert [-all| [-depth [levels | to_cells | all]] ]
             [HDL_scope]
```

You can use the `-assert` option to the `stop` command to define a single breakpoint on assertion

failure that is shared by multiple assertions in the design.

The `-all` option affects all assertions in the design, while `-depth` affects only those assertions under a particular instance scope. If no scope is specified, the current scope is used.

The `-assert` option can be used with other `stop` options, such as `-exec` and `-continue`.

The following examples show how to set a breakpoint on multiple assertions:

```
stop -assert -all -exec stopscript.tcl

stop -assert -depth all top.inst1

stop -assert -depth 3
```

Assertion-Severity-Based Breaks on VHDL assert Statements

```
assertion -simstop -severity note | warning | error | failure |
                                    never | global
```

The `-simstop` option specifies the minimum severity level for which VHDL `assert` statements stop the simulation. The effect of the `note`, `warning`, `error`, `failure`, and `never` values are the same as for `assert_stop_level` , except on a local level. This option overrides the `assert_stop_level` value locally.

Note: The local level is the scope in which the simulation stopped as a result of the `simstop` option.

For example, if `assert_stop_level` is set to `error`, when any assertion in the design hits the `error` severity level, the simulation stops. If, in addition to `assert_stop_level`, you specify `assert -simstop -cellname` with a `note` severity, the `note` severity overrides the severity specified with `assert_stop_level` for any assertion in the local scope of the specified cell.

The severity is initially set to `error`.

The `global` option sets the `assert_stop_level` value to the specified severity, and overrides the local value.

Tcl Variables for Global Assertion Breakpoints

Tcl variables are not scope-specific. You can see their settings and descriptions by using the following command:

```
help -variables
```

Additionally, you can get the value of an individual variable by using the `expr` command. For example:

```
ncsim> expr $assert_stop_level
error
```

Assertion-State-Based Global Breaks

`set assert_output_stop_level` *value*

You can use the `assert_output_stop_level` Tcl variable to enable/disable simulation breaks based on assertion state transitions, in both interactive and batch modes.

An interactive mode refers to the case when you issue Tcl commands at the simulator Tcl prompt, one command at a time. In this case, the `assert_output_stop_level` Tcl variable is set to a default value failed, and simulation halts at every assertion failure. This implies that if you issue a Tcl command `run 200 ns` and there is an assertion failure at `75 ns`, then the simulation will break at `75 ns`. You will need to run the simulation again from that point onwards.

A batch mode refers to the mode in which the simulation is run without any user control. In this mode, the default value of the `assert_output_stop_level` Tcl variable is `none`. This implies that if an input Tcl script runs simulation for `200 ns` and there is a failure at `60 ns`, the simulation does not stop at the assertion failure. The assertion failure gets reported at 60 ns, but the simulation halts only at `200 ns`.

Change in the default value in both the modes will enable simulation break on specified states. The arguments to this variable are the states listed in "Understanding Assertion States".

Example usage:

```
ncsim> set assert_output_stop_level {failed inactive}
```

This will result in simulation to break when the assertion state is failed or inactive.

If you do not want to break at any assertion transitions, use the `none` keyword as the argument, or an empty list:

```
ncsim> set assert_output_stop_level {}
```

You can also use the `all` keyword for all states. For example:

```
ncsim> set assert_output_stop_level {all}
```

Assertion-Severity-Based Global Breaks

```
set assert_stop_level warning | error | failure | never
```

Specifies the minimum severity level for which VHDL assertions stop the simulation.

The *value* can be `note`, `warning`, `error`, `failure`, or `never`. This variable is initially set to `error`.

If the severity level specified in the `assert` statement for this assertion is at or above the severity level specified by the `assert_stop_level` variable, the simulation stops. For details about setting the severity level for an assertion, see "severity" in "Writing PSL Assertions," of the *Assertion Writing Guide*. If the simulator is in interactive mode, it returns the Tcl prompt. In batch mode, the `assert_stop_level` variable is ignored--the simulator continues to run, and records all assertion failures in the simulation results.

When this variable is used, properties with a user-specified severity of `note` or `warning` will no longer cause the simulator to stop when they fail, and do not affect the exit status.

Note: This command is cumulative with the other commands to stop the simulation. For example, if `assert_stop_level` is set to `error` or greater, and the `assert_output_stop_level` is set to `{finished failed}`, the simulation will stop on any transition to `finished` or `failed` by a property that has an `error` or `failure` status.

Tcl Scripts for Debugging Assertions

You can create Tcl scripts to add flexibility to your assertion debugging. For example, the following executes a Tcl script when a given instance of an assertion fails:

```
stop -create -condition {#$my_assert_name == "failed"} \
-execute {tcl_procedure} \
-continue
```

where `$my_assert_name` is the instance name of an assertion.

Setting Assertion Breakpoints using SimVision

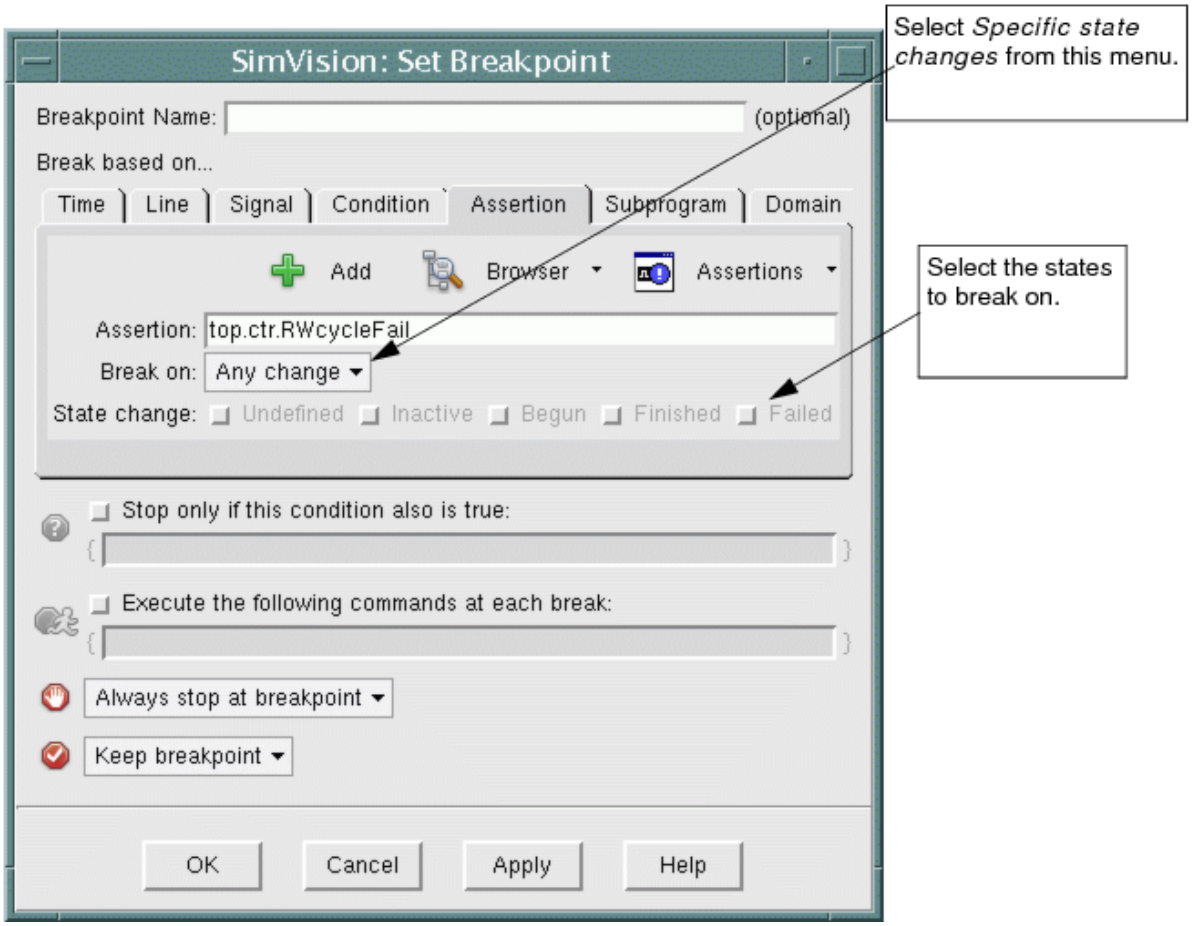
To set a breakpoint on an assertion from SimVision, do the following:

1. In the Assertion Browser or the Source Browser, click on an assertion to select it.
2. From a SimVision window *Simulation* menu, choose *Set Breakpoint - Assertion*. SimVision opens the Set Breakpoint form (Figure 6-5). You can set a breakpoint on any state

change, or on a particular state change.

3. Select *Specific state changes* from the *Break on* pull-down menu, then enable the buttons for the states you want to break on.
4. Click *OK* to set the breakpoint.

Figure 6-5 Set Breakpoint Form



Alternatively, to advance the simulation to a change in the assertion state without placing a breakpoint, you can

- Select the assertion in the Assertion Browser.
- Press the blue Find Next Edge arrow.

7

Running an Assertion Simulation

About Assertion Results during Simulation

After you probe the assertions you want to display in the waveform window, you can start the simulation. As the simulation runs, assertion results are printed in the SimVision I/O area, as shown in this example:

```
ncsim: *E,ASRTST (../verilog/utopia1_atm_rx.with_bug.v,106): (time 157520 NS)
Assertion top.soc.squat_duv.atm_rx_1.ATM_CELL_ENABLE has failed
```

Note: For more information about simulation messages, see "Understanding Simulator Error Messages".

Assertions that are displayed in the Assertion Browser change color to indicate their current state. For a description of these assertion states, see "Understanding Assertion States".

When you simulate assertions, the error message does not always print the line of code that fails. For example, this assertion

```
// psl TEST: assert always read_n;
```

produces the following output during simulation

```
ncsim> run

  ncsim: *E,ASRTST (./memory.v,14): (time 0 FS) Assertion memtest1.mem8x256.TEST has
failed
```

This behavior is intended. The source line is printed only in cases where it will be useful. For combinational properties, there is only one temporal term in the expression. There is no benefit to pointing to it, because the pointer never changes. For sequential properties, on the other hand, there is a real benefit, so the source line is printed for these assertions.

Note: The line that is printed is the line that contains the temporal term that failed. If you want to

print the entire assertion rather than just the term, write the entire assertion on one line.

For example:

```
//     (state == start_clear)

                |

   ncsim: *E,ASRTST (./memctl.v,52): (time 125 NS) Assertion
memtest2.mctl.START_CLEAR_MEMORY has failed (2 cycles, starting 100 NS)
```

Understanding Simulator Error Messages

When you use the PSL `report` option with the `assert` directive, or call an SVA severity system task, the simulator displays a message with the following format:

```
*level, msgID (file, lineNumber): (time) Assertion identifier has failed
(failCycles, startTime)
```

For example:

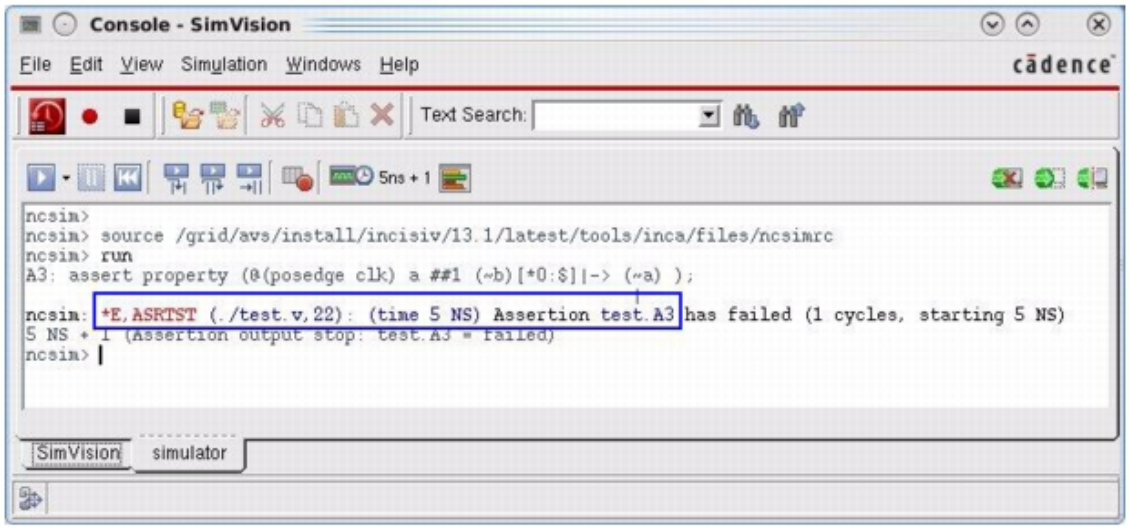
```
ncsim: *E,ASRTST (./memctl.v,86): (time 225 NS) Assertion
memtest2.mctl.CLEAR_MEM_WRITE_N has failed (3 cycles, starting 125 NS)
```

level	One of the following letters, indicating the level of severity: `F`--Fatal error; stops simulation `E`--Error; simulation continues in batch mode, stops in interactive mode `W`--Warning; simulation continues `N`--Note; simulation continues The `assert_stop_level` Tcl variable affects this default behavior. See "Assertion-Severity-Based Global Breaks".
msgID	A six-character string that uniquely identifies the message. You can get additional information about the message by supplying the *msgID* as an argument to the nchelp command. For more information, see "Getting More Information about Errors".
file, lineNumber	The name of the source file that contains the assertion and the line number of the assertion definition in the source file
time	The time at which the assertion failed

identifier	The complete path of the assertion that failed
failCycles, startTime	The number of cycles involved in the assertion failure, and the start time of the assertion trace that failed

When you view the above-given message in GUI, the file, and line number are hyperlinked and color-coded. Using this, you can click on any of these messages and it will take you to the source code of the assertion.

Figure 7-1 Color-coded hyperlinks



If the property definition or assertion directive, or both, are located within protected areas of code, messages might contain less information than described here. For details, see "Writing Assertions for Protected IP" in the *Assertion Writing Guide*.

Getting More Information about Errors

You can use the nchelp utility to print an extended error message. To use nchelp, type in

```
nchelp software_component msgID
```

where *software_component* is the name of the program that reported the error, and *msgID* is the six-letter error code. Both of these values are printed in the error message. For example, for this error message:

```
*E,SIMERR:Error during Simulation (status 1), exiting.
```

you can get more information by using nchelp as follows:

```
> nchelp irun SIMERR
```

```
ncverilog/SIMERR =
     A simulation error has occurred. If the error is coming from
     assertions in the design, simulation may or may not continue,
     depending on how the run is configured. Other types of errors will
     cause ncverilog to not continue to execute.
```

Interrogating Assertions

You can use the `describe`, `scope`, and `value` commands during simulation to get information about assertions.

Note: When you use Tcl commands with wildcards to interrogate assertions, endpoints and sequences will not be matched. For example, `describe` * does not return any `endpoint` or `sequence` statements.

assert_stop_reason

```
assert_stop_reason
```

The `assert_stop_reason` Tcl variable is intended for use in scripts, to make it easy to determine the reason for an assertion stop. You can include this variable in a Tcl script that is input to the simulation. You can use `assert_stop_reason` to identify which assertion caused a breakpoint in simulation. It returns a list containing

- The stop number assigned by the `stop` command
- The hierarchical assertion name
- The directive: `assume`, `assert`, `cover`, or `restrict`

  Note: An `expect` statement will be listed as an `assert`.
- The severity, if specified:
  - SVA--The fail action block severity task: `$error`, `$warning`, or `$info`
  - PSL--The severity value: `error`, `warn`, or `note`
  - When there is no specified severity, the word `default`
- The report value:
  - PSL--The `report` message argument specified for the assertion. For example, the

following assertion definition:

```
assert Clr_Mem_Write_N report "Memory failure"
   severity warning ;
```

will return

```
{stop_1 top.Clr_Mem_Write_N assert warning{Memory failure}}
```

- `{}`--For SVA, or when there is no PSL report message.

There are several ways to use this variable in a script:

- Handling assertion failures
  For example, you can set the assertion stop level:

  ```
  set assert_output_stop_level {finish, failed}
  ```

Then, when the simulation stops as a result of the assertion stop level, you can query the `assert_stop_reason`:

```
echo "$assert_stop_reason"
```

- Setting a breakpoint on specific assertions of interest, and taking some action when they fail
  For example:

  ```
  stop -assert {assertion_names} -exec {echo "$assert_stop_reason"}
  ```

Note: The `stop -assert` feature is not implemented for SystemC.

The following example echos the `$assert_stop_reason` value for all assertions in the design when any assertion break occurs. When the line in the Tcl script that contains the `stop` command is executed, the tool responds with the ID number of that stop. When you run and the stop occurs, you get the normal assertion failure command, then you get the output of the script execution, which in this case is the value of the `$assert_stop_reason` command:

```
  // Associate a script with all assertion failures
ncsim> stop -assert -exec {echo "$assert_stop_reason"}
ncsim> created stop 1
ncsim> run
ncsim> *E,ASRTST, …Assertion test.whatever has failed …
ncsim> {stop_1 test.whatever assert default {}}
```

The second example causes an additional stop point when the `psl_cover` assertion finishes.

```
 // Create a breakpoint when assertion test.psl_cover finishes
ncsim> stop -create -condition {#test.psl_cover == "finished"} -exec {echo
"$assert_stop_reason"}
ncsim> created stop 2
ncsim> run
ncsim> {stop_2 test.psl_cover cover default {}}
```

In both cases, after the run, you see the output of the `$assert_stop_reason`. The first number is the number of the stop point that is assigned by the `stop` command. This is followed by the full name of the assertion, the directive, the default severity, and no report.

Notes about using `assert_stop_reason` for assertion breakpoints:

- Any number of breakpoints can occur in a timestep, but the simulation is not necessarily stopped until the end of that timestep. To see the cause of all breakpoints, you must have a script associated with each breakpoint that captures the value of the `$assert_stop_reason` variable. Using the variable interactively will produce different results.
- The variable is set for each assertion, for each stop point. If there are simultaneous assertion failures, the script that is associated with that stop point will be executed once for each failure.
- If a breakpoint was caused by something other than an assertion, the output of echo `$assert_stop_reason` is `NONE`.
- The `assert_stop_reason` variable is intended primarily to be used in batch mode. If you use it interactively, additional elements will be reported in the stop reason list. In interactive mode, simulation stops that are due to assertions are also controlled by the `assert_output_stop_level` Tcl variable, so an additional stop reason will be reported in interactive mode that is not reported in batch mode. If you do not want to see this additional information, setting `assert_output_stop_level` to `none` or `never`, or specifying an empty list, will ensure consistent values from `assert_stop_reason` for both interactive and batch runs. The following script is an example of how you can use the `assert_stop_reason` variable to control a simulation when it might stop because of assertion failures. This example defines a procedure that executes simulation `run` commands, resuming after every assertion stop, until some other type of event interrupts the simulation. It also shows how to access individual entries and fields in the `$assert_stop_reason` list to control further processing.

```
# Run until the simulator stops with something other than
# an assertion breakpoint.
proc run_with_assertions {} {
  global assert_stop_reason
  while {1} {
    run
    # See if the simulator stopped due to an assertion breakpoint.
    if {$assert_stop_reason == "NONE"} break;
    # Loop over the one or two entries in the assert reason list.
    foreach reason $assert_stop_reason {
   if {[lindex $reason 0] == "assert_output_stop_level"} {
     puts "Processing an interactive assertion stop."
     # Put useful interactive stop processing here.
   } else {
     puts "Processing stop due to breakpoint on [lindex $reason 1]"
     # Put useful breakpoint processing here.
   }
    }
  }
}
```

Note: For more information about `assert_stop_reason`, type `help -variables` at the ncsim command line.

describe

```
describe assertion_name
```

You can use the `describe` command to report the current state of an assertion, including VHDL assertions. This command reports, in this order:

- The assertion name
- The current state of the assertion
- The state transitions to be logged and the name of the assertion log file
- Whether error counting is on or off (see "assertion -logging")

The *assertion_name* can also be an HDL scope, or a signal referenced by the assertion.

The following is an example of using this command:

```
ncsim> describe ATM_CELL_ENABLE

  ATM_CELL_ENABLE property = inactive Logging (failed) to default log. Error: on
```

You can get the same information by using the Assertion Browser, as follows:

1. Select the assertion for which you want information.
2. Press the right mouse button on the assertion and select *Describe*.

The description appears in the SimVision I/O pane.

For VHDL scopes, you can get additional details by using the `-verbose option:`

```
describe -verbose hdl_scope
```

For example:

```
ncsim> describe -verbose :inst1
:inst1.....direct instantiation of entity dut(a)
     VHDL assertion properties
     ------------------------
     Local settings:
         status ................enable
         Logging................default
     Global settings:
         assert_report_level....note
         assert_stop_level......error
```

scope -describe

scope -describe *HDL_scope*

You can use the `scope -describe` Tcl command to list assertions. The assertions are handled in the same way as scope, process, and signal objects are handled. This command returns the name and current state of assertions in the specified scope.

value

value *assertion_name*

The `value` command reports the current state of the assertion. For example:

```
ncsim> value rcv_state
```

```
inactive
```

Enabling and Disabling Assertions

You can use the `assertion` command during simulation to enable and disable assertions.

You can also use the *Disable*/*Enable* menu item on the Assertion Browser pop-up menu to disable and enable assertions. For more information, see "Using the Assertion Browser Pop-Up Menu".

You can use the SVA `$assertoff`, `$asserton`, and `$assertkill` assertion control system tasks in your HDL to enable and disable assertions. These tasks control both PSL and SVA assertions. For details, see "Assertion Control System Tasks" in the Assertion Writing Guide.

Disabling assertions improves performance. Assertions can be disabled at run time, which provides some performance benefit; at elaboration time, which provides more benefit; or at compile time , which provides the most benefit.

Using the simulation command-line option

You can use the `-abvoff` command-line option in ncsim/irun to disable assertions for a particular simulation run.

- *irun* cannot process multiple `-abvoff` switches. Only one `-abvoff` switch can be specified at a time. If more than one `-abvoff` switch is specified, then only the first switch is honored.
- `-abvoff` switch only works for Verilog and SystemVerilog assertions.

This option takes the list of assertions as input using an input argument file. The name of the assertions as specified in the file is the full hierarchical name of the assertion.

The syntax of the assertion file is as follows:

```
assertion -off | -on
[-directive directive_type]
[assertion_hierarchical name]
```

- *assert_hierarchical_name* is the full hierarchical name of an assertion or a wildcard specification (`*` and `?` wildcards are supported). For example:

```
module top;
    ...
    dut inst(...);
   endmodule


   module dut(...);
    ...
    always @(posedge clk)
    begin: blk
    ...
    asrt: assert (...);
    end
   endmodule
```

The full hierarchical name for assertion `asrt` is `top.inst.blk.asrt` which should be specified in the file with the `-abvoff` command.

Using the assertion Command

You can use wildcards with the `assertion` command to restrict the effect of the command based on the name of the object. To ensure that wildcard matching occurs after the recursive search for all assertions, you must combine this feature with the `-depth` option. For example:

```
assertion -off -depth all cov_*
```

If you want to apply the `assertion` command to VHDL `assert` statements only, or to PSL and SVA `assert` statements only, you can use one of the following options to the `assertion` command:

```
assertion -vhdl
assertion -psl
```

These options affect the `-on`, `-off`, and `-logging` options to the assertion command. For example, the following disables PSL and SVA assertions in the `memtest2.mctl` scope:

```
assertion -off -psl memtest2.mctl.*
```

The default is to enable assertion processing for PSL, SVA, and VHDL.

assertion -on

```
assertion -on assertion_locator
```

where *assertion_locator* can be

```
-all | [-depth [ levels | to_cells | all ]]
[-directive directive | { directive_list } | none | all]
assertions_and_HDL_scopes [-cellname vhdl_cell_name ]
```

You can use the `assertion` command with the `-on` option to turn on checking for a specified assertion, or all assertions within a scope. This option turns on assertion reporting, breakpoints, failures in the probe waveform, and incrementing the failure count for the specified property.

You can use the `-directive` option to restrict the effect of the assertion command, based on the kind of PSL or SVA assertion directive, where `directive` can be `assert`, `assume`, `cover`, or `restrict`, and `directive_list` can be any or all of these.

For example:

```
assertion -on -directive cover

assertion -on -depth all -directive {assert assume} top.inst1

assertion -summary -directive assert
```

If you used the `+allow_unused_properties` elaboration option ("Allowing Unused Property

Checking"), you can use `assertion -on` to turn on checking for a property that is not specified in a directive.

The `-cellname` option applies to VHDL `assert` statements only. It applies the command to the complete hierarchy of the specified cell in the design.

Note: You cannot use this command to re-enable assertions that are globally disabled. An assertion is permanently turned off in the following scenarios:

- When command line option "`-noassert`" is used at elaboration time.
- When cover's finish count reaches the value specified on using simulation command line switch, "`-abvfinishlimit`" or its tcl variant, "`-finish_limit`".
- When the total failure count from all the assertions reaches the value specified on using simulation command line switch, "`-abvglobalfailurelimit`" or its tcl variant, "`-global_failure_limit`".
- When the assertion's failure count reaches the value specified on using simulation command line switch, "`-abvfailurelimit`" or its tcl variant, "`-failure_limit`".
- When "`assertion -off -always`" tcl command or "`-abvoff`" command line option is used.
- By default for cover properties.
  To turn on cover properties, enable it using simulation command line switch "`-abvcoveron`" or the elaboration command line switch, "`-coverage u`" or "`-coverage all`".

assertion -off

```
assertion -off [-always] [-concurrent|-immediate][-onfailure
[fail_limit]] assertion_locator
```

where *assertion_locator* can be

```
   -all | [-depth [ levels | to_cells | all ]]
[-directive directive | { directive_list } | none | all]
assertions_and_HDL_scopes [-cellname vhdl_cell_name ]
```

Note: The `-depth` option applies to any given HDL scopes. If no scopes are given, this option has no effect.

You can use the `assertion` command with the

- `-off` option

  Turns off checking for a specified assertion or all assertions within a scope. This option turns off assertion reporting, breakpoints, failures in the probe waveform, and increments the failure count for the specified property. Messages are not recorded to the log file. Action blocks are not executed. Properties that are turned off report their state as `off` rather than `inactive`.

- `-always` option

  Turns off the selected assertions forever. When this command is executed during simulation, all the selected assertions will be disabled for rest of the simulation time. Such assertions cannot be turned on later during that simulation.

- `-concurrent|-immediate` option

  Starting 13.2 release, the runtime tcl command '`assertion -off/-on`' is extended to include a new switch '`-concurrent`' or '`-immediate`' to specify whether concurrent or immediate assertion should be turned `off/on`.

  While *`assertion -off -concurrent -all`* will turn off all concurrent assertions, *`assertions -off -immediate -all`* will turn off all immediate assertions.By default, if nothing is specified then both concurrent and immediate assertions will be turned off. This will also ensure backward compatibility. If a selection of assertions is specified as a list, then the selected list will be honored and the *`-concurrent/-immediate`* switch will be ignored with a warning.

  If a scope name (or list of scopes) is provided as argument(s), the command would apply to all the 'selected' type of assertions in that scope (or scopes). For example, if assertion *`-off -concurrent mymodule`* is used, it would turn-off all the concurrent assertions in 'mymodule'.

- `-onfailure` *`fail_limit`* option

  Automatically turns off a PSL or SVA assertion after its first failure (does not apply to VHDL `assert` statements). This option applies to each instance of an assertion separately, so the same assertion in different places in the design can still fail more than once. If you specify a *`fail_limit`* , the assertion is turned off after the specified number of failures. The failure limit applies as follows:

  - After reset, the assertion is again allowed to fail *`fail_limit`* times before being turned off, because the assertion failure count is reset to zero.

  - After restart, the number of times an assertion can fail before being turned off depends on when the command was issued and how many times the assertion failed before the save, in addition to the user-supplied limit.

- `-directive` option

  Restricts the effect of the `assertion` command, based on the kind of assertion directive, where *`directive`* can be `assert`, `assume`, `cover`, or `restrict`, and *`directive_list`* can be any or all of these. For example:

```
assertion -off -directive cover
```

```
assertion -off -depth all -directive {assert assume} top.inst1
```

```
assertion -summary -directive assert
```

- `-cellname` option

  Applies to VHDL `assert` statements only. It applies the command to the complete hierarchy of the specified cell in the design.

The `assertion -off` command is not affected by a reset or restart command; assertions remain turned off after reset/restart. To turn the assertions on again, use `assertion -on`.

assertion -strict

```
assertion -strict on | off
```

The Incisive simulator runs the pass statement of a SystemVerilog action block only when the status is finished, not when a vacuous pass occurs. You can override this behavior by using the `assertion` command with the `-strict` option to control simulation checking of SystemVerilog assertions. The default is off.

You must use this command *before* starting the simulation.

The following example shows the default assertion report for a simulation run that was generated with the `-strict` option set to `off`:

```
ncsim> assertion -summary
  Disabled Finish Failed    Assertion Name
         0      1      0    memtest2.mctl.CLEAR_MEM_WRITE_N
         0      1      0    memtest2.mctl.COUNT_BURST_READ_N
         0     13      0    memtest2.mctl.M_REQ_FOLLOWED_BY_DONE
         0      1      0    memtest2.mctl.READ_BURST_READ_N
         0      1      0    memtest2.mctl.READ_MEM_READ_N
         0     13      0    memtest2.mctl.RETURN_TO_READY
         0      1      0    memtest2.mctl.START_CLEAR_MEMORY
         0      1      0    memtest2.mctl.START_READ_BURST
         0      1      0    memtest2.mctl.START_READ_MEMORY
         0     10      0    memtest2.mctl.START_WRITE_MEMORY
         0     10      0    memtest2.mctl.WRITE_MEM_WRITE_N
         0    266      0    memtest2.mctl.mem8x256.CHECK_X
         0    266      0    memtest2.mctl.mem8x256.CHECK_Z
         0     11      0    memtest2.mctl.mem8x256.READ_N_AND_ENABLE
         0    554      1    memtest2.mctl.mem8x256.READ_N_AND_WRITE_N
         0    266      0    memtest2.mctl.mem8x256.WRITE_N_AND_ENABLE_N
  Total Assertions = 16,  Failing Assertions = 1,  Unchecked Assertions = 0
  Assertion summary at time 29125 NS + 0
```

When the `-strict` option is on, the simulation

- Executes the pass action block of concurrent assertions on a vacuous pass
- Counts the number of attempts (*Attempt* column)
- Counts the number of times passed due to vacuity, where the left operand of an implication is not met (*Vacuous* column)
- Counts the number of vacuous and non-vacuous passes
  This information is not reported by default. You must use the assertion -summary command with the `-show` option to report the pass information.
- Counts simultaneous finish and fail conditions

The following example shows the default assertion report for a simulation run that was generated with the `-strict` option set to `on`:

```
ncsim> reset
Loaded snapshot worklib.memtest2:v
ncsim> assertion -strict on
ncsim> run
ncsim> assertion -summary
 Disabled Finish Failed   Vacuous  Attempt    Assertion Name
        0      1      0     580      582   memtest2.mct1.CLEAR_MEM_WRITE_N
        0      1      0     580      582   memtest2.mct1.COUNT_BURST_READ_N
        0     13      0     569      583   memtest2.mct1.M_REQ_FOLLOWED_BY_DONE
        0      1      0     580      582   memtest2.mct1.READ_BURST_READ_N
        0      1      0     580      582   memtest2.mct1.READ_MEM_READ_N
        0     13      0     568      582   memtest2.mct1.RETURN_TO_READY
        0      1      0     580      582   memtest2.mct1.START_CLEAR_MEMORY
        0      1      0     580      582   memtest2.mct1.START_READ_BURST
        0      1      0     580      582   memtest2.mct1.START_READ_MEMORY
        0     10      0     562      582   memtest2.mct1.START_WRITE_MEMORY
        0     10      0     571      582   memtest2.mct1.WRITE_MEM_WRITE_N
        0    266      0       0      266   memtest2.mct1.mem8x256.CHECK_X
        0    266      0       0      266   memtest2.mct1.mem8x256.CHECK_Z
        0     11      0       0       11   memtest2.mct1.mem8x256.READ_N_AND_ENABLE
        0    554      1       0      555   memtest2.mct1.mem8x256.READ_N_AND_WRITE_N
        0    266      0       0      266   memtest2.mct1.mem8x256.WRITE_N_AND_ENABLE_N
  Total Assertions = 16,  Failing Assertions = 1,  Unchecked Assertions = 0
  Assertion summary at time 29125 NS + 0
ncsim>
```

Turning Assertions On/Off at a Given Simulation Time

You can use a combination of the Tcl `stop -time` and `-continue -execute` commands to turn assertions on or off at a specified simulation time. For example, the following command stops the simulation at 800 ns, then continues to execute with all assertions turned off:

```
  ncsim> stop -time 800ns -absolute -continue -execute {assert -off
-depth all top}
```

You can use two `stop` commands, one to turn assertions on and one to turn them off, to define a time range within which assertions will be checked.

Controlling Assertion Reporting

For more information about the `assertion` command, see "Using the assertion Command".

assertion -counter

```
  assertion -counter counter assertion_name
```

Prints the value of the given statistics counter, as of the current simulation time, for the specified assertion.

Valid counters are `finished`, `failed`, `checked`, and `disabled`. If you are using the `strict` option, you can also display the `pass`, `vacuous`, and `attempts` counters.

For example:

```
ncsim> assertion -counter failed memtest2.mctl.CLEAR_MEM_WRITE_N
ncsim> 0
```

assertion -resetcounter

`assertion -resetcounter`

The `-resetcounter` option applies to PSL/SVA assertions only.

Use the assertion command with the `-resetcounter` option to reset the assertion coverage counters to `0`.

This option can be used in IXCOM mode only. In non IXCOM mode this option will be ignored and appropriate warning/error message will be issued indicating the same.

Common filter for assertion command like `-scope`, `-assertion_locator`, or `-depth` are not supported with this option.

assertion -list

`assertion -list` `[-permoff]` *[-failed [-new]] [-uncovered] [-depth <numHierLevels>] [-directive <directiveTypes>] <scope>/<assertion_name> [-multiline] [-other_options]*

This option is added to enable examining a subset of design assertions. These are:

- `-list -permoff`

  Returns a list of assertions that are permanently turned off.

- `-list -failed`

  Returns a cumulative list of all the assertions that have failed since time 0.

- `-list -uncovered`

Returns a list of assertions that are still uncovered/unchecked at the time of state access. This will provide a list of assertions which have checked count 0 at the time of state access.

- `-list -failed -uncovered`

  Returns a set of assertions that have either failed since time 0 or are still uncovered at the time of state access.

- `-list -failed -new [-uncovered] <scope/assertion_name>`

  The failed command returns a list of all the assertion present in the given scope which have failed since time 0. If the `-new` option is given with `-failed`, it returns the list of assertions which have failed since the most recent invocation of "`assertion -list -failed`" command..

  If both `-failed` and `-uncovered` are present with new then the result would be a union of assertions meeting those conditions.

  If `-new` switch is used without failed, then a warning message will be given and the command will not be executed..

- `-list -depth <numHierLevels> <scope>`

  `-depth` is the hierarchical level specifier. The `<numHierLevels>` specifies the number of levels to be traversed in the process of locating assertions. `numHierLevels` can take the following values.

  - 0 or all : Indicates the entire depth
  - 1 : Indicates all the assertions in the specified scope(default)
  - 2 : Indicates the specified scopes plus one level hierarchy below each scope. Specifying negative values will result in an error message.

If the scope is given, then the default value of the depth would be 1 (local scope) . If the scope is not given, then the command would run on the entire design returning all the assertions in the design.

- `-list -directive <directiveTypes>`

  Filters the selected assertions by including only the assertions of the specified directive type. Directive types can be assert, cover, assume, restrict. These are also supported by IES.

- `-list <scope/assertion_name>`

  Lists down all the assertions in the given scope. If < *`assertion_name`* > is given, then the

output will display the name of the assertion as it is , provided the assertion name is valid.

- `-list -multiline [ other options]`

  Prints each assertion in a separate line.

Note: The output of all the above TCL commands is a list of assertions falling in the criteria explained above. The output will also be available as a Tcl Result just like any other tcl command.The list can be displayed in two formats depending on whether -multiline option is used in the `assertion -list` command or not. These formats are shown below:

```
assertion -list -failed -multiline

assertion1

assertion2

assertion3

.

.

assertionn
```

OR

```
assertion -list -failed

assertion1 assertion2 assertion3 assertion4 ....... assertionn
```

If there is no assertion in the design, then the simulator will display "No assertions found".

If the assertion is protected, then the simulator will skip those assertions and display only those assertions which are not protected.

assertion -logging

```
assertion -logging [log_options] assertion_locator
```

You can use the `assertion` command with the `-logging` option to specify which state transitions are reported, how to redirect the output, and whether assertions contribute to the total error count of a running simulation.

The *`assertion_locator`* can be

```
   -all | [-depth [ levels | to_cells | all]] assertions_and_HDL_scopes |
[-cellname vhdl_cell_name ]
```

The `-cellname` option applies to VHDL `assert` statements only. It applies the command to the complete hierarchy of the specified cell in the design.

The *`log_options`* can be

- `-append`

  Initially opens the log file specified with `-redirect` in append mode. You can accumulate assertion output from multiple simulations by using this option. The log file is not locked, so using `-append` for parallel simulations might result in overwritten output.

- `-depth levels | to_cells | all`

  The recursion depth is the same as for the probe command; see "Tcl Command to Create Assertion Probes".

- `-error on | off`

  By default, when an assertion fails, the global simulation error count is updated. When you specify off with this option, the default severity for assertion failures is changed from Error to Note. In addition, assertion failures stop contributing to the global error count:

  ```
  assertion -logging -all -error off
  ```

Note: You can also use the ncsim/irun command-line switch `-assert_logging_error_off` to change the default severity for logging assertion failures.

- `-redirect` *`filename`*

  Redirects `assertion` output, up to the next assertion command, to a file, *`filename`*. This option disables the default logging to the standard output device and the simulation log file. For example:

  ```
  assertion -logging -all -redirect run26.log
  ```

To reinstate the default logging, use an empty string as the *`filename`*:

```
assertion -logging -redirect ""
```

Ensure that the total number of open files does not exceed your system's limit on descriptors. An error will be reported and the redirect file will not be created or opened.

- `-severity note | warning | error | failure | never | global`

  This option applies to VHDL assert statements only. It sets the minimum severity level for which VHDL assertion report messages will be output. The effect of the `note`, `warning`, `error`, `failure`, and `never` values are the same as for assert_report_level, except on a local level. This option overrides the `assert_report_level` value locally.

  The `global` value sets the assert_report_level value to the specified severity, and overrides the local value.

- `-state` *`assertion_state | {state_list}`* `| none | all`

  Specifies one, or a complete set of, PSL or SVA assertion state transitions to report textually. Assertions are logged when transition to the `inactive`, `active`, `failed`, or `finished` state. For example:

  ```
  assertion -logging -state {finished failed} -all
  ```

To turn off state transition reporting, use `none` or an empty list. For example, to disable logging of all assertions below the current scope, use the following command:

```
assertion -logging -state {} -depth all
```

assertion -style

`assertion -style` `[` **`-multiline`** `|` **`-oneline`** `]` `[` **`-unit`** `|` **`-statement`** `]`

You can use the `assertion` command with the `-style` option to modify the style of your assertion log messages.

The following options are available:

- `-multiline`

  Assertion log messages are split into multiple lines for easier readability on narrow displays.

- `-oneline`

  Assertion log messages are printed on a single line; this is the default.

- `-unit`

  A *unit* is a standalone component that contains assertions. Components of assertion libraries such as IAL and OVL are examples of units. Assertion messages show the name of the simulation source file in which the unit is instantiated, instead of the name of the library file that contains the assertion.

  Note: This feature is superseded by component-level support for IAL and OVL. You do not need to specify this option for recognized IAL and OVL components. The advantage of using component-level support over the `-unit` option is that it does not affect all assertions the way the `-unit` option does. Cadence does not recommend using `-unit`.

- `-statement`

  Assertion messages show the name of the file in which the assertion is defined, with the exception of those configured for automatic component-level support. This is the default.

Note: Each of these options is global in its effects, so it cannot be used in the same Tcl command with instance-specific options, such as `-logging`.

assertion -summary

```
assertion -summary [-byname | -byfailure] [-final] [instance_name]
                   [-redirect file_name] [-extend_immediate]
                   [-show counter | { counter_list } | all | none]
```

You can use the `assertion` command with the `-summary` option to get a plain text listing of the information displayed in the Assertion Browser--the PSL or SVA assertions in your design and their finished, failed, and disabled counts.

For example:

```
ncsim> assertion -summary
  Disabled Finish Failed    Assertion Name
         0      1      0    memtest2.mctl.CLEAR_MEM_WRITE_N
         0      1      0    memtest2.mctl.COUNT_BURST_READ_N
         0     13      0    memtest2.mctl.M_REQ_FOLLOWED_BY_DONE
         0      1      0    memtest2.mctl.READ_BURST_READ_N
         0      1      0    memtest2.mctl.READ_MEM_READ_N
         0     13      0    memtest2.mctl.RETURN_TO_READY
         0      1      0    memtest2.mctl.START_CLEAR_MEMORY
         0      1      0    memtest2.mctl.START_READ_BURST
         0      1      0    memtest2.mctl.START_READ_MEMORY
         0     10      0    memtest2.mctl.START_WRITE_MEMORY
         0     10      0    memtest2.mctl.WRITE_MEM_WRITE_N
         0    266      0    memtest2.mctl.mem8x256.CHECK_X
         0    266      0    memtest2.mctl.mem8x256.CHECK_Z
         0     11      0    memtest2.mctl.mem8x256.READ_N_AND_ENABLE
         0    554      1    memtest2.mctl.mem8x256.READ_N_AND_WRITE_N
         0    266      0    memtest2.mctl.mem8x256.WRITE_N_AND_ENABLE_N
  Total Assertions = 16,  Failing Assertions = 1,  Unchecked Assertions = 0
  Assertion summary at time 29125 NS + 0
```

This table will appear in the assertion log file and on the console.

Several columns are added to the summary when the `assertion -strict` option is used. See "assertion -strict".

For more information about this summary, see "Analyzing Assertions in the Assertion Browser".

The following options are available:

- `-byname`

  Properties in the text summary are sorted by hierarchical name; this is the default.

- `-byfailure`

  Properties in the text summary are sorted by failure count.

- `-final`

  The summary is deferred until the end of simulation, after all property checks are complete. If multiple `assertion -summary -final` commands are issued, the last one is used.

- `instance_name`

  The summary starts at the specified instance name. Only assertions in that module and below are included in the summary report.

- `-redirect`

  Redirects the assertion summary to a separate file specified by *`file_name`*.

- `-extend_immediate`

  Displays full/hierarchical name for labeled immediate assertions. By default, full names are

not printed in the summary text for labeled immediate assertions, as shown below:

```
ncsim> assertion -summary
  Disabled Finish Failed    Assertion Name
         0    148      0    :mctl.__assert_1
         0      6     81    :mctl.en_set_during_read
         0      0      3    :mctl.mem8x256.__assert_1
         0    148      0    :mctl.state_is_valid
Total Assertions = 4,  Failing Assertions = 2,  Unchecked Assertions = 0
  Assertion summary at time 25 NS + 0
```

With the `-extend_immediate` option, the full names are displayed in the summary text, as shown below:

```
ncsim> assertion -summary -extend_immediate
  Disabled Finish Failed    Assertion Name
         0    148      0    :mctl.__assert_1
         0      6     81    :mctl.en_set_during_read.en_set_during_read
         0      0      3    :mctl.mem8x256.__assert_1
         0    148      0    :mctl.state_is_valid.state_is_valid
Total Assertions = 4,  Failing Assertions = 2,  Unchecked Assertions = 0
  Assertion summary at time 25 NS + 0
```

- `-show`

  Specifies which assertion statistics counters to include in the summary report. Valid statistics counters are `finished`, `failed`, `checked`, and `disabled`. If you use `-strict` mode, you can specify the `pass`, `vacuous`, and `attempts` counters. To specify more than one counter, use the option multiple times, or specify a list of the counter names. You can use all to print all counters. To omit all counters, you can use an empty list or `none`. For example:

  ```
  assertion -summary -show failed
  assertion -summary -show failed -show disabled
  assertion -summary -show {failed disabled}
  ```

To see whether an assertion is enabled or disabled, you must include the *Checked* counter.

assert_report_level

```
set assert_report_level value
```

Sets the global minimum severity level at which VHDL and PSL assertion messages will be reported, for all scopes and assertions. The *value* can be `note`, `warning`, `error`, `failure`, or `never`.

If the severity level specified in the assertion statement is at or above the severity level specified by this variable, the report message is written to standard output, along with the time and the name of the design unit in which the assertion occurred.

This variable is initially set to `note`.

8

Viewing Assertions

Viewing Assertions in the Source Browser

Assertions are cross-selectable in the Source Browser. That is, when you select an assertion in the Source Browser, it is also selected in the other SimVision windows, with the exception of assertion state change, failure event, and transaction traces in the waveform window.

When you double-click on an assertion in the Assertion Browser, the assertion is highlighted in the Source Browser, as shown in Figure 8-1.

When you roll the cursor over a property or assertion name in the Source Browser, the current state, assertion name, and database pop up over the Source Browser window.

When assertion processing is enabled, the Source Browser adds *Assertions* to the *Select* menu. Choosing this menu item selects all assertions in the scope displayed in the Source Browser, regardless of whether the module or file containing the assertions is currently displayed in the window.

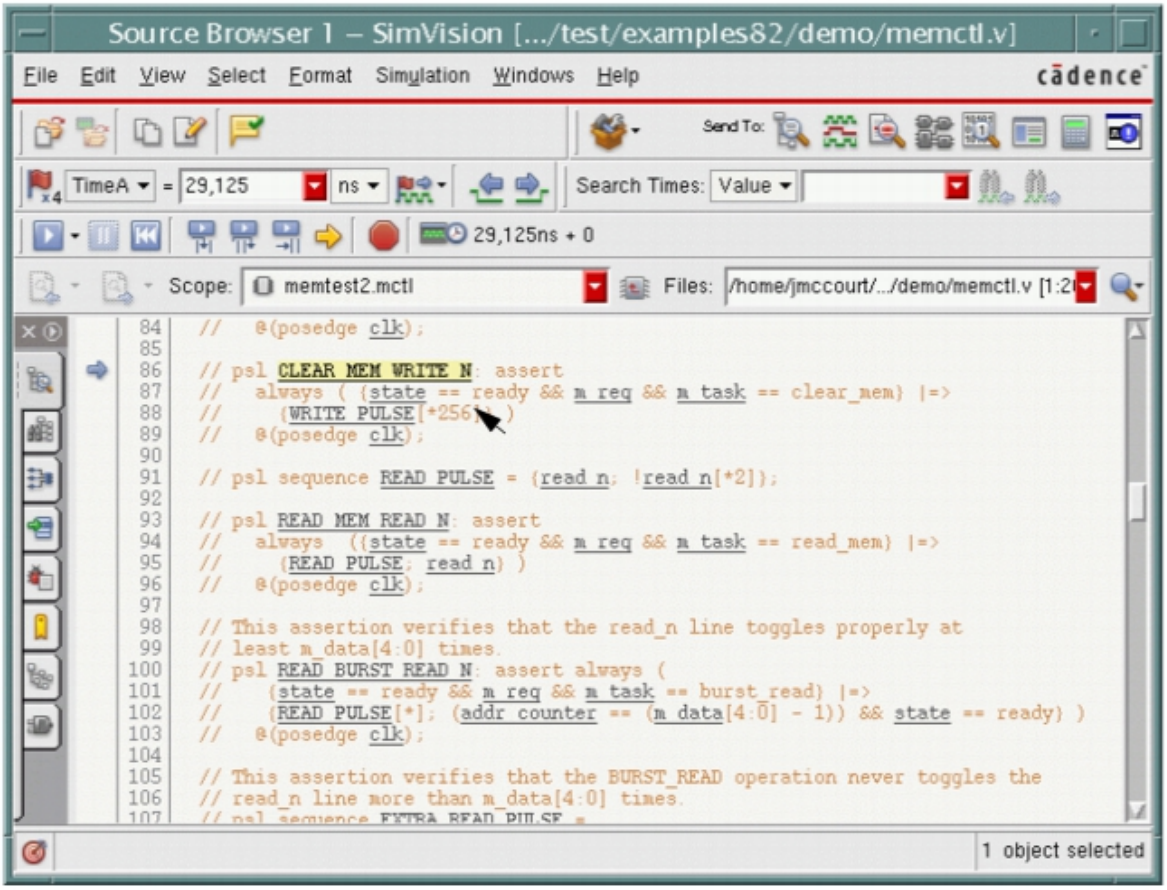
Clicking on a sequence or property instance in the Source Browser displays its definition in the source code. For example, clicking on `READ_PULSE` in this source code

```
  // psl COUNT_BURST_READ_N: assert
// never ( {state == ready && m_req && m_task == burst_read;
// READ_PULSE[*]; EXTRA_READ_PULSE} )
// @(posedge clk);
```

takes you to the `READ_PULSE` sequence definition in the source code:

```
  // psl sequence READ_PULSE = {read_n; !read_n[*2]};
```

Figure 8-1 Source Browser with Selected Assertion



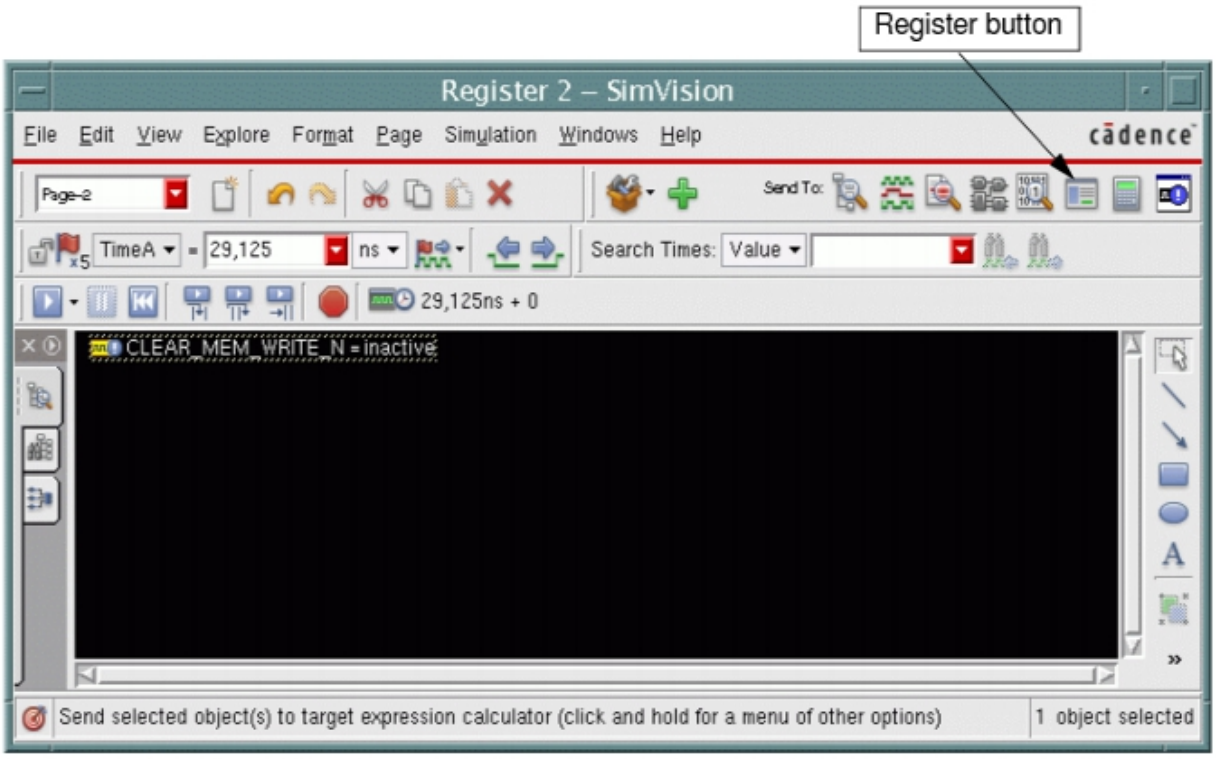
Viewing Assertions in the Register Window

You can place assertions in a Register window and monitor their values as the simulation progresses.

To monitor assertions in the Register window:

1. Select an assertion in any simulation window and click the *Register* button.
   The Register window opens with the assertion displayed in it, as shown in Figure 8-2.
2. Advance the simulation to watch the assertion values change.

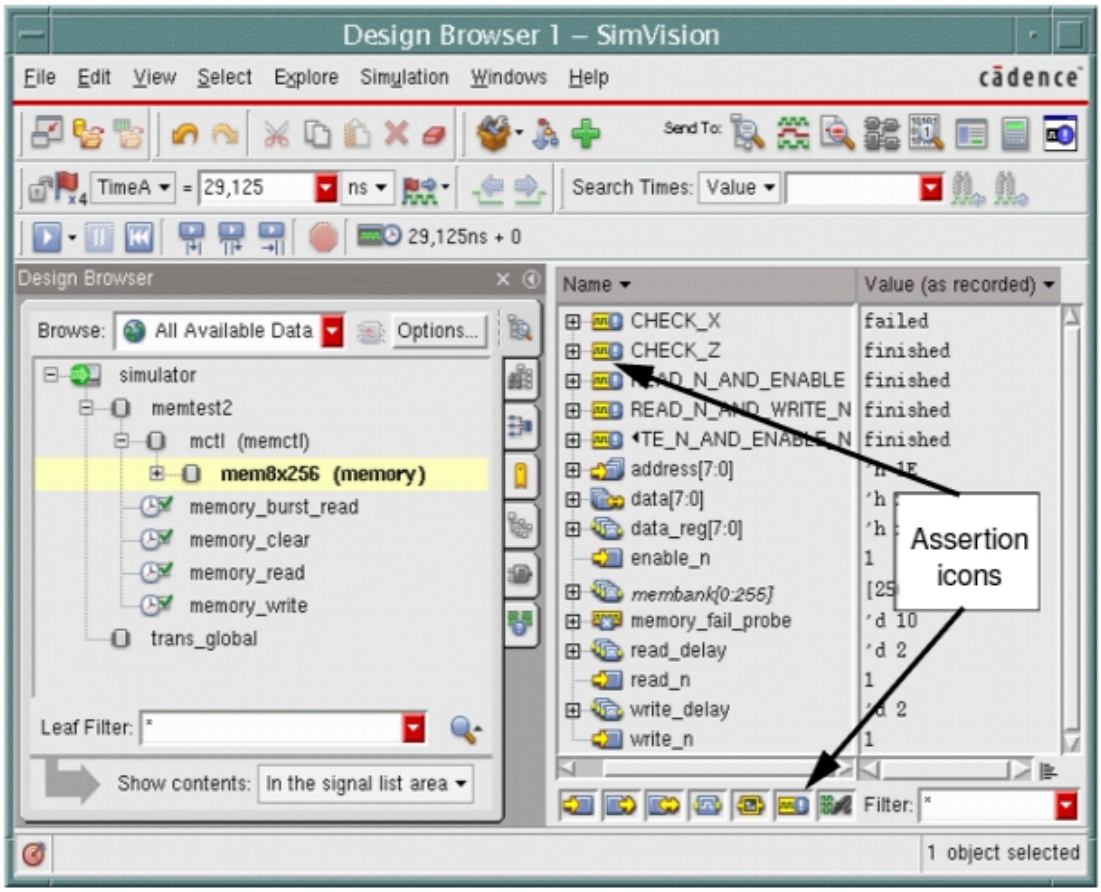
Figure 8-2 Assertion in the Register Window



Viewing Assertions in the Design Browser

Assertions are displayed in the Design Browser in the scope in which they occur, as shown in Figure 8-3.

Figure 8-3 Assertions in the Design Browser



Viewing Assertions in the Trace Signals Sidebar

You can trace a failing assertion back to its source in the Trace Signals sidebar.

To trace an assertion:

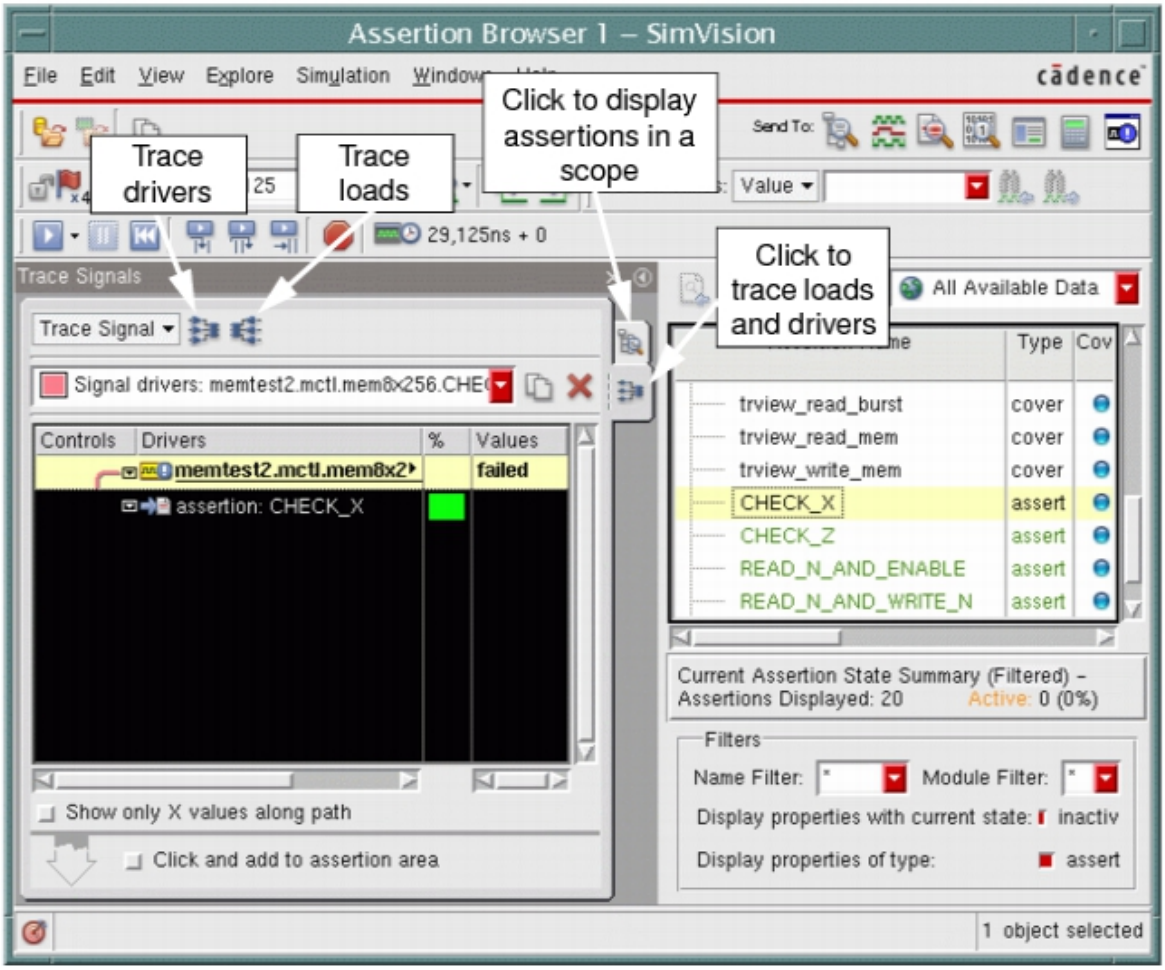
1. Select an assertion in a SimVision window and click the *Trace signal flow through the design hierarchy* tab in the Trace Signals sidebar; the button looks like this:

   This button opens the Trace Signals sidebar

2. Click either the Trace Drivers or Trace Loads button, shown in Figure 8-4.

Figure 8-4 Assertion in Trace Signals Sidebar



3. To display the drivers of any of the referenced signals, select a signal in the Trace Signals sidebar and click the *Trace driving logic* button again.
4. Continue in this way until you find the driver that you want to locate.

Viewing Assertions in the Assertion Browser

The Incisive graphical user interface provides an Assertion Browser to help you monitor assertions during simulation. The Assertion Browser displays a flat list of all of the assertions in a design. The assertions are listed in table format, which you can customize.

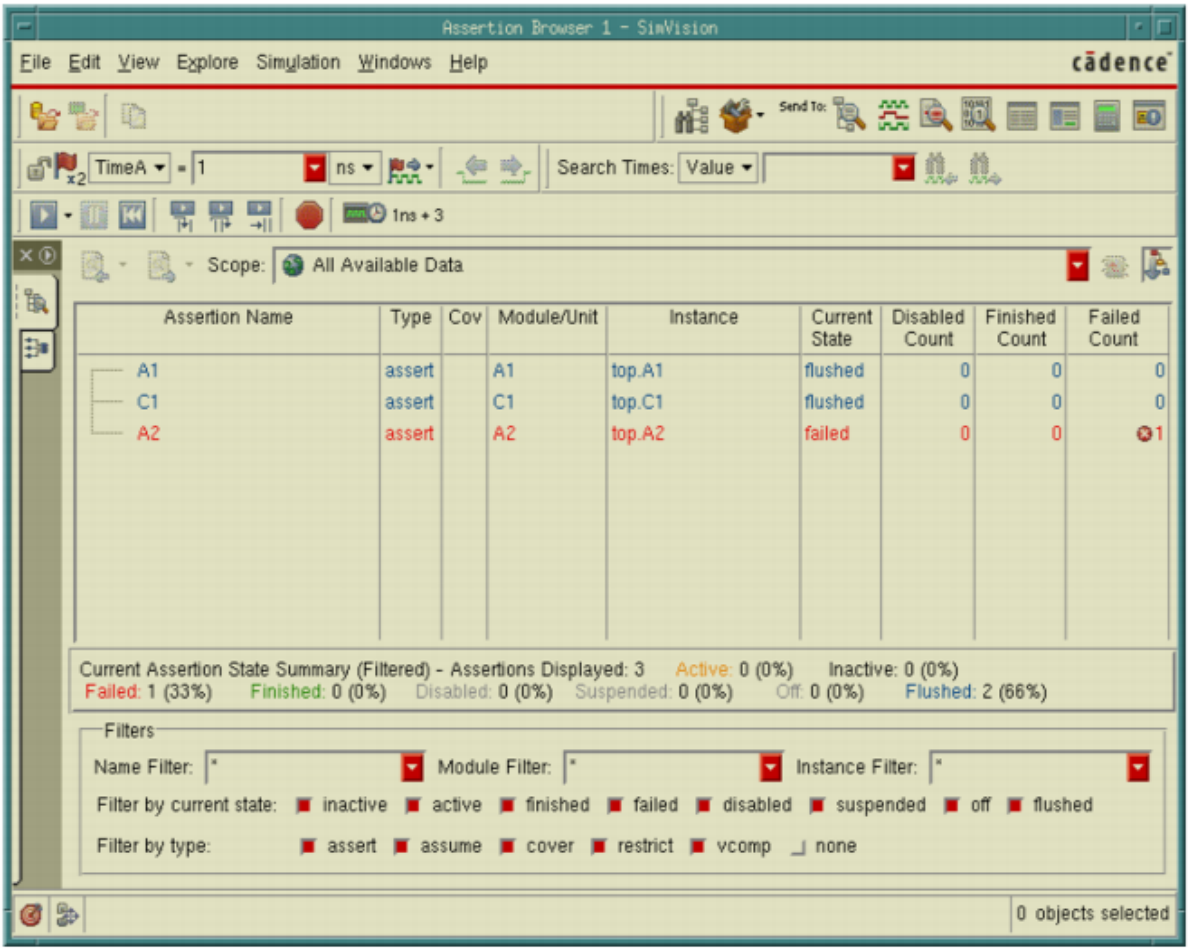
Opening the Assertion Browser

To open the Assertion Browser window, do one of the following:

- Click the *Assertion Browser* button. 
- Choose *Windows - New - Assertion Browser* from the SimVision menu.

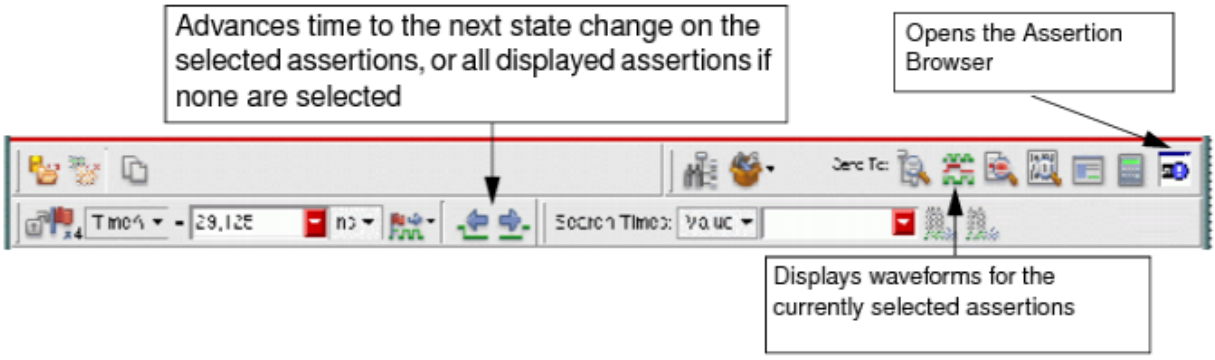
The Assertion Browser window is shown in Figure 8-5.

Figure 8-5 Assertion Browser Window



Several of the standard toolbar buttons do specialized operations when they appear on the Assertion Browser, as shown in Figure 8-6.

Figure 8-6 Assertion Operations for Toolbar Buttons



Information Displayed by the Assertion Browser

The Assertion Browser displays the following lines and columns by default:

- *Current Assertion State Summary*--This line shows summary values associated with the state of the assertions at the current time step. These are not cumulative values.
- *Assertion Name*--By default, this column shows the label that you assigned to the assertion.
- *Type*--The directive type of the assertion: `assert`, `assume`, `cover`, `restrict`, or `vcomp` (OVL or IAL verification library component).
- *Cov*--A filled circle icon indicates that an assertion is selected for functional coverage data collection. This icon will not appear until one of the following commands has been issued:
  - `coverage -func -list`
  - `coverage -func -select`
  - `coverage -func -deselect`
  - `coverage -off`
  - `run`

    You must use the ncsim coverage command to specify coverage data collection; you cannot change the state of this indicator to turn coverage selection on and off. For details, see "Analyzing Functional Coverage". For information about specifying functional coverage, see the ICC User Guide.

    Note: Immediate assertions are never recorded to the coverage database.
- *Module/Unit*--Name of the module or verification library component where the assertion is

defined

- *Instance*--Name of the instance in which the assertion is located
- *Current State*--Current state of the assertion, either `inactive`, `active`, `finished`, `failed`, `disabled`, or `flushed`
- *Disabled Count*--Number of assertion attempts or traces that were cancelled due to a PSL `abort` statement. Number of times the assertion transitioned to `disabled` state for SVA.
- *Finished Count*--Number of assertion attempts or traces that finished successfully; this count is also the coverage count for properties
- *Failed Count*--Number of assertion attempts or traces that failed

When an assertion state is `active` and `finished/failed` at the same time, only the `finished` or `failed` condition is reported in the Assertion Browser; the `active` condition is suppressed.

Note: To see whether an assertion is enabled, you must include the *Checked* column. By default, this column is not included. For details, see "Changing the Assertion Browser Column Layout".

The tab to the left of the columns lets you view assertions by hierarchy, as for the Design Browser, and trace the signal flow of signals referenced by a selected assertion, as for the Trace Signals sidebar ("Viewing Assertions in the Trace Signals Sidebar"). When you expand this area and select

- The design browser tab
  The Assertion Browser displays the assertions in the selected module. If the scope arrow is enabled (in), assertions are displayed for the selected scope and all scopes beneath it. If it is disabled (out), only assertions for the selected scope are displayed.
- The signals tab
  You can select an assertion in the browser, then click the left icon to trace drivers, or the right icon to trace loads.

Sorting Assertions in the Assertion Browser

To sort assertions:

- Click once on a column heading to sort the assertions in ascending order by the value of that

column; click again to sort them in descending order.

An up or down arrow appears in the heading, indicating the sort order.

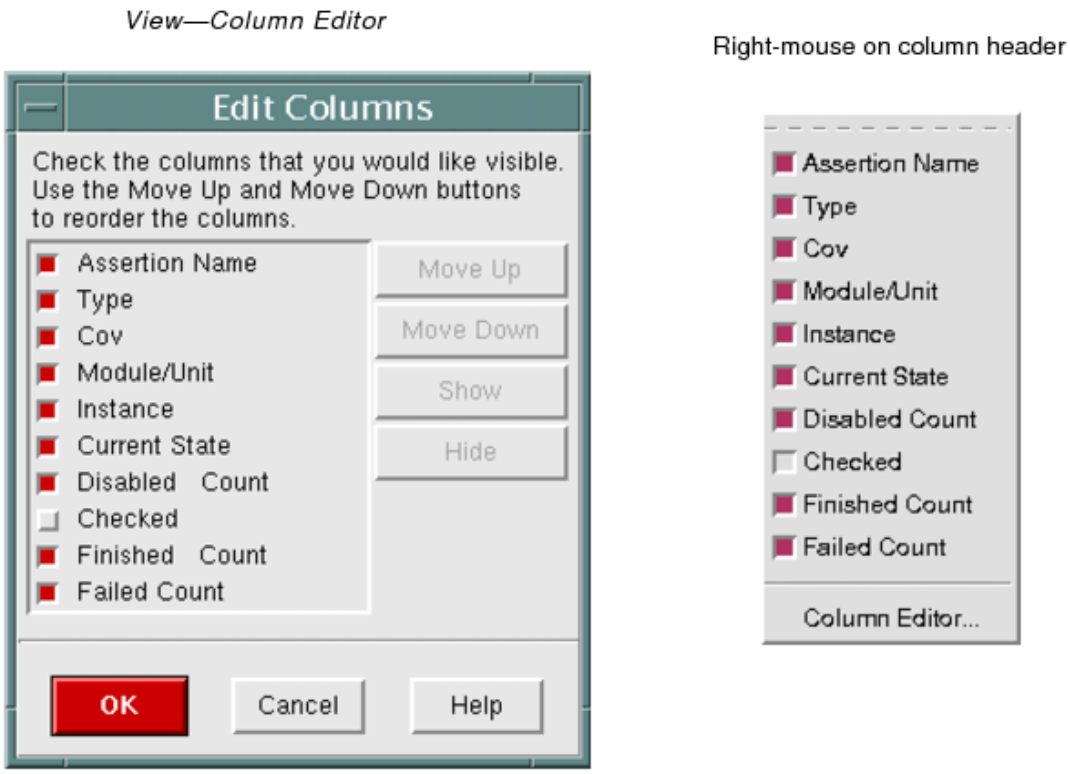
Sorting is additive--that is, a new sort is based on the prior sorting results.

Changing the Assertion Browser Column Layout

To change the layout of the columns:

1. Drag and drop the columns horizontally to rearrange them.
2. Do one of the following to display the Edit Columns form, shown in Figure 8-7:
   - Choose *View-Column Editor* from the menu bar, or
   - Press the right mouse button on a column header, then choose *Column Editor* from the popup menu.

Figure 8-7 Edit Columns Form



- You can edit the columns, as follows:
  - Enable the buttons for the columns you want to display in the window.
  - Disable the buttons for the columns you want to remove from the window.
  - Select a column and click the *Move Up* or *Move Down* button to change the position of the column in the window.
    You can also select the column, then drag and drop it where you want it.
  - Select a column and click the *Show* or *Hide* button to include or omit the column in the window.

You can use the Assertion Browser to detect assertions whose enabling condition was never true. Left-clicking on the *Finished Count* column sorts by the number of assertions that have finished, which brings those assertions that never finished to the top of the column.

Design engineers can write these assertions as they are designing a block, and use the information to see that there is adequate functional coverage at the full-chip level. Coverage assertions are especially useful for making sure that known corner cases get tested. Additionally, you can gather coverage data by writing assertions that test for an expected response.

Filtering Assertions in the Assertion Browser

To choose which assertions are displayed in the window, in the Assertion Browser

- Enter one or more strings in a Filter field:
  - The *Name Filter* field--Displays only those assertions whose assertion names match the strings.
  - The *Module Filter* field--Displays only those assertions whose module names match the strings are displayed in the window.
  - The *Instance Filter* field--Displays only those assertions whose instance names match the strings are displayed in the window.
- Use the *Display properties with current state* buttons to filter assertions based on their current state.
- Use the *Display properties of type* buttons to filter assertions based on their directive type.

The `none` type indicates properties that are not specified in an `assert`, `assume`, `cover`, or `restrict` directive.

To make more room for showing your assertions, you can hide the Filters frame by choosing *View - Filters*.

Note: Because SimVision keeps a history of the strings you enter, you can type a partial string into a *Filter* field and press Tab. If the history contains a matching string, SimVision seeds the field with that string. If the history contains more than one matching string, it displays a list of those strings from which you can choose.

Using the Assertion Browser Pop-Up Menu

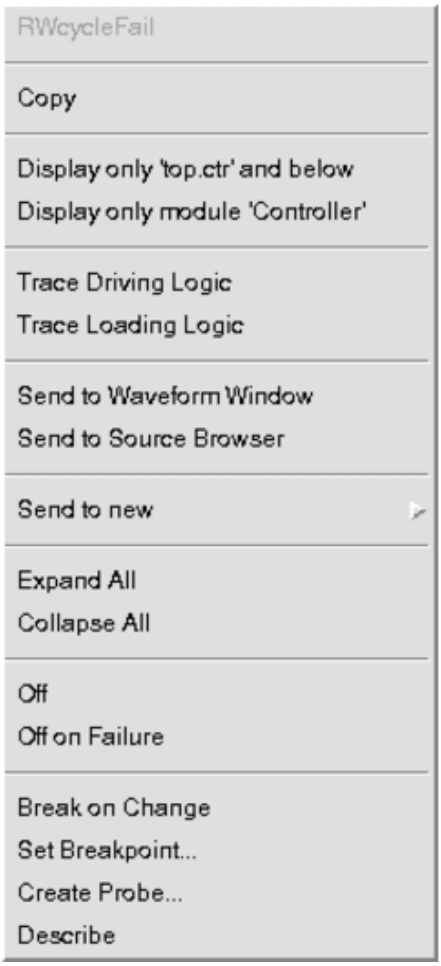
Some frequently-used SimVision commands are available on a pop-up menu (Figure 8-8) when you press the right mouse button on an assertion in the Assertion Browser:

- *Expand All, Collapse All*--Expands or collapses the list of library component instances for all components.
- *Off*--Turns off assertion reporting, breakpoints, failures in the probe waveform, and incrementing the failure count for the specified assertion. Assertions that are turned off are reported as `off` rather than `inactive`. See also "assertion -off".

  This command is a toggle. If the selected assertion is currently turned off, the *Off* and *Off on Failure* entries in the menu are replaced by *On*. See also "assertion -on".
- *Off on Failure*--Automatically turns off an assertion after its first failure. See also "assertion -off".
- *Break on Change*--Creates a breakpoint that will stop simulation when the highlighted assertion transitions to the state you specify. See "Setting Breakpoints on Assertion States".
- *Set Breakpoint*--Displays the SimVision Breakpoint form, which lets you fully customize the breakpoint definition. See "Setting Breakpoints on Assertion States".
- *Create Probe*--Lets you probe an assertion to view in the waveform window, provide a name for use in the waveform window, and further customize assertion probing. See "Setting Up Assertion Probes".
- *Describe*--Prints a text description that includes the assertion name, the current state of the assertion, the state transition to be logged, the name of the assertion log file, and the access

flag for the assertion. See "describe".

Figure 8-8 The Assertion Browser Pop-Up Menu



Viewing Library Components in the Assertion Browser

Components, such as IAL and OVL models, are automatically recognized and treated as verification components. To reduce clutter, the Assertion Browser only shows the rolled-up counts for the verification component. You can expand the component to view the individual assertions and their associated counts (Figure 8-9). The rolled-up count shows the summary of the counts for that instance.

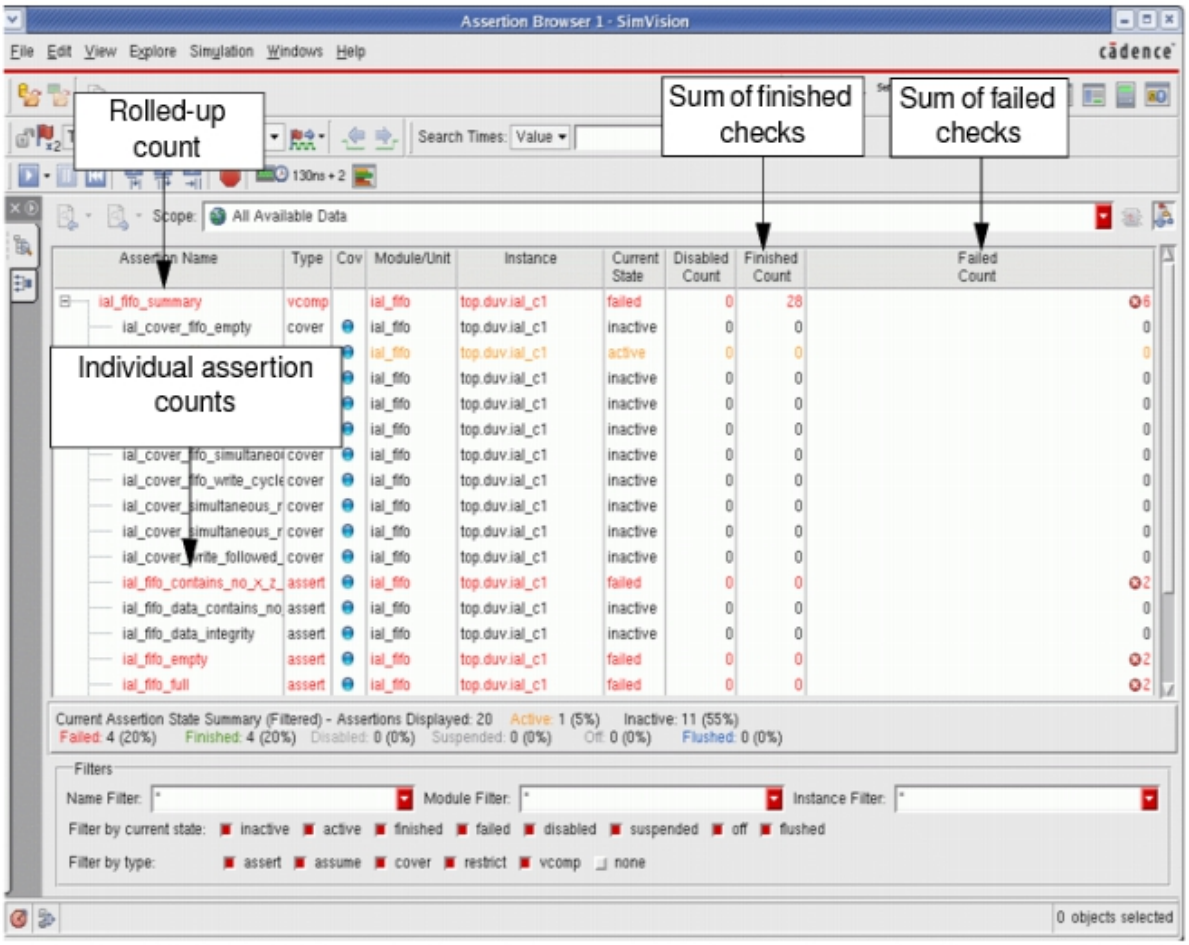
Assertion Browser sorting is based on the rolled-up counts. That is, for each component type, a summary line rolls up the values of the components of that type. Rolled-up counts show the following values:

- Failed--Sum of counts
- Finished--Sum of counts

By default, results for any components whose names start with `ial_` (IAL), `assert_` (OVL), and `ovl_` (OVL 2.x) are reported at component level, and not at the level of individual assertions/coverage points in these verification components. If you want to specify your own naming convention for this feature, you can edit the following file:

install_dir`/tools/inca/files/assertion.conf`

Figure 8-9 IAL Component Instances in the Assertion Browser



To expand or collapse the hierarchy of all displayed component instances at once, you can use the *View - Expand All* and *View - Collapse All* menu items in the Assertion Browser.

Controlling the Assertion Browser using Tcl Commands

You might want to save Assertion Browser commands in a file, so you can re-create the same objects in another SimVision session, or add shortcuts for repetitive tasks. You can use the SimVision command language `assertbrowser` command to control the Assertion Browser.

For example, in the Assertion Browser you can right-click on the `RWends` assertion to select it, then choose *Trace Driving Logic* from the pop-up menu. The SimVision command language equivalent for these actions is the following:

```
assertbrowser sidebar select tracesignals
assertbrowser sidebar access tracesignals trace top.ctr.RWends
```

To list the driving signals for this trace in the console window, use the following:

```
assertbrowser sidebar access tracesignals entry top.ctr.RWends
```

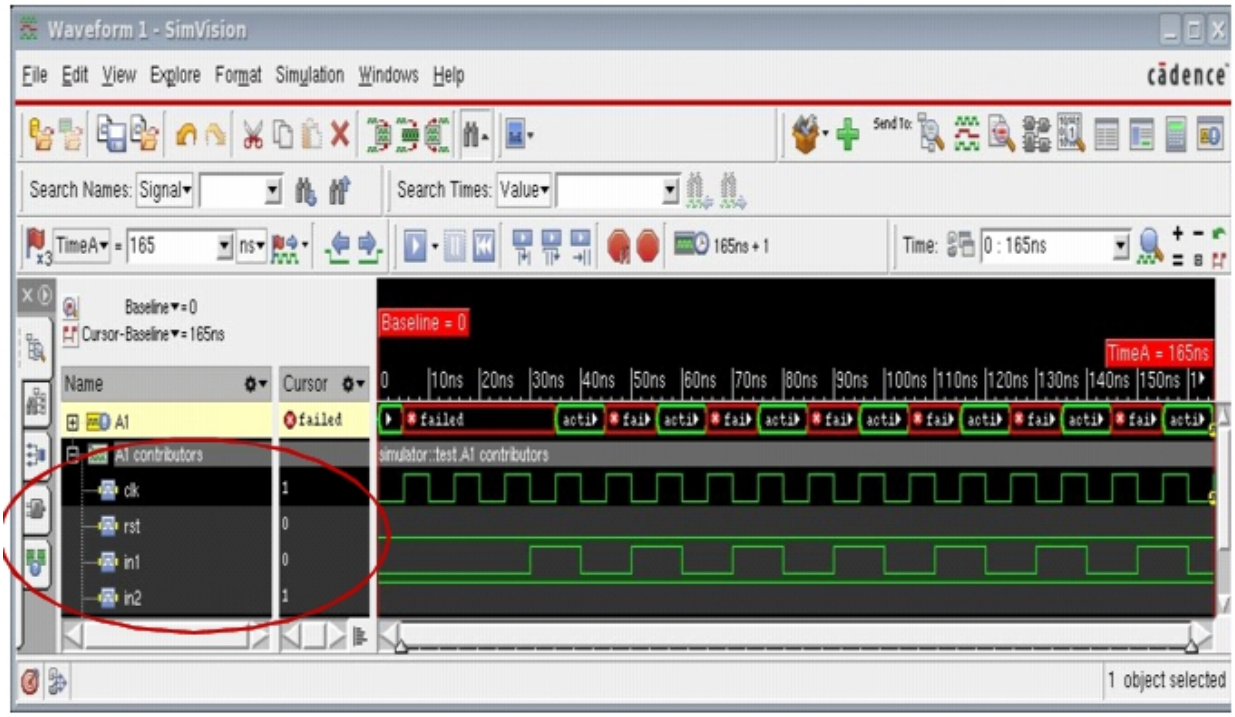
For details, see the *SimVision Command Language Reference* .

Viewing Assertions in the Waveform Window

To view assertions in the SimVision waveform window, you must

- Probe the assertions; see
  - "Tcl Command to Create Assertion Probes"
  - "SimVision Commands to Create Assertion Probes"
- Add the assertions to the waveform display.
  Use the *Add to waveform display* button on the Set Probe form to add assertions to the waveform window, or use the `-waveform` option to the Tcl `probe` command.

  When you add an assertion to waveform display, the contributing signals automatically get added, and become visible. This is shown in the figure below.



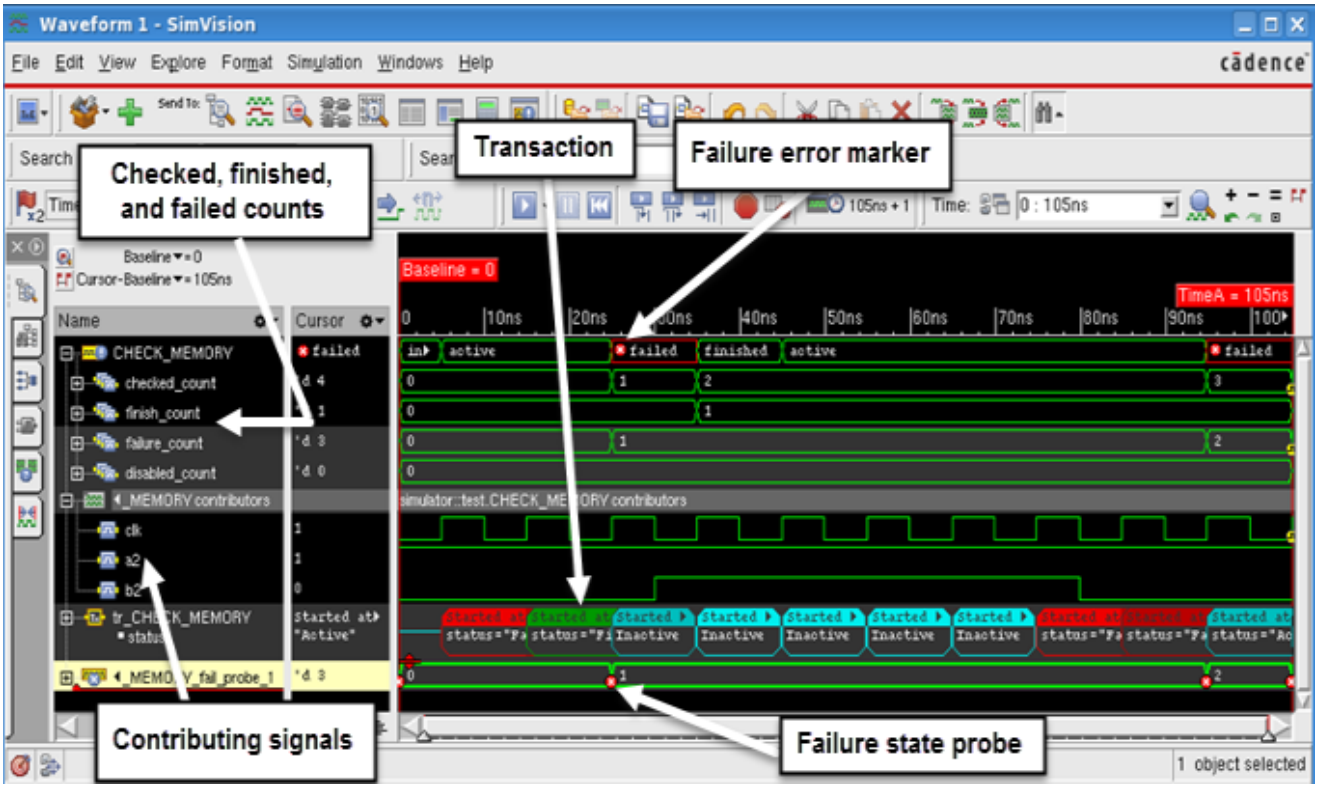
For more debugging information, you can probe the signals that the assertion monitors. See "Probing Signals Referenced by Assertions".

When you select an assertion failure probe transition in the waveform window, you can use *Explore - Go To - Cause* to link back to the assertion definition in the Source Browser.

Figure 8-10 shows how the SimVision waveform window displays assertions recorded as state probes, failure probes, and transactions.

Note: Expanding a count trace shows a binary representation of the count.

Figure 8-10 Assertion State Probe

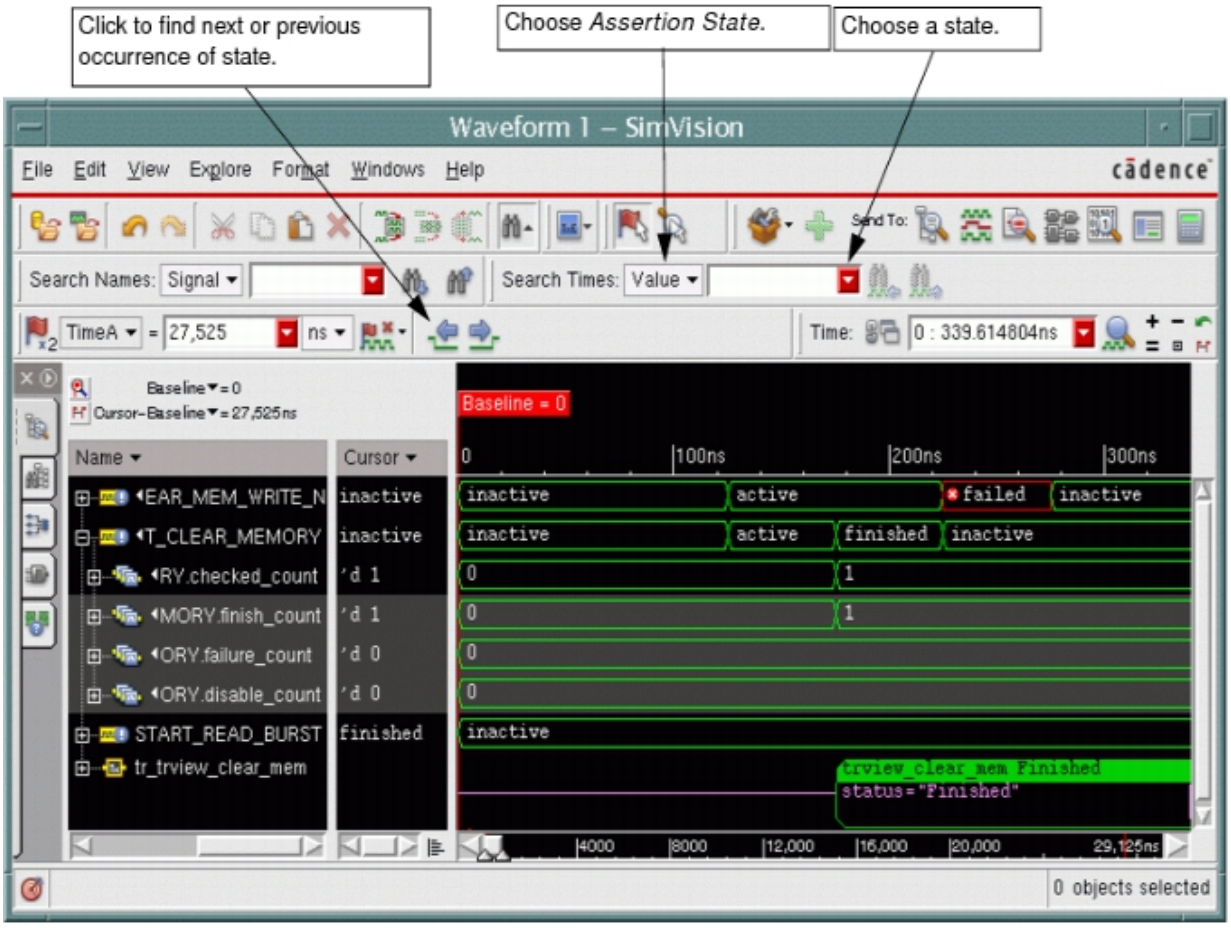


Searching for Assertion States in the Waveform Window

You can use the waveform window to search for a given assertion state, as follows:

1. Select the state probe in the waveform window.
2. In the *Search Times* pull-down, choose *Assertion State* (Figure 8-11).
3. Choose a state to search for in the state pull-down; the default is `failed`.
4. Click the blue *Find Next* or *Find Previous* arrow to move the primary cursor to the selected state.

Figure 8-11 Finding Assertion States



Viewing Overlapping Assertion Transactions in the Waveform Window

For assertions probed as transactions, you can view overlapping transactions in the SimVision waveform window. The information displayed is the same that is reported by using

```
assertion -logging -state {finished failed}
```

Note: For a complete description of transaction viewing in SimVision, see "Viewing Transactions in the Waveform Window " in *Using the Waveform Window*.

Example of Overlapping Assertion Transactions in the Waveform Window

The following example contains an assertion that does the following:

- Finishes twice
- Both finishes and fails
- Fails twice

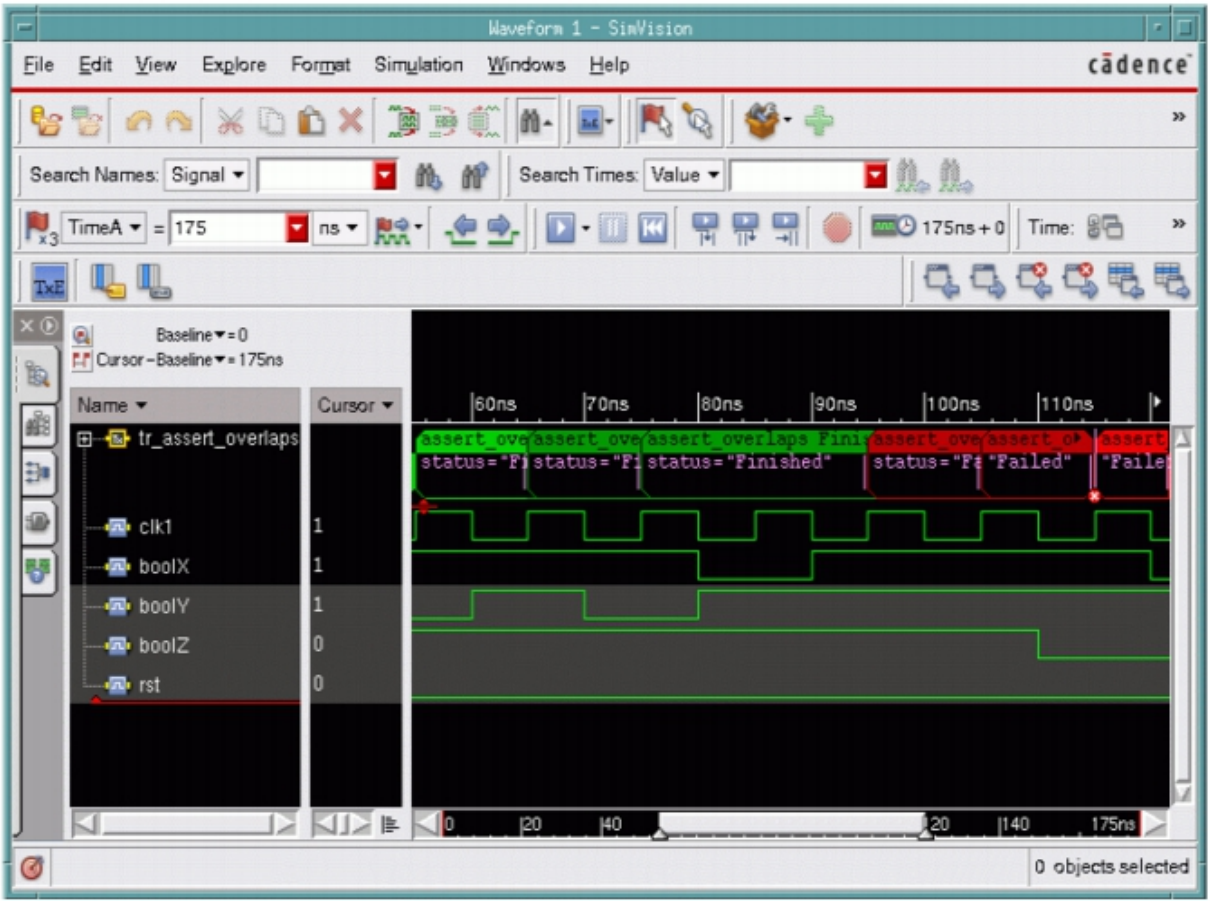
```
property overlaps_demo;
  @(posedge clk1) disable iff (rst)
  boolX ##[1:3] boolY |-> boolZ;
endproperty

assert_overlaps: assert property (overlaps_demo);
```

Figure 8-12 shows how assertions are displayed in the waveform window. The original stream for each assertion shows the overlapping display. To see an expanded view of the full transactions on the stream, expand them as described in "Expanding Overlapping Transaction Streams".

Note: For assertions that finish and fail simultaneously, only the failure is shown.

Figure 8-12 Overlapping Transactions



Expanding Overlapping Transaction Streams

When viewing the transactions, you can use the + to expand the stream. The stream is expanded by attributes that have more than one value, the first of which is the transaction type. However, a more useful method for viewing overlapping transactions on a stream is the following:

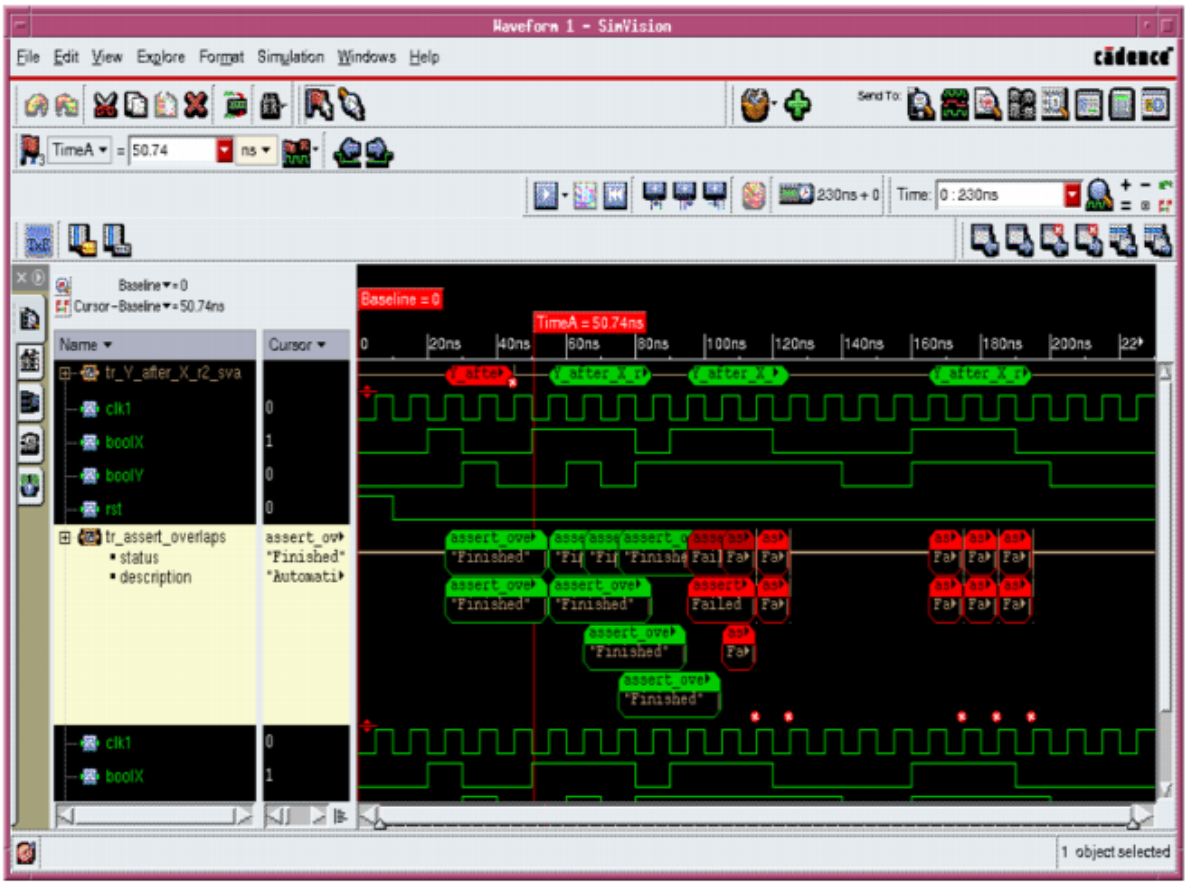
1. Increase the vertical height allocated to the stream by pulling down the red slider so that the transaction attributes are visible.
2. Right-click on the transaction or transaction attribute of interest and select *Expand Stream* or *Expand Stream by <attribute>*.

You can expand the stream by the status attribute, which separates the stream into Finished and Failed substreams. If you use the + to expand a substream, it is expanded by the begin time attribute.

Note: Substreams are created whenever all existing substreams are occupied. A substream is reused if it is available when needed.

To get a menu of all attributes, press the right mouse button over the +/- button to the left of the stream name and choose the *Transaction Attribute* menu. You can then choose the attribute of interest. An example of expanded overlapping transactions is shown in Figure 8-13.

Figure 8-13 Expanding Overlapping Transactions



Note: For trace-based counting, transactions, counters, and log file messages do not report simultaneous finishes, simultaneous fails, or simultaneous finish/fails of different threads.

9

Analyzing Assertions

Analyzing Assertions in the Assertion Browser

Debugging Dashes in the Assertion Browser

If assertions are not probed to the database and the Assertion Browser is not synchronized with the simulator, it displays dashes, "-", for all count values. To synchronize the Assertion Browser with the simulator, you need to put the cursor at the end of the simulation. The easiest way to do this is to enable marching waveforms by clicking the button to the right of the time display:

For more information about using the Assertion Browser in post-processing mode, see Chapter 10, "Post-Processing Assertions.

Analyzing Assertions in the Waveform Window

For information about how to set up assertions for viewing in the waveform window, see "Viewing Assertions in the Waveform Window".

Debugging Interface Assertions in the Waveform Window

The following example demonstrates how to debug an interface by using assertions. This example uses assertions to monitor the inputs on an interface, as a way to isolate problems that can occur when connecting modules together to form a system. The assertions become monitors that constantly check that the signals on an interface meet the required specifications. This example is described in "An ABV Walkthrough ," in the *Assertion Writing Quick Start*.

This example uses a Tcl command to probe all assertions in the memory module to a single failure probe called `memory`, as follows:

```
probe -create -assertions -failure -waveform -name memory memtest2.mct1.mem8x256
```

You can do the same thing using the Assertion Browser, as described in "Probing All Assertions to a Single Failure Probe".

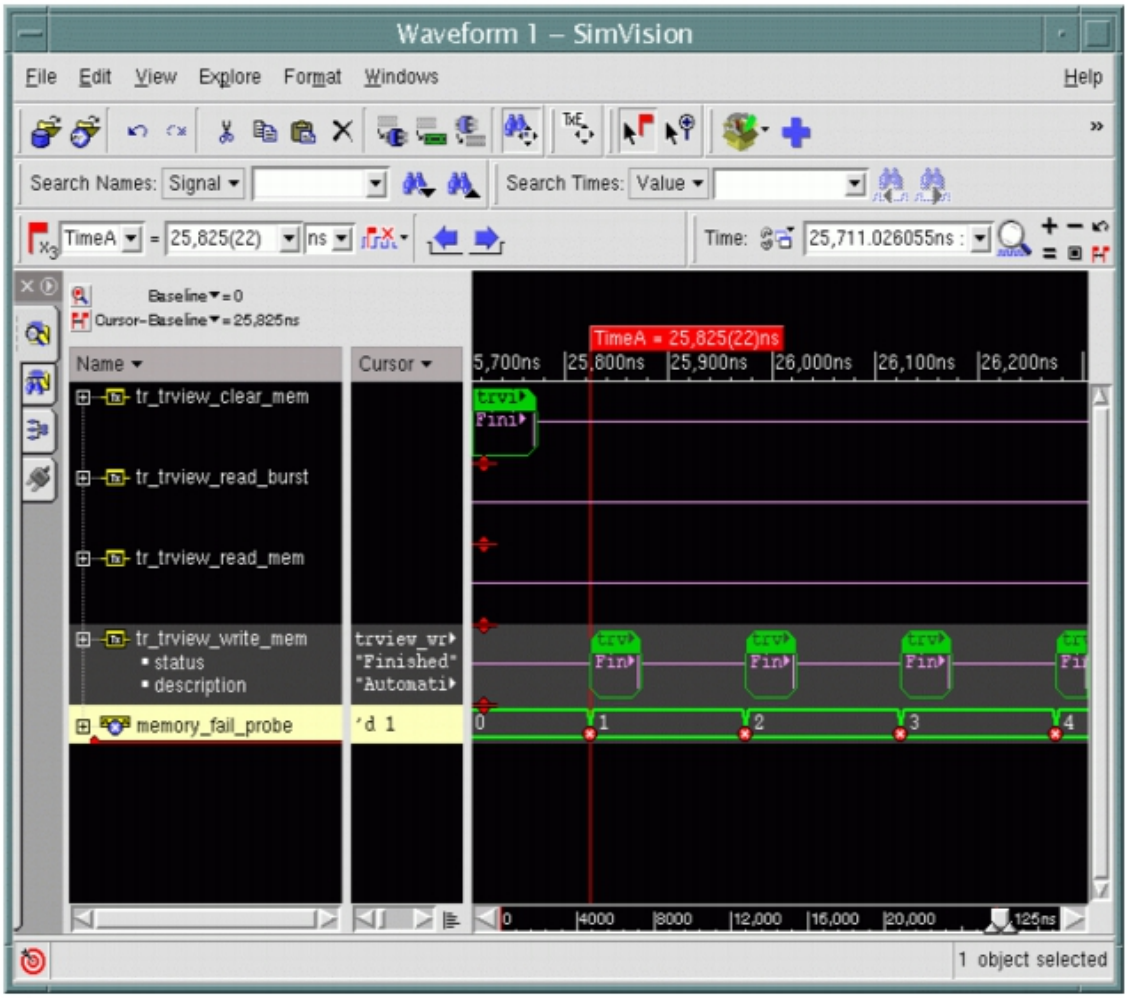
When viewed in the waveform window, this probe is called `memory_fail_probe`. As shown in Figure 9-1, this probe displays an error marker each time an assertion fails, as well as the count of failures.

You can find out which assertion caused a particular failure by

1. Selecting the `memory_fail_probe` trace in the waveform window.
2. Clicking the blue Go To Next Edge arrow.
3. Choosing *Explore - Go To - Cause* from the menu.

SimVision opens the Source Browser with an arrow pointing at the assertion that failed.

Figure 9-1 Debugging Interfaces



"No Drivers Available" Message in the Waveform Window

Sometimes when you put the cursor on a failure probe in the waveform window where there is no "X" and use *Explore - Go To - Cause*, you get a message that says that no drivers are available for `signal_name`:

In this case, try the following:

1. Select the assertion name.
2. Use the blue arrow button to go to the next edge.
3. Choose *Explore - Go To - Cause*.

Viewing Transaction Probes versus State Probes for Assertions

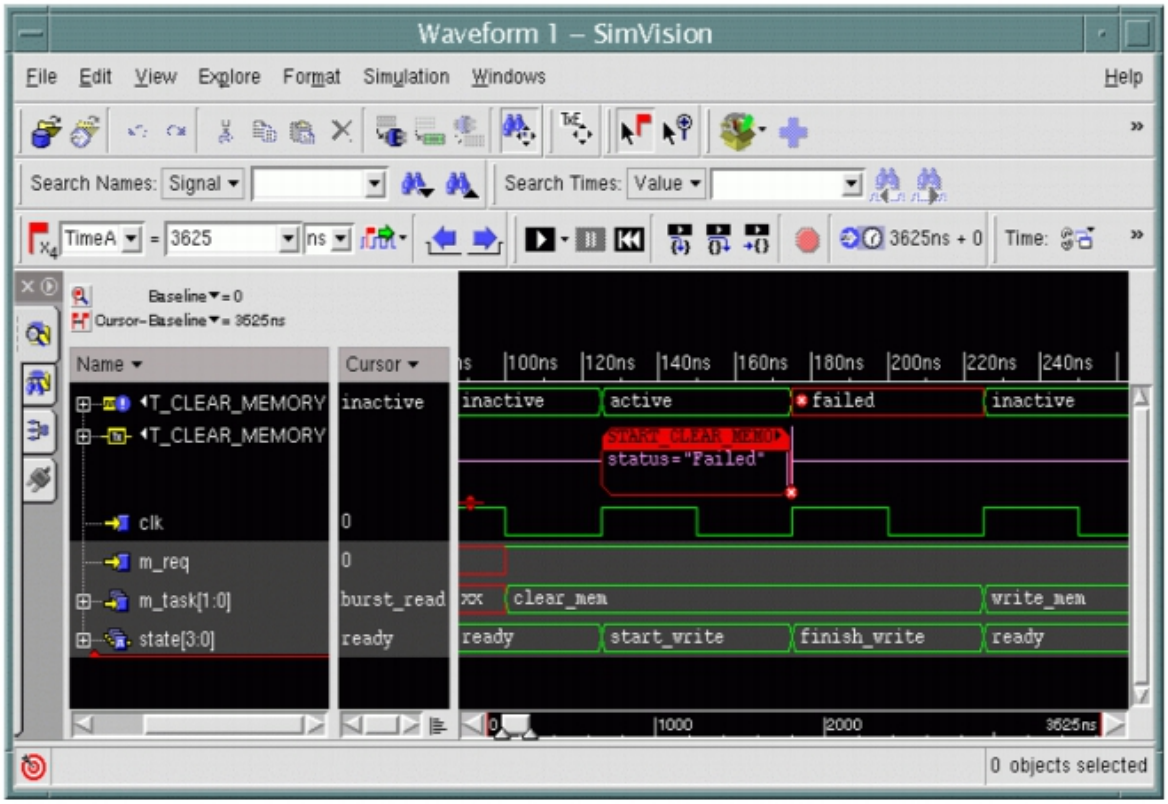
Although the box representing a transaction probe spans a length of time, during which the assertion might be in different states, only the status of the assertion at the end of the transaction is reported. The status reported on the transaction represents the evaluation result. A red X is displayed to highlight failures.

For example, Figure 9-2 shows the same assertion, `START_CLEAR_MEMORY`, displayed as a state probe and as a transaction probe. The assertion starts at time 125 and ends at time 175:

The transaction probe is labeled `Failed`, because the assertion transitions to the failed state at this time.

The state probe shows the same data in a different way: It shows that the assertion value transitions to the failed state at time 125. It maintains this state until the next transition.

Figure 9-2 State Probe versus Transaction Probe



Overlapping Assertion Transactions

An assertion that has repetition in the enabling condition can result in overlapping transactions. For information about viewing these transactions in the waveform window, see "Viewing Overlapping Assertion Transactions in the Waveform Window".

Analyzing Assertion Performance using Simulation profile

IES has two levels of profilers - Basic and Advanced. The Basic profiler is enabled with the -profile option passed to ncsim/irun.

The output of the ncsim/irun -profile option reported in the ncprof.out includes information about the costs of specific assertions. This information supplements the data that is reported for the system library functions, such as `rtl_nc_sequence_eval`, and for `always` statements that are part of the assertion model.

Conditions that are reported include the following:

- A large number of attempts "in flight," which indicates that the enabling condition might be too permissive, or that old attempts are not finished (*W,RUNAWY).
- Frequent false starts, where an inactive property activates, tests its data, and becomes inactive again (*W,FLSTRT).
- Continually increasing memory usage (*W,MEMHOG).

Warnings are reported once per assertion. They appear in the simulation log file, and as a marker next to each assertion in the profile report. The following table shows the numeric legends and conditions corresponding to each of these warnings.

Legend	Warning in the log file	Condition
[1]	*W,RUNAWY	Unusually large number of attempts in progress
[2]	*W,FLSTRT	Frequent false starts
[3]	*W,MEMHOG	Requires increasing memory over time
[4]		Evaluation load accounted for multiple assertions
[5]		Evaluation load accounted in another assertion

Basic Profiling Example using an Artificial Performance Test

An example of a Basic profile report is shown in Example 9-1. The assertion information is shown in the Assertion Summary Counts section of the report.

When analyzing whether there are performance issues with assertions in the design, first check whether the `rtl_nc_sequence_eval` library function is being visited a significant percentage of the time, as shown in the stream counts table in Example 9-1. If it is, check the assertion summary counts to see which assertions are contributing the most.

Example 9-1 Basic Profiling Output

```
------------------------------------------------------------
Stream Counts (48070 hits total)
------------------------------------------------------------
%hits #hits  #inst  name
 72.9 11191 [   ] Library function rtl_nc_sequence_eval
 18.2  2801 [   ] Library function rtl_nc_assertion_update
  1.4   208 [   ] Method SSS_KM_CL2TA (method)
...
  0.1    16 [   ] Method SSS_MT_EMTPES (method)
...

------------------------------------------------------------
Assertion Summary Counts (13880 hits, 90.4% of total)
------------------------------------------------------------
%hits #hits  #inst  name
 29.8  4571 [  2] A1 (assert stmt, file: ./test.sv, line:61)
 16.1  2473 [  1] ABC_NEXT20_TRUE (assert stmt, file: ./test.psl, line: 4)
 13.1  2017 [  1] A_NEXT_B (assert stmt, file: ./test.psl, line: 5)
 10.1  1549 [  1] IM1 (immediate assert stmt, file: ./test.svp, line: 45)
  8.0  1232 [  1] NEVER_A (assert stmt, file: ./test.psl, line: 7)
  6.7  1030 [  1] NEVER_A_THRU_E (assert stmt, file: ./test.psl, line: 6)
  6.3   974 [  1] COV1 (cover stmt, file: ./test.psl, line: 8)
  0.3    34 [  1] EX1 (expect stmt, file: ./test.svp, line: 49) [2]

[2] ncsim: *W,FLSTRT: Frequent false starts
```

The `%hits` column shows the percentage of the total count, not just the assertion-model count. It shows per-assertion counts, not per-instance counts, which is why the names are not hierarchical. The instance count, `#inst`, can help distinguish assertions that are especially slow from those that are just heavily used.

When a specific performance problem is detected, it is reported with a footnote-style notation at the end of the line. For example, the [2] at the end of this line:

```
  0.3    34 [  1] EX1 (expect stmt, file: ./test.svp, line: 49) [2]
```

refers to this footnote in the report:

```
[2] ncsim: *W,FLSTRT: Frequent false starts
```

Assertions that take less than 0.2% of the assertion time are not shown.

Note: Protected models are counted in the profile, but the names and locations are not shown. This output is consistent with the way protected code is handled in other parts of the profile report, such as in the Stream Counts table.

Advanced Profiling Example using an Artificial Performance Test

The Advanced profiler provides additional information on top of the basic profiler, such as the type of Assertion and the detailed load distribution for each of those assertions. It can be enabled using the -iprof command to ncsim/irun.

To view assertion information, use the `report -detail -metrics assertion` command on the iprof shell.

Example 9-2 Advanced Profiling Output for Assertions

```
Assertion Profile Detail Report, Instance Based ==============================================

Instance name: test
File name: <source_file_path>/test.sv

Number of profile hits: 139750 of 187186 (74.66%) Number of Evaluation profile hits: 100146 of
187186 (53.50%) Number of Sampling profile hits: 3260 of 187186 ( 1.74%)
Number of Miscellaneous profile hits: 36344 of 187186 (19.42%)
Grade Assertion Type Line Source Code
-------------------------------------------------------------------------------
27.62%               assert_concurrent1      Concurrent             69         assert_concurrent1:
(51709/187186)                                                                          assert
property(p1)
24.04% (45001/187186) Evaluation Load
               1.84% (3448/187186)           Miscellaneous Load
         1.74% (3260/187186)                 Sampling Load

24.15%             assert_deffered1          Deferred               84         assert_deffered1:
(45213/187186)                                                                          assert
#0 ($stable (d))
               17.35% (32472/187186)         Miscellaneous Load
          6.81% (12741/187186)               Evaluation Load

12.34%             assert_deffered           Deferred                76         assert_deffered:
(23101/187186)                                                                        assert
#0(a == b)
12.34% (23101/187186) Evaluation Load

6.90%             assert_immediate          Immediate               39         assert_immediate:
(12916/187186)                                                                  assert (a == c)
6.90% (12916/187186) Evaluation Load

3.64%             assert_immediate1         Immediate              49        assert_immediate1:
(6811/187186)
assert($sampled(c)==1)begin
3.41% (6387/187186) Evaluation Load
          0.23% (424/187186)                Miscellaneous Load
```

The report shows different loads (sampling, evaluation, miscellaneous) corresponding to each assertion.

Time taken to evaluate assertions can be broadly categorized into:

- Sampling load (Time taken to sample the variable/expression values)
- Evaluation load (Time taken to evaluate the assertion expression)
- Aborting load (Time taken by disable condition specified in the `disable iff`)
- Miscellaneous loads (Time taken by multiclock blocks, action blocks, SVFs etc.)

Example 9-3 A1 : assert property (@(posedge clk) disable iff (reset) $rose(req) ##1 gnt |=> done);

For the above mentioned assertion "A1", time taken to sample the design variables i.e. `req`, `gnt` and `done` add to the sampling load. Whereas time taken in evaluation of property expression and `disable iff` condition contributes to evaluation and aborting load respectively. Time taken in evaluation of `$rose(a)` adds to the miscellaneous load.

Assertion loads (sampling, evaluation, aborting and miscellaneous) depends on the following major factors:

- Number of expressions and variables used in assertions
- Frequency of the change in variable values
- Sample Value Functions usage
- Complexity of the expression
- Use of different sequence operators 'and', 'or', 'throughout', 'intersect'
- Clock frequency and multiclock usage
- Ranges and repetitions
- Disable iff condition
- Action block usage

Note: Time cost of sequence methods used in assertions will still be reported separately as execution of a sequence method may be shared across multiple assertions. This is applicable for both, Basic as well as Advanced profilers.

Identifying Specific Performance Problems

In addition to the basic profiling improvements, which will be useful for diagnosing almost any performance problem, warnings will identify certain problems more specifically. These warnings will provide some indication as to why some assertions might be particularly slow, so that you can improve them, including:

- Large number of attempts in flight, indicating that the enabling condition might be too permissive, or that old attempts never completed
- Large number of false starts, when an inactive property is activated, tests its data, and goes becomes inactive again
- Continually increasing memory usage

If profiling is enabled, these warnings are included in the simulation log file, once per affected assertion.

These warnings also appear as part of the profile report, following the Assertion Summary section, so that they can be analyzed without the need to cross-reference the profile report against the simulation log file.

For more information about performance profiling, see "Using the Profiler to Identify and Eliminate Simulation Bottlenecks," in the "Maximizing Simulation Performance" topic of the *Verilog Simulation User Guide* and the *VHDL Simulation User Guide*.

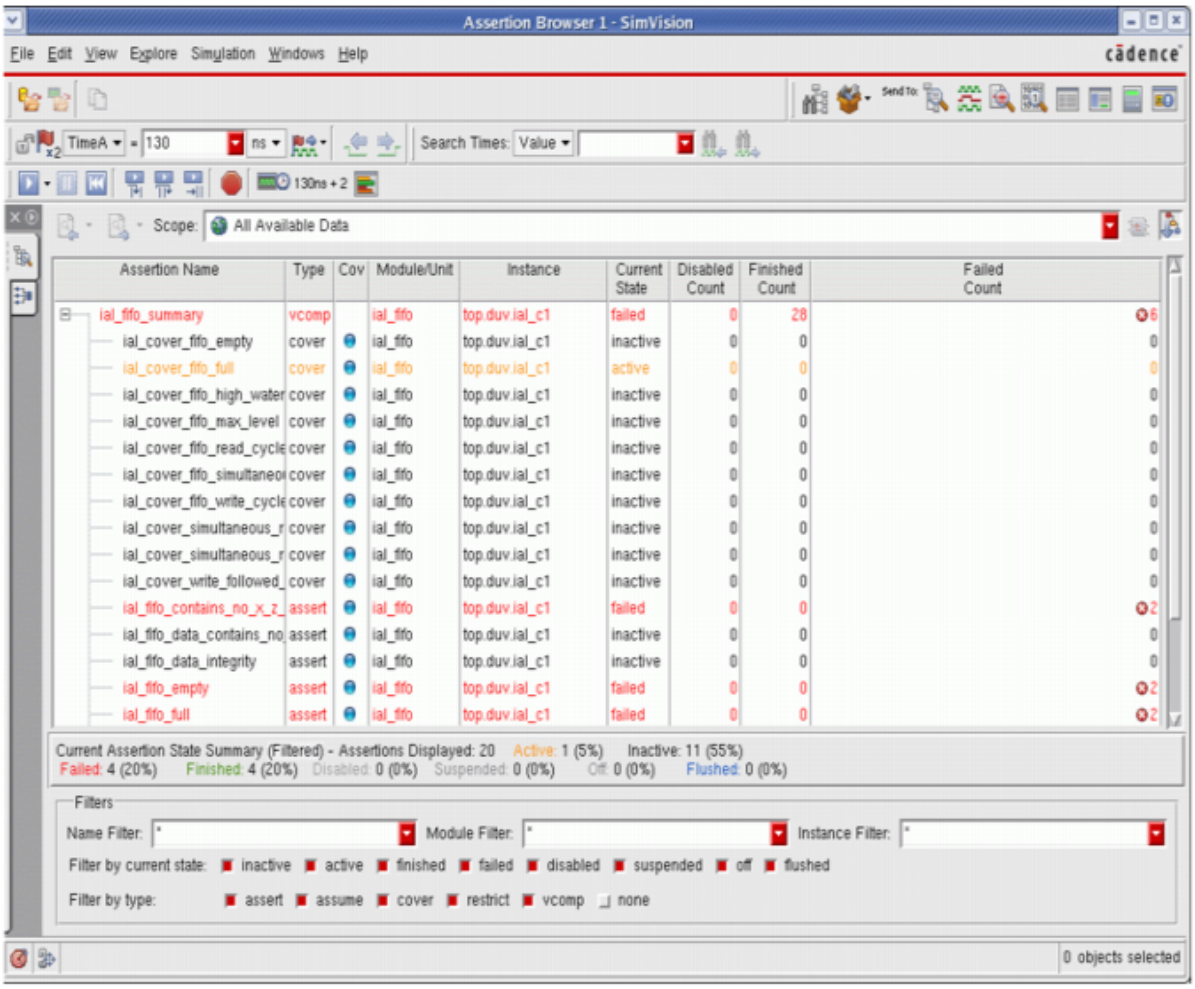
Analyzing Functional Coverage

Using Incisive Comprehensive Coverage for Assertions

You can view coverage information in the Assertion Browser. Assertions that are selected as coverage points are marked in the *Cov* column of the Assertion Browser, as shown in Figure 9-3.

You must configure assertions as coverage points *before* simulation. If you do not, the coverage database will be empty, and ICC and Enterprise Manager will be unable to view coverage data. For more information, see the *ICC User Guide* .

Figure 9-3 Coverage Markers in the Assertion Browser



Using Transaction Explorer for Assertion Functional Coverage

Using assertions, you can create and probe coverage monitors as transactions. These monitors record transactions that represent specific conditions into the simulation database. Assertions that define error preconditions, when probed, also record transactions representing those preconditions. To analyze them, you can

- Post-process recorded transactions after simulation with the Transaction Explorer (TxE) facility
- Analyze them during simulation by using the Run-time TxE facility with your SystemC testbench

You can use TxE to compile coverage information, such as counting the number of transactions that occurred while a design was in a particular internal state.

Ethernet Switch Example

The following example has assertions embedded in the design module of an input port FSM, `mii_inport`. This module is part of a four-port MII Ethernet switch. The module contains a state machine to specify the tasks that are needed to receive a frame. There are seven legal paths through the state machine. Most of these paths are taken when a bad frame is received, after which the state machine must branch to an error state, then return to Idle to wait for the next frame.

To verify that the design handles bad frames correctly, each of the paths through the state machine must be checked. In this example, seven assertions were created, each of which represents the correct response to a possible error condition.

Assertions

One of these assertions is shown in Example 9-4. It first creates some sequences that apply meaningful names to various states, which makes the assertions that use them easier to read and understand. It then defines an assertion called `DEST_ADDR_ERROR` to describe the correct way to handle a destination address error. This assertion, when probed, will be recorded in the database. It reports an error whenever the specified sequence of states does not occur after a destination address error.

Note: `[*]` represents zero or more repetitions of a state, and `[+]` represents one or more repetitions of a state.

Example 9-4 DEST_ADDR_ERROR Assertion

```
// psl sequence IDLE_STATE = {fsmstate == `IDLE};
// psl sequence PREAM_STATE = {fsmstate == `PREAM};
// psl sequence DEST_AD_STATE = { fsmstate == `DEST_AD };
// psl sequence HDR_ERR_STATE = {fsmstate == `HDR_ERR};
//
// psl DEST_ADDR_ERROR: cover {
//  IDLE_STATE;
//  PREAM_STATE[+];
//  DEST_AD_STATE[+];
//  HDR_ERR_STATE[+];
//  IDLE_STATE }
// @ ( posedge mii_clk );
```

A transaction probe is used for this assertion, which applies to the Port 0 scope:

```
  probe -create -name Port0_Dest_Addr_Error -shm DEST_ADDR_ERROR -assertions
-transaction
```

You can use TxE to search for the assertion transactions, either in the database after simulation, or during simulation with Run-time TxE. The TxE search in "TxE Search" creates a summary of

assertion transactions. The `DEST_ADDR_ERROR` assertion is included in the list of transactions for which to search.

The design has four instances of this state machine, one for each incoming Ethernet port. This search counts each path through each state machine, using the transactions as an indication that a specific path was taken.

TxE Search

The following is the TxE search code for this example.

Note: This custom search is written in the TxE/Tcl command language. TxE also offers easy-to-use standard forms for common post-processing searches, and a SystemC run-time facility for transaction analysis during simulation.

```
txe_file_version 4

txe_search_create search AssertionTransactionSummary {
source{
}
init {
    #Create a list of possible transactions
    set trans_list [list PREAMBLE_ERROR DEST_ADDR_ERROR SRC_ADDR_ERROR \
        LENGTH_ERROR PAYLOAD_ERROR CRC_ERROR GOOD_FRAME]

    # Initialize an array to store the summary information
    for {set port 0} {$port <= 3} {incr port} {
        foreach trans $trans_list {
           set summary($port,$trans) 0
     }
    }
}
apply {
    # Look through all of the streams in the mii_inport modules
    fiber "top.soc.duv0.core.mii_inport*" {
        # For all transactions
        trans_type * {
            # Extract the port number and assertion name from the stream name
            set port [string index [attribute fiber] 28]
            set assertion [string range [ attribute fiber] 30 end]

            # Increment the appropriate element in the summary array
            incr summary($port,$assertion)
     }
    }
}
exit {
}
table {
    # Create the header for the summary table
    row Port Preamble_Error Dest_Address_Error Source_Address_Error

    # Create each row of the table
    for {set port 0} {$port <= 3} {incr port} {
     set row_list ""
     foreach trans $trans_list {
         lappend row_list $summary($port,$trans)
     }
     row inport$port $rowlist
    }
}
doc {
}
}
```

TxE Results

The table that summarizes the results has four rows, one for each state machine. Each row has seven entries, one for each path through the state machine. When you view the table in TxE, you might find that there were no frames with destination address errors, or that a certain port was never tested. You can then create new tests to increase the verification coverage in these areas.

The results table created by this search is shown in Table 9-1, which shows that the `PREAMBLE_ERROR`, `SRC_ADDR_ERROR`, `LENGTH_ERROR`, and `CRC_ERROR` sequential assertions did not execute--they might need to be covered by directed tests. Also, the `inport3` row shows all zeros--no frames were sent to this port, so additional directed tests might be required here as well.

Table 9-1 Search Results

Port	Preamble_ Error	Dest_ Address_ Error	Source_ Address_ Error	Length_ Error	Payload_ Error	CRC_Error	Good_ Frame
inport0	0	0	0	0	1	0	2
inport1	0	0	2	0	1	0	0
inport2	0	1	0	0	2	0	0
inport3	0	0	0	0	0	0	0

For information about the Transaction Explorer, see the *Transaction Explorer Reference* and *Transaction Explore User Guide* .

10

Post-Processing Assertions

In post-processing debugging mode, you can view assertion information that was saved to a simulation database in the SimVision waveform window. Post-processing mode can be started in one of these ways:

- Running the simulation in batch mode by using scripts and opening the resulting database
- Running interactively to the desired point in the simulation, then choosing *Simulation - Terminate & Post-Process*

Setting Up Assertions for Post-Processing

As for interactive debugging, if you want to view assertions in the SimVision waveform window and Assertion Browser in post-processing mode, you need to do the following *before* running the simulation:

- Probe the assertions you want to examine.
  For details, see information about the `-depth all` option in "Tcl Command to Create Assertion Probes", and "Probing Signals Referenced by Assertions".
- Execute the following Tcl command during the simulation if you want to get assertion statistics:
  ```
  assertion -summary -final
  ```

  Note: This command, like all Incisive simulator Tcl commands, is unavailable in post-processing mode, because only a SimVision license is checked out for post-processing.

You can include Tcl commands in your batch simulation scripts to perform these tasks, as described in "Tcl Setup Script Example for ABV Post-Processing".

The `.key` file contains all of the Tcl commands executed in your run, including the Tcl equivalent of graphic commands. You can use the Assertion Browser to select and probe all assertions in the design as state probes, then add the appropriate lines from the `.key` file to your Tcl run file.

Using Batch Scripts for ABV Post-Processing

The `run.f` and `setup.tcl` scripts used in the *ABV Writing Quick Start* demonstrate how to set up a simulation for post-processing. These scripts are located in the following directory:

Cadence_install_dir`/doc/abvquickstart/examples/demo`

- The `run.f` script (Example 10-1) is used as input to the `irun` command. It contains the list of files to process, and specifies various runtime options. It also invokes the `setup.tcl` script.
- The `setup.tcl` script (Example 10-2) opens a database, probes signals and assertions, and does other useful tasks.

You can also run a SimVision script in post-processing mode; for details, see "Starting an ABV Post-Processing Session".

Following are examples of how the scripts in the `demo` directory were modified for a batch run:

- The `+gui` command for SimVision access was deleted from `run.f`.
- These changes were made to `setup.tcl`:
  - The `-waveform` option to send probes to the waveform window was removed from the `probe` commands.
  - The `assertion -summary -final` command was added to save assertion state count statistics.
  - The `assertion -logging -all -error off` command was added to turn off assertion error reporting (the `SIMERR` error upon exit).
  - The `exit` command was added to end the simulation.

ABV Post-Processing Run Script Example

The run script supplies input to the command that starts the simulator.

Example 10-1 The run.f Script

```
// Files to be processed
memtest2.v
memctl.v
memory.v

// Enable SystemVerilog constructs and SVA assertion processing.
+sv



// Enable access to data.
+access+r

+propfile+memctl_psl.trview


// Invoke a Tcl script to probe signals and set Tcl variables.
+ncinput+setup.tcl
```

For information about the options used in this script, see Chapter 4, "Compiling and Elaborating a Design with Assertions."

The simulation is started by using this command:

```
irun -f run.f
```

Tcl Setup Script Example for ABV Post-Processing

The setup script supplies Tcl commands to the simulator.

Example 10-2 The setup.tcl Script

```
#
# Open a database with the -event option.
# This will allow the recording of every value change
# event to the database, instead of only one event
# per simulation time at which multiple events occur.
# Also, probe all the signals in the design.
#
database -open waves -into demo_waves.shm -default -event
probe -create -shm memtest2 -all -depth all
# Create failure probes for assertions (one per major block).
probe -create -assertions -failure -name mem_controller memtest2.mctl
probe -create -assertions -failure -name memory memtest2.mctl.mem8x256

# Create transaction probes to show activity at the interface of the memory.
probe -create -assertion -transaction memtest2.mctl.trview_*

# Probe the key state and task signals.
probe -create memtest2.mctl.m_task
probe -create memtest2.mctl.state

# Knowing which assertions are failing, create a state probe;
# also probe all related signals.
probe -create -assertion -signals memtest2.mctl.mem8x256.CHECK_X

# Only stop the simulation if the severity level is error or failure.
set assert_stop_level {error}

# Save assertion state count statistics for post-processing.
assertion -summary -final

# Capture coverage data.
coverage -setup -dut memtest2.mctl -testname memtest2
coverage -functional -select memtest2.mctl

# Suppress error messages
assertion -logging -all -error off

# Run the simulation and exit the simulator.
run
exit
```

For more information about the `assert_stop_level` variable, see "Tcl Commands for Assertion Breakpoints". For information about other Tcl commands and variables that you can use in a setup script, see Chapter 7, "Running an Assertion Simulation."

Using Tcl to Add Assertions to a Database Opened with $shm_open()

You can use the `$shm_open()` system task in your HDL to open a database that you can use to record both signals and assertions. To create this database successfully, you need to include the following Tcl commands when starting the simulation:

- `run` *`number_of_cycles`*

  The database is not created until `$shm_open()` is executed. You must first run the simulation until this system task is executed.

- `database -setdefault database_name.shm`

  When you use the `-setdefault` option to the Tcl database command, the database you opened with `$shm_open()` to record signals will also be used as the default database for assertion probes and other operations. For example, if you open a database for signals with the following system task in your Verilog HDL

  ```
  $shm_open("waves.shm");
  ```

  the simulator will create a database called `_waves.shm` (the underscore indicates that the database was specified from the HDL). If you want to add assertion probes to this database, use the `-setdefault` option to make `_waves.shm` the default SHM database, then probe the signals and assertions.

For example:

```
run 1

database -setdefault _waves.shm
```

Failure and transaction probes are not supported for `$shm_probe()`.

Note: You can use the `"A"` option to the `$shm_probe()` command to create state probes for assertions:

```
$shm_probe("A");
```

Opening databases from Tcl avoids recompilation of source code.

Starting an ABV Post-Processing Session

To start post-processing assertions after a batch run, use the following command:

```
ncsim -ppdb shm_database_name snapshot_name
```

where

- *shm_database_name* is the name you specified with the `-into` option of the `database` command (see Example 10-2):
  ```
  database -open waves -into demo_waves.shm -default -event
  ```
- *snapshot_name* is printed in the simulation log file:
  ```
  Writing initial simulation snapshot: worklib.memtest2:v
  ```

For example:

```
ncsim -ppdb demo_waves.shm worklib.memtest2:v
```

If you start post-processing mode using only the simvision `database_name` command, the Assertion Browser icon will not be present when the SimVision design browser is displayed. SimVision has not yet connected to the snapshot, so it does not know if there are assertions in the design. If you need to see a list of the assertions in the design, you can use the Windows - New - Assertion Browser menu item.

If you have a SimVision command file that you want to use in post-processing mode, you can use the following techniques:

- From the simulator command line:
  ```
  ncsim -ppe -simvisargs "-input simvision_file" shm_database_name
  ```
- When starting SimVision:
  ```
  simvision -input simvision_file -snapshot shm_database_name
  ```

Viewing Assertion Simulation Batch Results

The results from a batch run can include both text and graphic simulation data.

Text Output

The simulation log file contains messages about assertion failures that look the same as the console output does during an interactive simulation. Example 10-3 shows a few of these messages.

Example 10-3 Log File Assertion Messages in irun.log

```
ncsim> run
             100: Clear Memory
    // psl sequence WRITE_PULSE = {write_n; !write_n};
                                          |
ncsim: *E,ASRTST (./memctl.v,86): (time 225 NS) Assertion
memtest2.mctl.CLEAR_MEM_WRITE_N has failed (3 cycles, starting 125 NS)
             25725: Memory Cleared
ncsim: *W,ASRTST (./memory.v,68): (time 25825 NS) Assertion
memtest2.mctl.mem8x256.CHECK_X has failed
    address or data have X's
                25875: Write Memory - address:1e data:21
...
```

The results produced by the `assertion -summary -final` command in the `setup.tcl` file might look similar to the log file output in Example 10-4.

Example 10-4 Log File Assertion Summary

```
ncsim> exit
  Disabled Finish Failed    Assertion Name
         0      1      0    memtest2.mctl.CLEAR_MEM_WRITE_N
         0      1      0    memtest2.mctl.COUNT_BURST_READ_N
         0     13      0    memtest2.mctl.M_REQ_FOLLOWED_BY_DONE
         0      1      0    memtest2.mctl.READ_BURST_READ_N
         0      1      0    memtest2.mctl.READ_MEM_READ_N
         0     13      0    memtest2.mctl.RETURN_TO_READY
         0      1      0    memtest2.mctl.START_CLEAR_MEMORY
         0      1      0    memtest2.mctl.START_READ_BURST
         0      1      0    memtest2.mctl.START_READ_MEMORY
         0     10      0    memtest2.mctl.START_WRITE_MEMORY
         0     10      0    memtest2.mctl.WRITE_MEM_WRITE_N
         0    266      0    memtest2.mctl.mem8x256.CHECK_X
         0    266      0    memtest2.mctl.mem8x256.CHECK_Z
         0     11      0    memtest2.mctl.mem8x256.READ_N_AND_ENABLE
         0    554      1    memtest2.mctl.mem8x256.READ_N_AND_WRITE_N
         0    266      0    memtest2.mctl.mem8x256.WRITE_N_AND_ENABLE_N
Total Assertions = 16, Failing Assertions = 1, Unchecked Assertions = 0
Assertion summary at time 29125 NS + 0
```

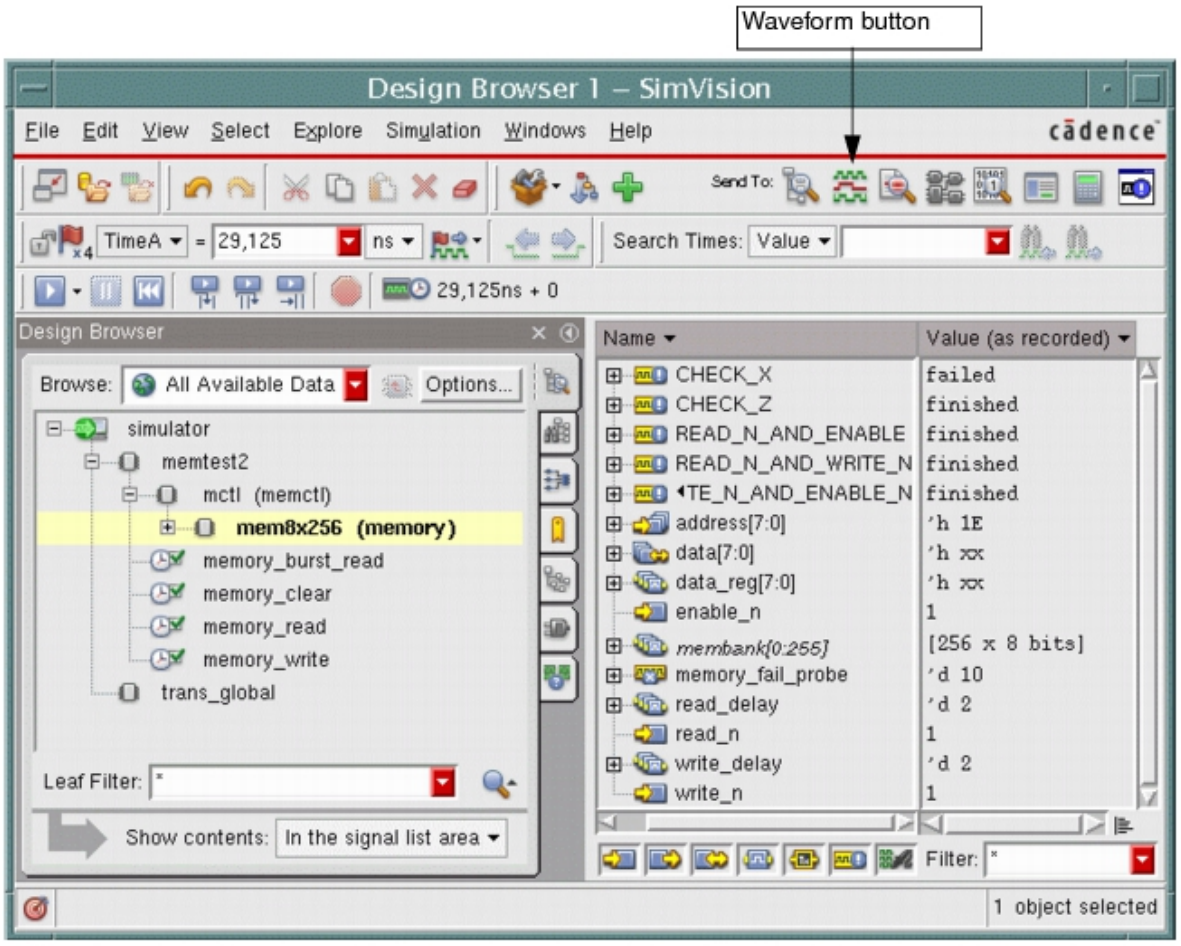
Graphic Output

Once you have loaded your simulation results, you can view them as you would in an interactive session, as described in Chapter 8, "Viewing Assertions." The Assertion Browser operates the same during post-processing as it does during interactive debugging.

To see the count values, all assertions must have been recorded to the database as state probes.

The Design Browser, shown in Figure 10-1, lists the elements in the design. Elements that were not probed are shown in italic; these elements are not available for analysis, because no data was stored in the database for them.

Figure 10-1 Design Browser



Using the Design Browser, you can send signals, assertions, and transactions to the waveform viewer, as you do for an interactive session:

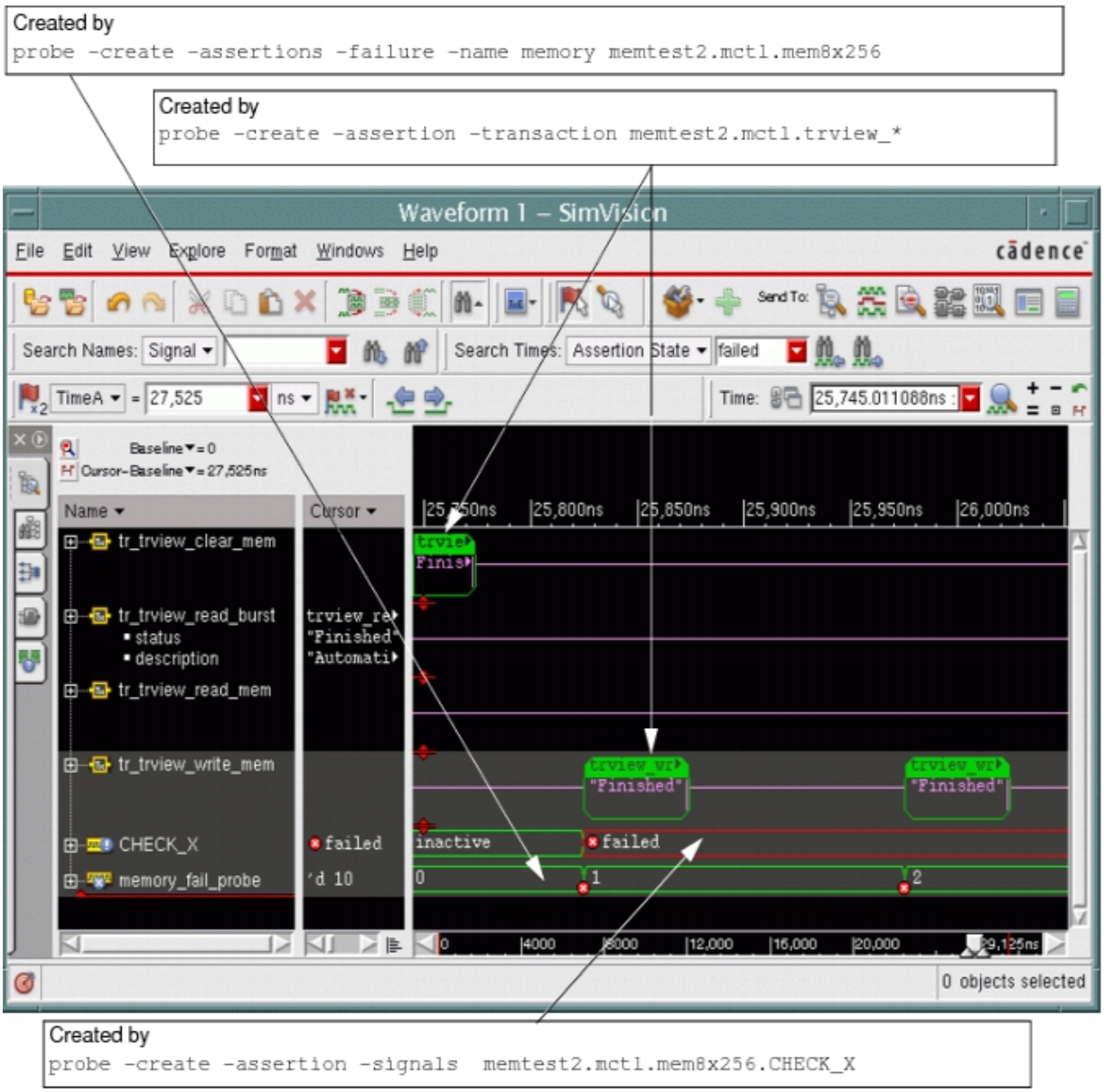
1. Select a module in the left-hand pane.

The elements in that module are displayed in the right-hand pane.

2. Select one or more elements in the right-hand pane and click the waveform button.

An example of the SimVision waveform display of post-processing data is shown in Figure 10-2. Assertions are displayed as they were probed in the `setup.tcl` script--as failure events, state changes, or transactions.

Figure 10-2 Waveform Window



The second waveform trace demonstrates the debugging technique of probing all assertions in a given module to a single trace. For more information about using this technique, see "Creating Assertion Failure Probes using Tcl".

11

Index

Symbols

A

B

C

D

E

F

G

H

N

O

P

R

S

T

V

W