cadence®

Assertion Writing Quick Start

Product Version 16.11

November 2016

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

set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor

Contents

1

An ABV Walkthrough

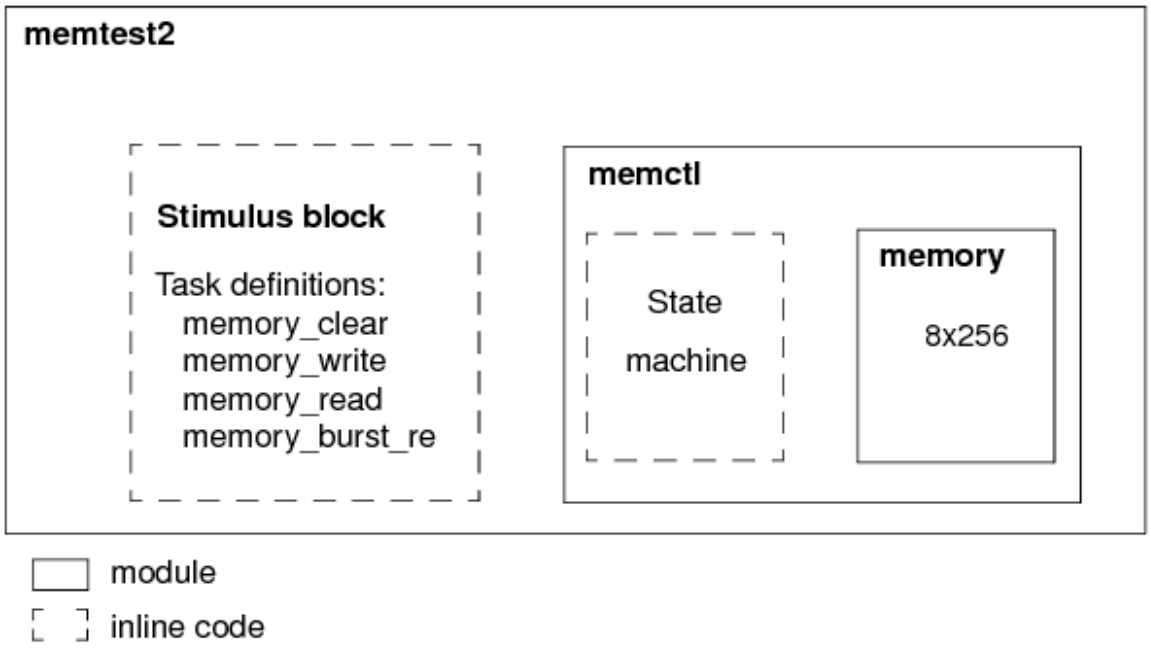
Overview of the ABV Quick-Start Design

The design that is used in the example is a simple memory controller. It has a few errors injected for demonstration purposes. A high-level block diagram of the test environment is shown in Figure 1-1.

In this example, a top-level test initiates various transaction-level tasks. The name of the desired task is passed to the memory controller. The function of the memory controller, `memctl`, is to provide flow control, and to generate the appropriate signal sequences to the memory device.

Most of the assertions in the design are in the memory controller. The behaviors that these assertions describe are the behaviors for which you will try to create assertions. There are also a few assertions in the memory itself that check to see that the memory is being driven correctly. We will focus on these, which will not conflict with the work that you will be doing in subsequent lab exercises.

Figure 1-1 Memory Controller Test Environment



Getting Started with the ABV Quick-Start Example

The design is located in <*Cadence_install_dir*> `/doc/abvquickstart/examples/ demo`. Copy the files from this location to your working directory. Go to that directory and run the simulation by typing:

```
xrun -f run.f
```

The `run.f` command file contains the list of files and compile options to use. It also sources a Tcl file that sets up some example probes, runs the test, and opens the SimVision GUI.

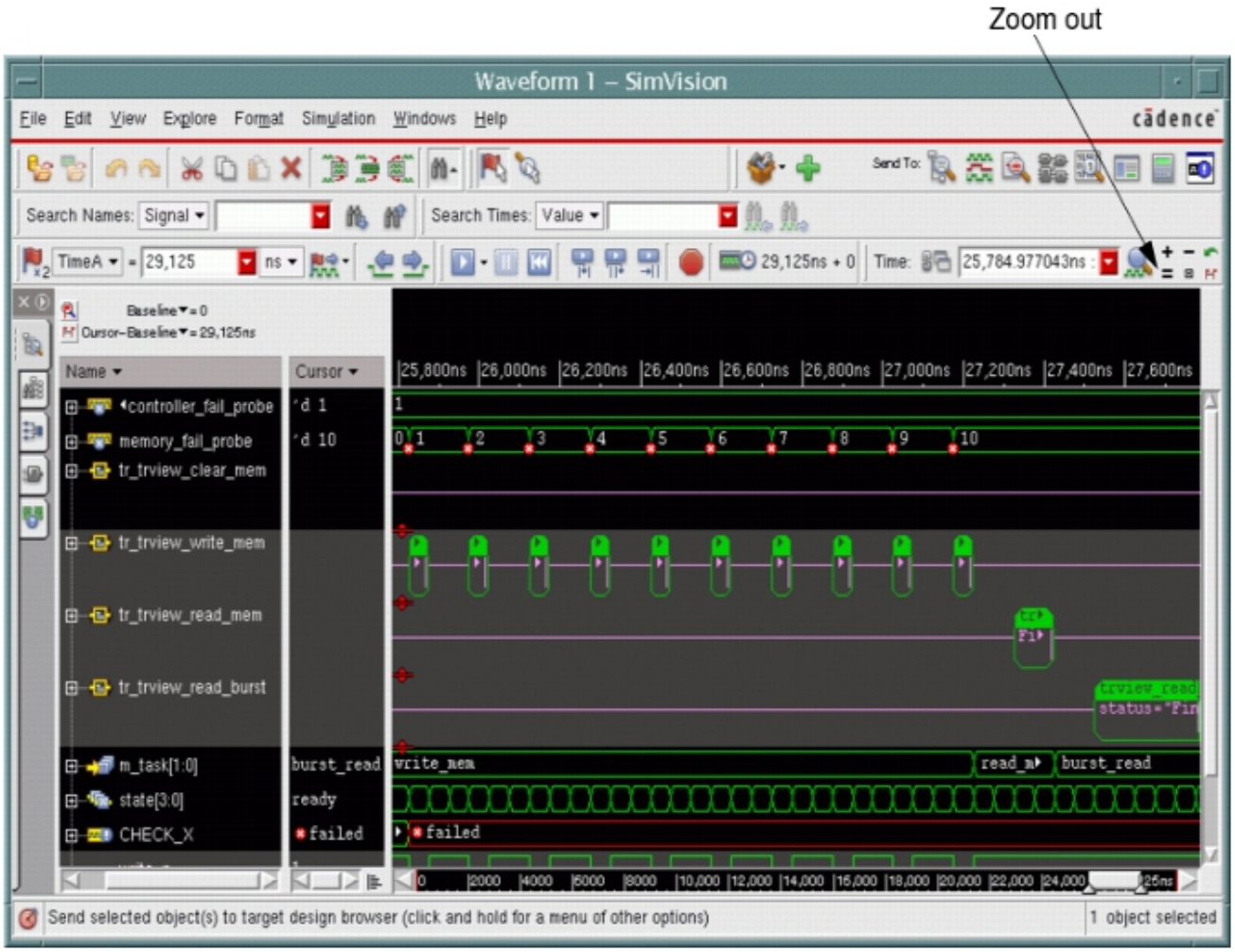
Waveform Viewing in the ABV Quick-Start Demo

When you run the demo simulation, one of the windows that will appear is the SimVision waveform viewer. You can zoom out by using the "=" on the zoom toolbar.

Note: If your waveform does not look the same as the image on the following page, you might have setup SimVision differently, and therefore the same toolbars might not be active. To match this example, from the menu bar, select *View - Toolbars* and select the Cursor Control, Send To, Standard, and Zoom toolbars.

The waveforms that appear will look like the ones shown below.The Probe commands in the Tcl file specify what to display in the waveform viewer.

Figure 1-2 SimVision Waveform Viewer



At the top of the waveform window are assertion failure probes. A failure probe will display a red X at the time when any assertion that is probed to that failure probe fails. You can probe one or more assertions to a failure probe, although the popular use model is to put many assertions on one probe. We provide one failure probe for the memory controller block and another for all assertions in the memory itself. By selecting the failure probe and placing the cursor at the red X (use the blue Next Edge arrow to get to the X) you can select the *Explore* - *Go To* - *Cause* from the menu bar to determine which assertion failed at that point in time.

Notice that there are transactions for the `clear_mem`, `read_mem`, `write_mem`, and `read_burst_mem` tasks, which were recorded from assertions that are probed as transactions. They are useful for understanding what the testbench is doing. The testbench clears the memory, does some memory reads, memory writes, and then a burst read.

Together with the failure probes, you can get a good idea where the problems are. For example, all of the assertion errors in the memory itself occur during the `write_mem` cycle. There appears to be another issue with the clearing of memory, based on the memory controller failure probe.

Assertions can also be probed as state probes, as shown in the `CHECK_X` probe. The valid assertion states include:

- `active`--An assertion attempt is in progress.
- `inactive`--The assertion is being monitored, but there are currently no partial matches of the sequence of conditions described by the assertion.
- `disabled`--The property has been disabled by an SVA `disable iff` expression or a PSL abort expression.
- `failed`--The fulfilling condition for the assertion has evaluated to false.
- `finished`--The fulfilling condition for the assertion has evaluated to true.
- `off`--The assertion is not being monitored due to external controls: a Tcl disable command, a compile-time disable (`-controlassert`, `-noassert`) command, a VPI control, or by using `$assertoff` or `$assertkill control tasks`.
- `suspended`--The assertion is located in a power domain that has been shut off.

Another useful feature is that all types of assertion probes allow you to display all of the signals referenced by the assertion in the waveform.

Assertion Browser for ABV Simulation

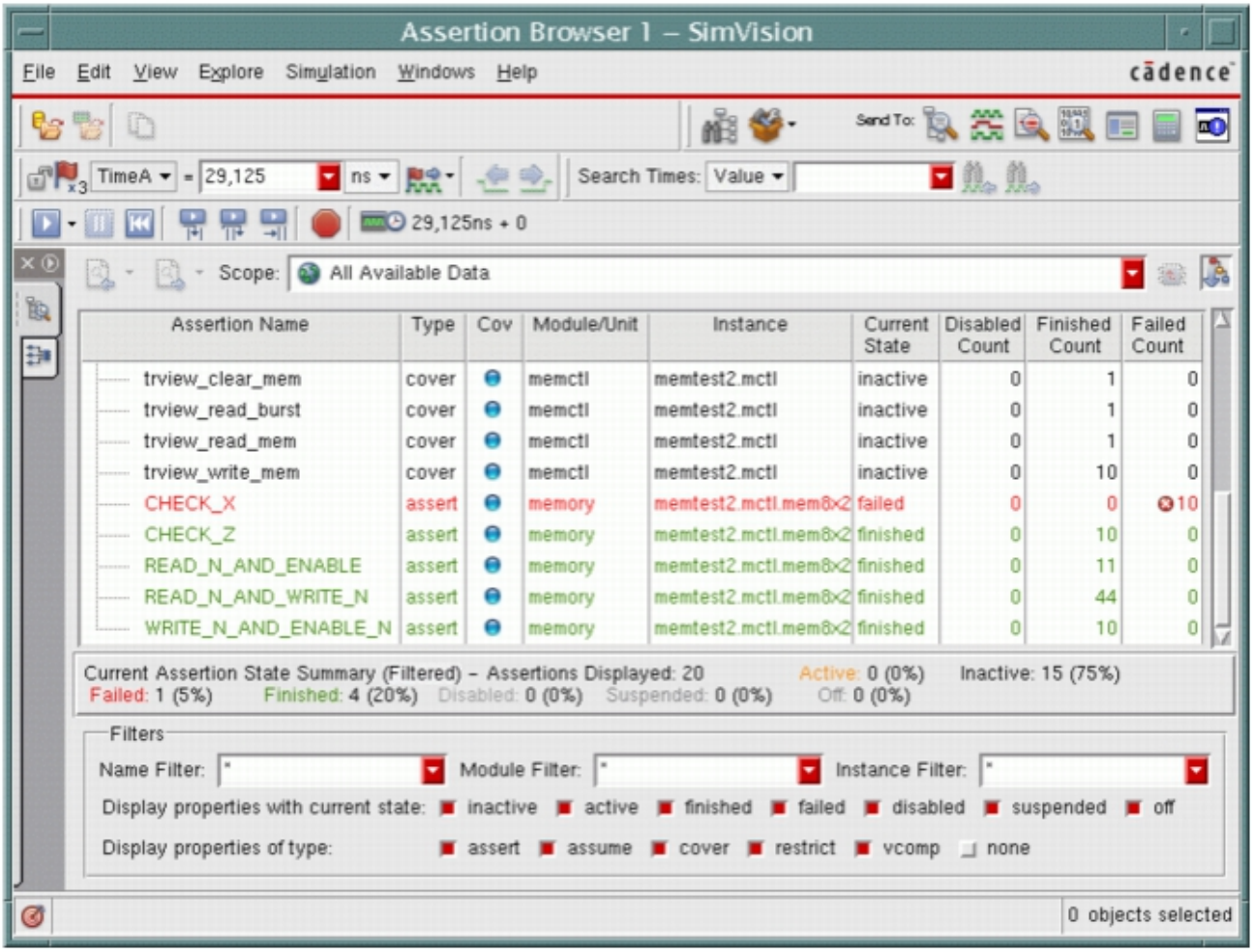
The Assertion Browser lists all the assertions in the design and test.

1. Open the Assertion Browser by selecting the Assertion Browser icon--the rightmost icon on the Send To toolbar--on the waveform viewer.
2. Alternatively, you can also use the *Windows* - *New* - *Assertion Browser* option from the menu bar. The Assertion Browser will show up.
3. Scroll down to see the `CHECK_X` assertion.
4. Click on the column heading to sort assertions. You can drag and drop the columns to any position you want.

You can choose not to view columns if you want, by using *View - Column Editor* from the menu bar. You can also use filters at the bottom to limit what is visible. Assertions can be sorted by:

- Assertion name
- Module
- Instance
- Current state--The state of the assertion at a given simulation time
- Disabled count--The number of times the property has been disabled by an SVA `disable iff` expression or a PSL abort expression
- Finished count--The number of times the assertion evaluated to true
- Failed count--The number of times the assertion revaluated to false

Figure 1-3 Assertion Browser



Select an assertion name and right-click. The pop-up menu that appears shows the operations that can be performed on an assertion, such as setting a breakpoint.

Note: In the operations that can be performed, an assertion is much the same as any signal.

To debug an assertion failure, you will first want to look at the source code for that failure. To do that:

1. Double-click on the name of the assertion. This will open up the Source Browser.
2. The Source Browser can be reached from the waveform viewer through the *Windows* menu, and it can also be reached from the Assertion Browser.

The Source Browser for ABV Simulation

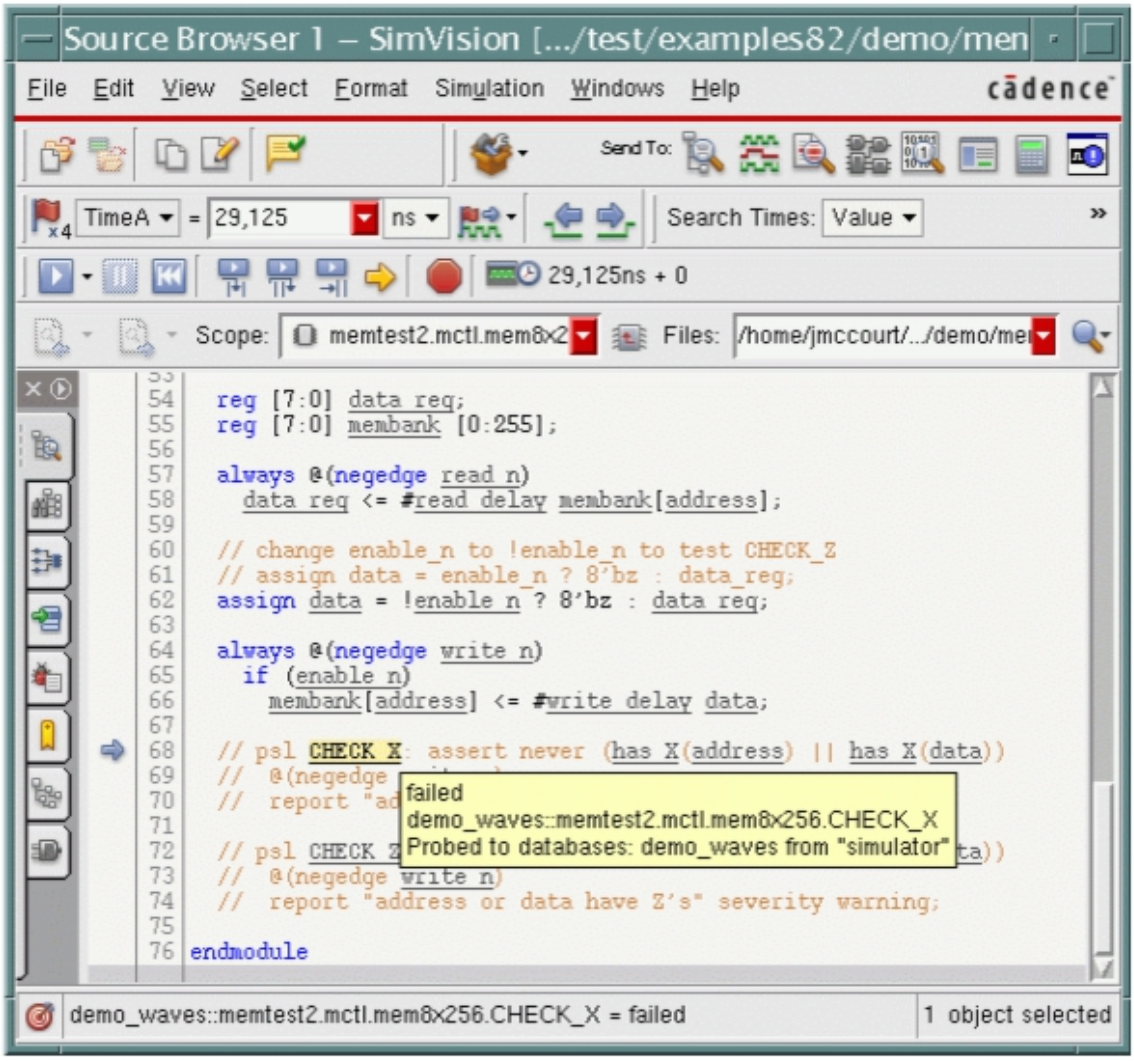
To get to the Source Browser:

1. select the `CHECK_X` assertion and double-click. The Source Browser will show up and appear as shown in Figure 1-4.

   Note: The arrow points to the definition of the selected assertion. You can place your cursor over any signal to see the value at the time of the cursor in the waveform window.
2. Point the cursor to the value of the data during a `write_mem` transaction. You will notice that the data is "`h xx".
3. Select the data, then scroll up to where it is assigned.

You will notice that the data is being assigned on reads instead of writes due to the inversion of the `enable_n` signal. Here the assertion has isolated the cause of failures close to the source. Without the assertion, the test would have to determine the failure by reading the memory and determining that the expected data values were not being read. It would then have to isolate the issue to the read or the write transaction.

Figure 1-4 Source Browser



Another thing to note about the `CHECK_X` definition is that a custom error message and severity level are specified. The default message is:

```
Assertion memtest2.mctl.mem8x256.CHECK_X has failed
```

By using the PSL `report` construct, you can customize the message. In this case, a failure will print the following: "`address or data have X's`." All failure messages are printed to the console and to the log file. You can use the severity to control whether the simulation stops when an error occurs. In this case, the `CHECK_X` assertion has a "warning" severity associated with it. By default, the simulator will not stop for warnings, only errors and failures, but you can specify the severity on which to stop.

2

Introduction to the ABV Quick Start

This tutorial will guide you through the process of writing and simulating assertions in the Cadence Xcelium™ simulator. It provides:

- Basic instructions for using assertions within the Verilog, VHDL, and SystemC simulation environments.
- Exercises to illustrate these concepts.
- Solutions to these exercises.

To get the most out of this book, the following approach is suggested:

1. Read a task description.
2. Perform all the exercises that follow it.
3. Compare your solution with the solution included in the examples directory.

Prerequisites for the ABV Quick Start Tutorial

To run the tutorials, you need to have access to the Cadence Xcelium™ Single Core simulator, software version Xcelium 16.11 or newer.

This tutorial assumes that you are:

- Familiar with Verilog, SystemVerilog [1] , VHDL, or SystemC® for digital design and verification

  > Cadence supports both the SystemVerilog flavor of PSL and SystemVerilog Assertions (SVA).

- Experienced with using the Xcelium simulator
  For more information, see the *SystemVerilog in Simulation*, the *Simulating Your Design*, or the *SystemC Simulation User Guide*.
- Experienced with the SimVision graphical debugging environment, including the waveform window

ABV Documentation

For information that will help you quickly getting started with assertion checking, see the following Cadence manual within your Xcelium installation:

- *Assertion Writing Quick Start* (This manual)
  Describes the process of writing and simulating assertions in the Cadence Xcelium simulator. It provides hands-on exercises to get you started with dynamic simulation of Property Specification Language (PSL) assertions in the Verilog, SystemVerilog, VHDL, and SystemC flavors; and with SystemVerilog Assertions (SVA). Also includes printable quick reference pages for PSL, SVA, and ABV Tool Commands (Xcelium simulator Tcl and GUI commands for assertion simulation).

For detailed information about writing and using assertions, see the following Cadence manuals within your Xcelium installation:

- *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*
  Describes how to enable assertion checking, how to control it, and how to interpret the results.
- *Integrated Coverage User Guide*
  Describes how to use PSL and SVA constructs for control-oriented functional coverage analysis.

1	Cadence supports both the SystemVerilog flavor of PSL and SystemVerilog Assertions (SVA).

3

Using SimVision for ABV Debugging

Getting Started with SimVision for ABV Debugging

> Read this topic first, before starting the ABV quick-start exercises. "Exercise 1: Setting Up ABV Debugging in SimVision" will enable you to practice these techniques with example files.

SimVision allows you to choose the state transitions that you want to log during simulation.

During simulation, the Assertion Browser lets you monitor the state of the assertions in your design. By default, simulation stops whenever an assertion fails. However, you can change this default behavior and choose which assertion states will stop the simulation, as described in "Stopping the Simulation at Assertion State Changes".

> You can use the Assertion Browser to monitor your assertions during simulation. However, if you want to view assertion information in the waveform window, you must probe the signals and assertions that you want to view. These operations are described in "Using the Assertion Browser for Debugging" and "Setting Probes on Assertions".

Using the Assertion Browser for Debugging

The Assertion Browser displays all the assertions compiled into your design. It displays the current state of the assertion--inactive, begun, finished, or failed. In addition, the browser also displays the number of times each assertion has been disabled, finished, and failed. You can also filter the list of assertions and display only those assertions that interest you.

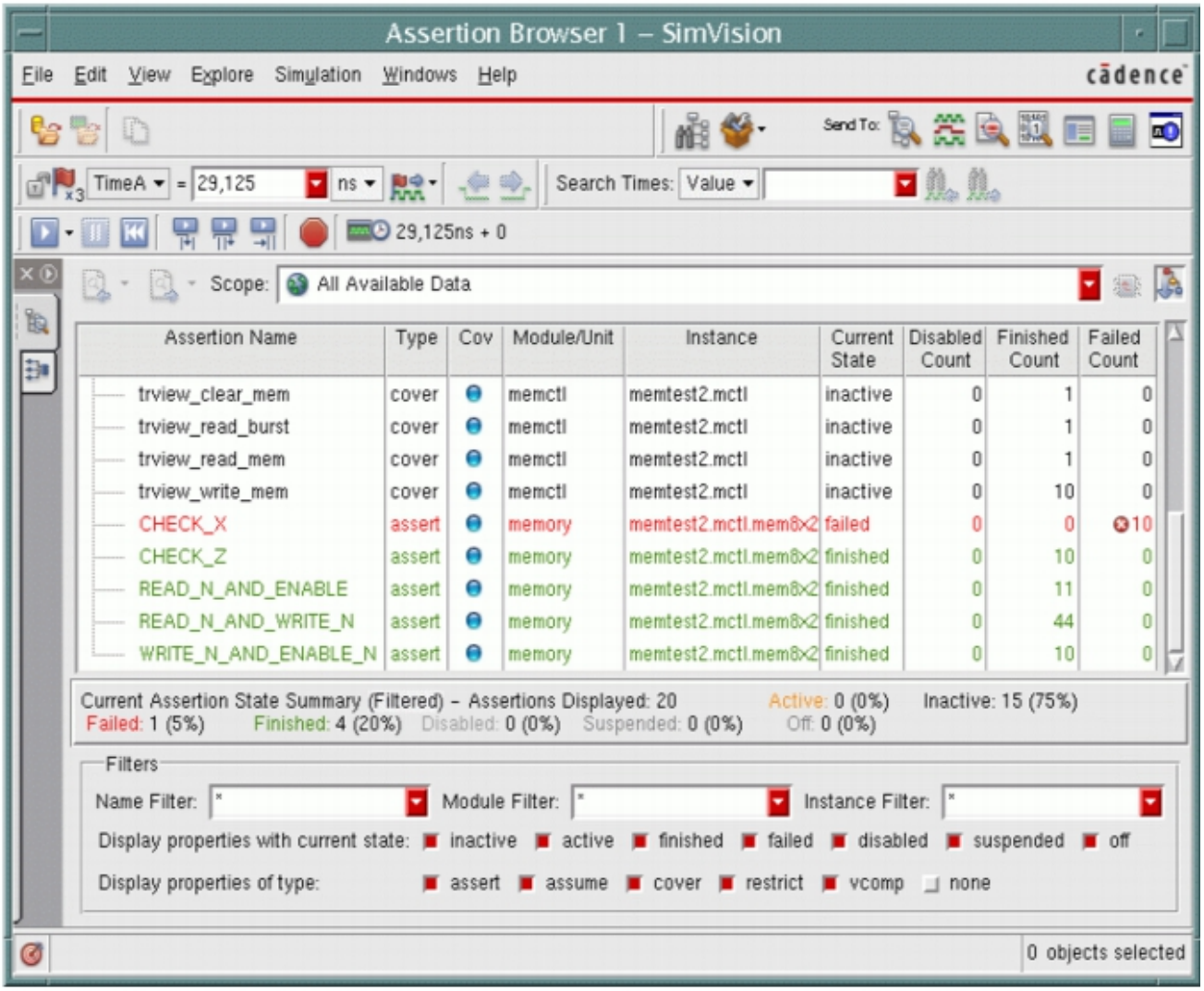
To open the Assertion Browser, do one of the following in a SimVision window:

→ Click the *Assertion Browser* button.

→ Choose *Windows - New - Assertion Browser* from the menu bar.

SimVision opens the Assertion Browser window, as shown in Figure 3-1.

Figure 3-1 Assertion Browser



By default, the Assertion Browser displays all assertions in the design. The bottom of the window contains a filter, where you can specify which assertions you want to display.

Setting Probes on Assertions

When you probe an assertion, you can specify whether the simulator will record this assertion information as a state change event, a failure event, or a transaction.

- A state change probe shows the states and state transitions for an assertion.
- A failure event counts the number of times one or more assertions fail.
- A transaction creates a fiber, which spans the period of time during which the assertion is active.

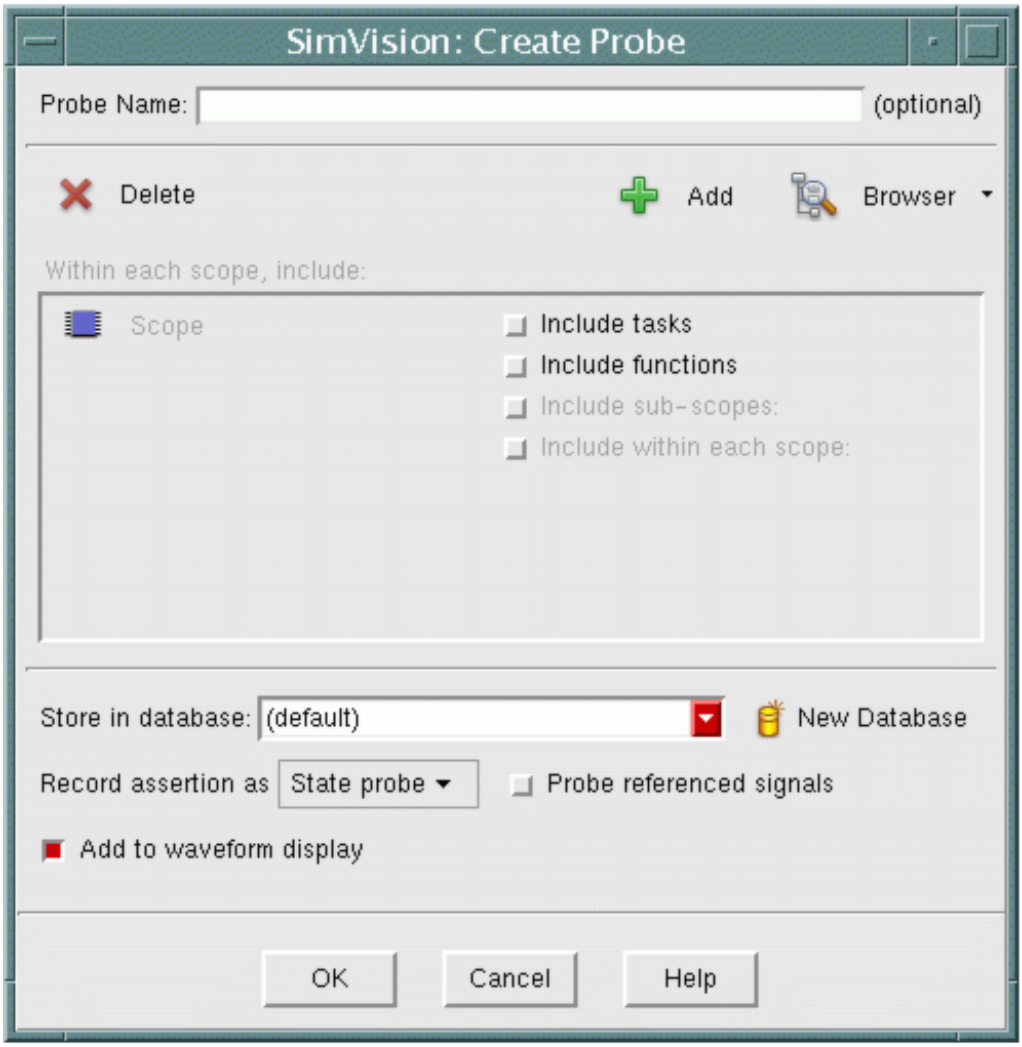
When you probe signals, the simulator saves every value change for those signals during recorded simulation time. You must probe the signals monitored by the assertions if you want to view them in the waveform window along with the assertion information.

There are several ways to create a probe. This tutorial demonstrates how to create a state probe, a single failure probe for all assertions in the Assertion Browser, and a transaction probe.

To create a probe that shows waveform information for an assertion:

1. In the Assertion Browser, right click on an assertion.
2. Choose *Create Probe* from the drop down menu.
   SimVision opens the Set Probe form, as shown in Figure 3-2.

Figure 3-2 Setting a State Probe



3. Optionally, enter a name for the probe in the *Probe Name* field.
   If you do not specify a probe name, by default SimVision will assign state probe the same name as that of the assertion.
4. Set the *Record assertion as* field by clicking on the drop down menu and selecting the type of probe that you need.
5. Optionally, enable the *Probe referenced signals* button by clicking on the checkbox.

   This feature enables you to view all the contributors to an assertion in the waveform window.

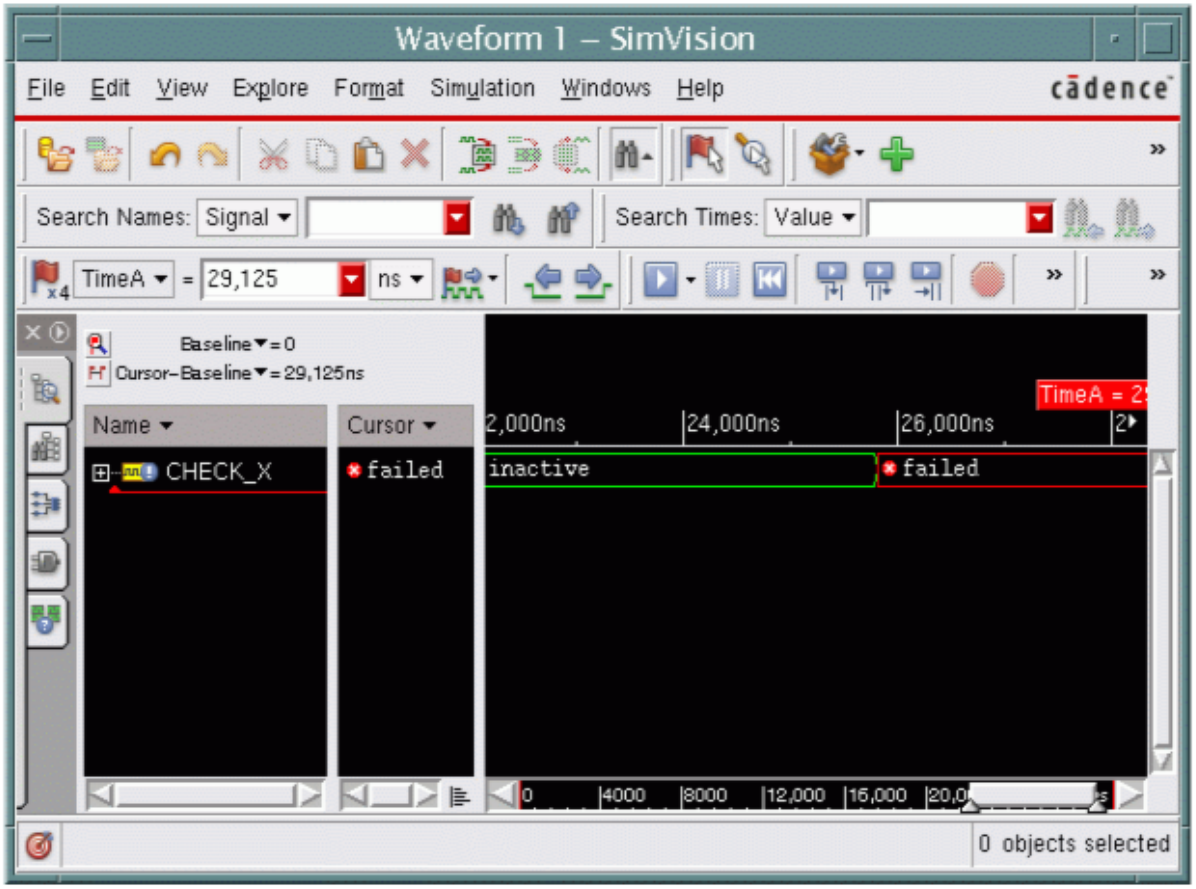
6. Click *OK*.

The probe is displayed in the waveform window, as shown in Figure 3-3. By default, the probe name is the same as the property name.

When you create a probe, SimVision automatically creates a database for the information.

> You can use the *Store in database* field to enter the name of the database where you want to store a certain type of information. During simulation, you can open multiple databases and store different information in each. This is shown in Figure 3-2 .

Figure 3-3 State Probe in the Waveform Window



Stopping the Simulation at Assertion State Changes

At any time during simulation, an assertion can be in `inactive`, `active`, `finished`, `failed`, disabled, suspended, or off state (see "Understanding Assertion States" in the *Assertion Checking in*

Simulation manual).

The simulator logs every time an assertion reaches the `failed` state. It displays a message in the I/O area of the window, and it writes the message to the `xmsim.log` file. However, you can choose to report on any or all states by setting the `assert_output_stop_level` variable.

Exercise 1: Setting Up ABV Debugging in SimVision

This exercise uses the files in < *Cadence_install_dir*> `/doc/abvquickstart/ examples/demo`, as for Chapter 1, "An ABV Walkthrough." For this exercise:

→ Compile and elaborate the design by using the following commands in the `run.f` script, after omitting the Tcl setup file:

```
xrun *.v -sv -access r -assert -propfile_vlog memctl_psl.trview -gui
```

Step 1: Set Different Types of Probes

Here are some ways by which you can try setting probes in SimVision:

- Set a state probe on an assertion, such as `CLEAR_MEM_WRITE_N`. Set a transaction probe on the same assertion. Later, when you run the simulation, notice how these probes display the same information in a different way.
  For more information, see " Viewing Transaction Probes versus State Probes for Assertions " in "Analyzing Assertions," of the *Assertion Checking in Simulation* guide.
- Probe all the assertions in the design to a single failure probe.
  Use *Edit - Select All* from the Assertion Browser menu bar to select all the assertions.

> Combining a large number of assertions into a single failure probe provides a concise way of viewing the failures.Assertion failures are generally rare events, therefore, displaying each assertion on a separate line is inefficient. For information about using this type of probe, see "Creating Assertion Failure Probes using Tcl" in the *Assertion Checking in Simulation* manual.

- Create a transaction probe for an assertion that also probes its referenced signals.
  Transaction probes are appropriate for assertions composed of SEREs (Sequential Extended

Regular Expressions). Transactions appear as fibers in the waveform window.

For more information, see " Setting Up Assertion Probes " in the *Assertion Checking in Simulation* manual.

Step 2: Set Breakpoints

- Set a breakpoint on an assertion by selecting the assertion and choosing the *Simulation - Set Breakpoint - Assertion* from the menu bar.
- Set the global `assert_output_stop_level` variable by choosing *Simulation - Create Debug Variable f*rom the SimVision menu.

> You can also type a Tcl command into the I/O area of the console window to set the `assert_output_stop_level` variable. For example:
> ```
> set assert_output_stop_level failed
> ```
> If you are setting the variable to more than one value, you must use braces to enclose them:
> ```
> set assert_output_stop_level {inactive failed}
> ```

- In the console window, use the Tcl `stop -create -condition` command to set a breakpoint on an assertion. For example:

```
xcelium> stop -create -condition { #mtest2.mct1.START_READ_MEMORY == "active" }
```

Step 3: Simulate

Once you have the probes and breakpoints set up for debugging, you can run the simulation.

You can use the + and - icons given in the upper right corner of the SimVision window to zoom in or zoom out on the selected waveforms.

4

Writing Simple Assertions

This section will help you write some simple assertions about a memory design. After you read the quick primer of basic PSL or SVA assertions that is provided, you can get the example files and begin writing your first assertions.

A Quick PSL Primer for Simple Assertions

Simple assertions describe how you want the design to always or never behave. To specify this behavior, you can use the PSL `always` and `never` operators, followed by a Boolean expression that describes the behavior. The basic syntax for assertions that are embedded in a design is:

```
comment_chars psl assertion_label : assert always | never ( behavior ) ;
```

Use the comment characters and expression syntax of your design language in the PSL statement. For example:

```
//Verilog
// psl WRITE_N_AND_ENABLE_N: assert always (enable_n)
// @(negedge write_n);



-- VHDL
-- psl memcontrol1: assert never (write='1' AND read='1')
-- @(rising_edge( clk )) ;
```

In many cases, you might want an assertion to be evaluated only at a specific event, like a clock edge. To specify an event for evaluating assertions, use:

```
@( event_expression )
```

at the end of your property declaration. If you do not include a clock specification, the assertion will be evaluated any time a signal in the assertion changes.

For details about the PSL syntax, see "Writing PSL Assertions ," in the *Assertion Writing Guide*.

A Quick SVA Primer for Simple Assertions

An assertion in SystemVerilog is a statement that the behavior the assertion describes must be true. Assertions in SystemVerilog are composed of sequences of Boolean expressions with time steps between them. For example, a delay of `##1` specifies one clock tick later than the current clock tick. A delay of `##0` specifies the same clock tick as the end of the first sequence.

```
 assertion_label : assert property (
@( event_expression ) ( behavior ) ;
```

For example:

```
state_is_valid : assert property (
 @(posedge clk) (state >= `ready && state <= `finish_burst));
```

For details about the SystemVerilog Assertions syntax, see "Writing SystemVerilog Assertions ," in the *Assertion Writing Guide*.

Writing Your First Assertions

To use the assertion language concepts that you have just read about, you can now do a hands-on lab exercise.

The following subdirectories hold a simple memory design:

- Verilog--The `memory.v` file in

  `<Cadence_install_dir> /doc/abvquickstart/examples/verilog/lab1`

- SystemVerilog Assertions--The `memory.v` file in

  `<Cadence_install_dir> /doc/abvquickstart/examples/sva/lab1`

- VHDL--The `memory.vhd` file in

  `<Cadence_install_dir> /doc/abvquickstart/examples/vhdl/lab1`

- SystemC--The `memory.h` file in

  `<Cadence_install_dir> /doc/abvquickstart/examples/systemc/lab1`

The memory interface consists of:

- An 8-bit bidirectional data bus
- An 8-bit address bus input
- A low-true `enable_n` input

When low, the data bus is driven by the memory. When high, the data bus can be used as an input to the memory.

- A low-true `read_n` input

  Data is read out of the memory on the falling edge of this signal.

- A low-true `write_n` input

  Data is written into the memory on the falling edge of this signal.

Your task is to add assertions to the `memory` file to test the input signals. These assertions can be used to verify that the memory is being driven properly by other hardware.

Exercise 2: Writing Assertions from Design Requirements

The requirements for the memory interface that you want to check with assertions are:

- The `read_n` and `write_n` lines are never low at the same time.
- The `enable_n` line is high on the falling edge of `write_n`.
- The `enable_n` line is low when the `read_n` line goes low.

In this exercise, write the assertions and verify that they all compile properly.

Note: A property of the form `never (x)` or `always (x)`, where `x` is a simple Boolean expression, is considered combinational and will never finish; it will always be in the active state.

To compile the design, use the following command:

- Verilog and SystemVerilog

  ```
  xrun -sv -assert memory.v
  ```

- VHDL

  First create a `worklib` directory, then run the compiler:

  ```
  mkdir worklib
  xrun -assert -top memory memory.vhd
  ```

- SystemC

  ```
  xrun -assert_sc *.cpp -top memtest1
  ```

For this exercise, verify only that the assertions compile properly. You will verify their behavior in a later exercise. You can see a suggested solution in the following subdirectory:
`<Cadence_install_dir>/doc/abvquickstart/examples/design_language/lab1/solution`

Extra Practice in Writing Assertions

For extra practice, write assertions to verify that the address and data buses do not have any bits with Z or X values on the falling edge of `write_n`.

5

Writing Complex Assertions

You can describe complex behavior as a collection of simpler behaviors. By combining behavioral descriptions to create more complex behavior descriptions, you can construct complex sequential expressions.

PSL Operators

The following PSL operators are useful for defining more complex assertions:

- `->`

  In the following example, the `->` logical-if operator indicates that if `in1` is high, then `in2` must be high in the same sample:

  ```
  // psl IN2_UNTIL_NOT_IN1: assert always (in1 -> in2);
  ```

- `next`

  In the following example, the `next` operator indicates that if `in1` is high, then `in2` must be high in the next sample:

  ```
  // psl IN2_UNTIL_NOT_IN1: assert always (in1 -> next in2);
  ```

- `eventually!`

  In the following example, the `eventually!` operator indicates that if `in1` is high, then `in2` must eventually be high, either in the current sample, or in any later sample:

  ```
  // psl IN2_UNTIL_NOT_IN1: assert always (in1 -> eventually! in2);
  ```

For details about more complex PSL syntax, see "Writing PSL Assertions ," in the *Assertion Writing Guide*.

SVA Operators

The implication operators, `|->` and `|=>`, specify that if the left-hand operand holds, the right-hand operand must also hold, with the following differences:

- `|->`

  If the left-hand side matches the specified sequence, the end point of the match is the starting point for evaluating the right-hand side. In the following example, `d` will be evaluated starting in cycle 4:

  ```
  ASSN_1 : assert property
      (@(posedge clk) (a[*2] ##1 b[*2]) |-> (d) );
  ```

- `|=>`

  If the left-hand side matches the specified sequence, the starting point for evaluating the right-hand side is one clock tick after the end point of the match. In the following example, `d` will be evaluated starting in cycle 5:

  ```
  ASSN_2 : assert property
      (@(posedge clk) (a[*2] ##1 b[*2]) |=> (d) );
  ```

For details about the SystemVerilog Assertions syntax, see "Writing SystemVerilog Assertions ," in the *Assertion Writing Guide*.

Exercise 3: Writing More Complex Assertions

To use the assertion language concepts that you have just read about, you can now do a hands-on lab exercise.

The <*Cadence_install_dir*> /doc/abvquickstart/examples/ *design_language* /lab2 subdirectory holds a modification to the simple memory design. The design now consists of:

Verilog	SystemC	VHDL	SVA	Description
memory.v	memory.h	memory.vhd	memory.v	The memory design used in exercises 1 and 2.

`memctl.v`	`memctl.h`	`memctl.vhd`	`memctl.v`	A memory controller that takes high-level commands from the testbench and generates the handshake signals needed by the memory. The memory module in `memory` is instantiated in `memctl`.

The memory controller is a state machine that accepts commands for the kind of memory task to perform. It then generates all the signals necessary to control the memory for these tasks.

The inputs to the memory controller are:

- A 2-bit bus called `m_task`. The value on this bus specifies what kind of memory task to perform. The legal values are:

0	Clear memory
1	Write memory
2	Read memory
3	Burst read

- An 8-bit bus called `m_addr`. This bus provides the address for the "read memory" and "write memory" tasks, and the starting address for the "burst read" task.
- An 8-bit bus called `m_data`. This bus provides the data for the "write memory" task and the number of bytes to read in the "burst read" task.
- An `m_req` signal. When this signal is high, the controller is instructed to do the task specified by `m_task`.
- A clock called `clk`.

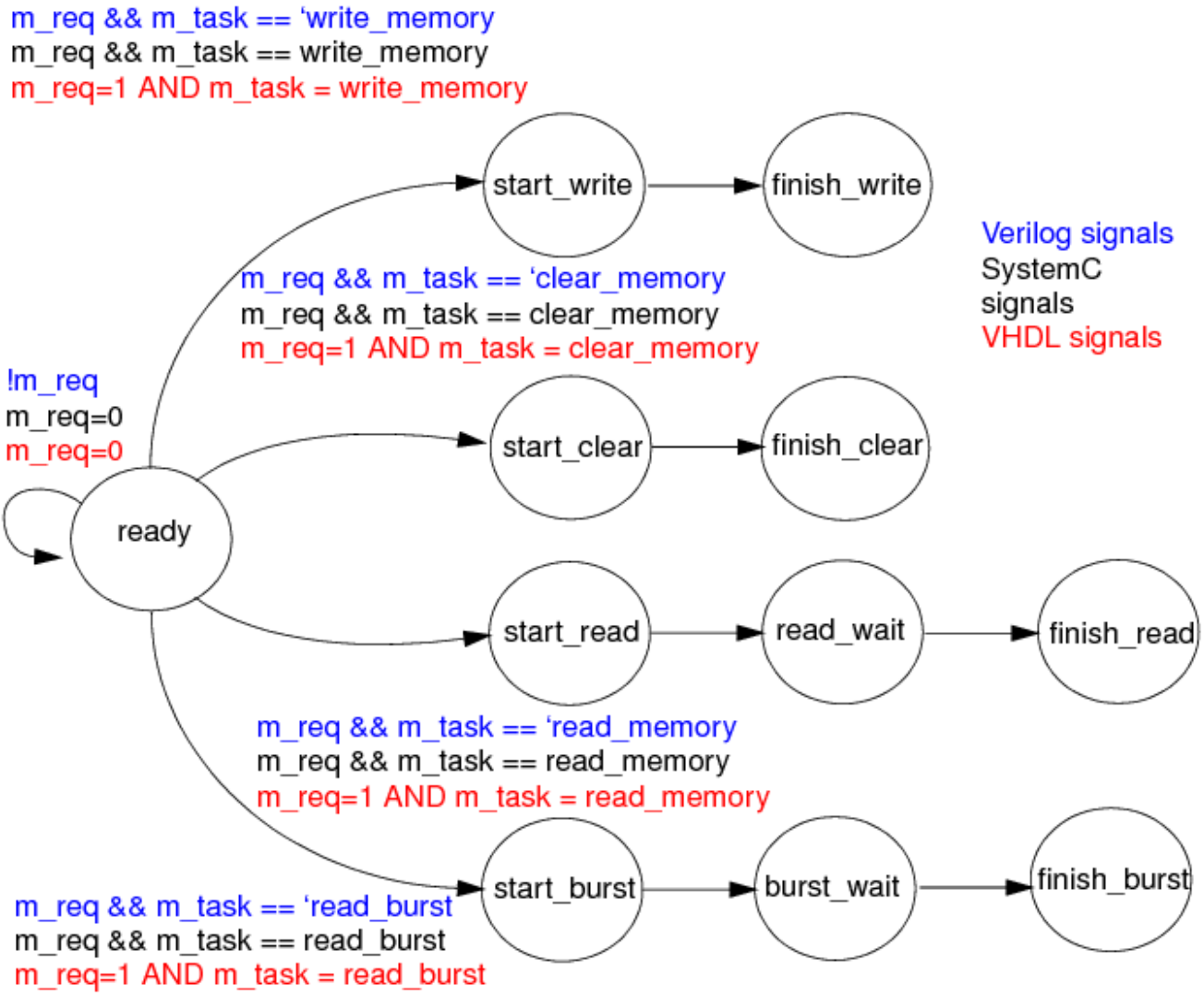
The outputs from the memory controller are:

- An 8-bit `databus` bus. This bus returns the values read by the "read memory" and "burst read" tasks.
- A `readbus` signal. The positive edge of this signal indicates that the `databus` value is valid.
- A `done` signal. The positive edge of this signal indicates that the requested memory task is

complete.

An abbreviated state diagram for the memory controller is shown in Figure 5-1.

Figure 5-1 Partial State Diagram of the Memory Controller



The `m_addr`, `m_task`, and `m_req` lines are only read by the memory controller when the state machine is in the `ready` state. Signals are sampled on the rising edge of `clk`.

In this exercise, you create and test assertions to verify the behavior of some of the signals generated within the memory controller. You will also check some of the behavior of the state machine. This exercise illustrates another use of assertions - to debug the signal values that you generate.

The requirements for the memory controller that you want to check with assertions are the following:

1. Every `m_req` generated by the testbench is answered at some time by the `done` signal going high from the memory controller.
2. At the start of each task, the state machine must branch to the appropriate first state of the state sequence.
3. After the start of each task, the state machine eventually returns to the `ready` state.

This kind of assertion is useful in simulation for generating a transaction that represents an FSM micro-operation--that is, a path through the state space from idle back to idle. It is not very useful as an error check in simulation, as it will only fail if the FSM is not idling at the end of simulation, which might well be the case most of the time.

A more useful assertion for error checking might be one that is bounded in time. For example, the following assertion limits the maximum time allowed for an FSM micro-operation:

```
assert always (state==`idle && start) -> next {{state==`idle} within [*10]};
```

An even more useful technique is, as part of FSM extraction, to generate `cover` directives for each possible path from the idle state back to the idle state, so you can tell which FSM micro-operations were covered in a simulation. Also, each one generates a unique transaction.

Step 1: Add Assertions

Add assertions to the `memctl` file and verify that all assertions compile properly. To compile the design, use the following command:

- Verilog and SystemVerilog

  ```
  xrun -assert -sv memory.v memctl.v
  ```

- SystemC
  See "Step 2: Simulate."
- VHDL

  ```
  mkdir worklib

  xrun -v93 -assert memory.vhd memctl.vhd -top memctl
  ```

Step 2: Simulate

Simulate the design and testbench using the following command:

- Verilog and SystemVerilog

  ```
  xrun -f run.f
  ```

  The `run.f` command file starts the simulation with the correct Verilog files and command-line options. It also runs a Tcl script that probes all of the signals in the design for the waveform window.

- SystemC

  ```
  xrun -assert_sc *.cpp -top memtest2 -gui
  ```

  The `xrun` command file compiles the SystemC files and starts the simulation with the correct command-line options in one step.

- VHDL

  ```
  mkdir worklib
  xrun -f run.f
  ```

  The `run.f` command file starts the simulation with the correct VHDL files and command-line options. It also runs a Tcl script that probes all of the signals in the design for the waveform window.

  Note: You will need to create the required `worklib` subdirectory before you start.

Run the simulation. The design and testbench will simulate without error. And, if your assertions are correct, you will not see any assertion failures.

If you see assertion failures, use the debugging resources in the simulator to debug the assertions.

For more information about using the GUI for debugging, see Assertion Checking in Simulation.

Step 3: Generate Failures

After getting the assertions to run without error, modify the design or the testbench to deliberately generate failures for each assertion. This exercise lets you see how failed assertions can help you in the debugging process.

You can see a suggested solution in the <*Cadence_install_dir*> `/doc/abvquickstart/examples/`*`design_language`*`/lab2/solution` subdirectory.

6

Creating Complex Sequences

Assertions can become hard to read if a behavioral sequence becomes complex. PSL and SVA include a `sequence` declaration, which lets you declare a sequence by name, then use the name in an assertion in place of this sequence. This declaration makes it easy to write complex sequences.

PSL Sequences

The syntax for declaring a PSL sequence is:

```
comment_chars psl sequence name = { SERE } ;
```

To use a named sequence in an assertion, simply put the sequence name in the SERE where you normally put a Boolean equation. For example:

```
// psl sequence READY_STATE          = { state == `ready } ;
// psl sequence START_CLEAR_STATE    = { state == `start_clear } ;
// psl sequence FINISH_CLEAR_STATE   = { state == `finish_clear } ;

// psl CLEAR_MEMORY_TASK: assert always
//         ( { READY_STATE ; START_CLEAR_STATE } |=>
//           { FINISH_CLEAR_STATE [*] } )
//       @( posedge clk ) ;
```

For details about more complex PSL syntax, see "Writing PSL Assertions," in the *Assertion Writing Guide*.

SVA Sequences

The syntax for declaring an SVA sequence is:

```
sequence sequence_name ;
  ( behavior )
endsequence
```

To use a named sequence in an assertion, simply put the sequence name in the expression where

you normally put a Boolean equation. For example:

```
sequence S1;
 (a ##1 term2 ##1 term3);
endsequence

sequence S2;
 (a ##1 CC) or (b ##1 DD);
endsequence

s2_after_s1 : assert property
 (@(negedge clk) (S1) |=> (S2));
```

For details about the SystemVerilog Assertions syntax, see "Writing SystemVerilog Assertions ," in the *Assertion Writing Guide*.

Exercise 4: Writing Sequences

For the next exercise, you will write sequences to test additional aspects of the memory controller that was used in the last exercise. The files for this exercise are in the <*Cadence_install_dir*>`/doc/abvquickstart/examples/` *`design_language`* `/ lab3` subdirectory.

In this exercise, you will write assertions to verify the `write_n` and `read_n` lines that the memory controller drives. This exercise is an example of writing assertions to help verify the behavior of the signals in your design.

The requirements for the `write_n` line that you will check with assertions are:

- When doing the `write_memory` task, the `write_n` line must have the following sequence:
  1. The `write_n` signal is high for one sample
  2. Followed by `write_n` low for one sample
  3. Followed by `write_n` high for one sample

  This assertion is sampled on the positive edge of `clk`.
- When doing the `clear_memory` task, the `write_n` line must have the following sequence:
  1. The `write_n` signal must be high for one sample
  2. Followed by `write_n` low for one sample
  3. This sequence repeats 256 times.

This assertion is sampled on the positive edge of `clk`.

The requirements for the `read_n` line that you will check with assertions are:

- When doing the `read_memory` task, the `read_n` line must have the following sequence:
  1. The `read_n` signal is high for one sample
  2. Followed by `read_n` low for two samples
  3. Followed by `read_n` high for one sample

  This assertion is sampled on the positive edge of `clk` .

Step 1: Add and Check Assertions

Add the assertions to the `memctl` design and verify that they all compile properly. To compile the design, use the following command:

- Verilog and SystemVerilog

  ```
  xrun -assert -sv memory.v memctl.v
  ```

- SystemC
  See "Step 2: Simulate."
- VHDL

  ```
  mkdir worklib

  xrun -v93 -assert memory.vhd memctl.vhd -top memctl
  ```

Step 2: Simulate

Simulate the design and testbench, using one of these commands:

- Verilog and SystemVerilog

  ```
  xrun -f run.f
  ```

  The `run.f` command file starts the simulation with the correct Verilog files and command-line options. It also runs a Tcl script that probes all of the signals in the design for the waveform window.

- SystemC

  ```
  xrun -assert_sc *.cpp -top memtest2 -gui
  ```

  The `xrun` command file compiles the SystemC files and starts the simulation with the correct command-line options in one step.

- VHDL

  ```
  mkdir worklib
  xrun -f run.f
  ```

  The `run.f` command file starts the simulation with the correct VHDL files and command-line options. It also runs a Tcl script that probes all of the signals in the design for the waveform window.

  Note: You will need to create the required `worklib` subdirectory before you start.

When you run the simulation, the design and testbench will simulate without error. If your assertions are correct, you will not see any assertion failures.

If you see assertion failures, use the debugging resources in the simulator GUI to debug the assertions, as described in Chapter 3, "Using SimVision for ABV Debugging."

For more information about using the GUI for debugging, see Assertion Checking in Simulation.

Step 3: Generate Failures

After getting the assertions to run without error, modify the design or the testbench to deliberately generate failures for each assertion. This technique will allow you to see how failed assertions can help you in the debugging process.

Extra Practice in Writing Complex Sequences

Verify the values generated on the `read_n` line during the `burst_read` task. When doing the `burst_read` task, the `read_n` line must have the following sequence:

1. The `read_n` signal must be high for one sample
2. Followed by `read_n` low for two samples
3. This sequence can be repeated up to 32 times

This assertion is sampled on the positive edge of `clk`.

You can see a suggested solution in the <*Cadence_install_dir*> `/doc/abvquickstart/examples/`*`design_language`*`/lab3/solution` subdirectory.

7

Quick References

This section provides links to PDF versions of the Quick Reference cards.

> Clicking on a link starts a viewer to display the PDF file.

PSL

The following are links to PDF versions of the PSL Quick References:

- *PSL Verilog Quick Reference*
- *PSL VHDL Quick Reference*
- *PSL SystemC Quick Reference*

SVA

The following is the link to a PDF version of the SVA Quick Reference:

- *SVA Quick Reference*

ABV Tool Commands Quick Reference

The following is a link to the PDF version of the ABV Tool Commands quick reference guide:

- *ABV Tool Commands Quick Reference*