

Xcelium Simulator Tcl Command Reference

**Product Version XCELIUM AGILE
December 2024**

© 2024 Cadence Design Systems, Inc.

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

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522. All other trademarks are the property of their respective holders. Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522. Open SystemC, Open SystemC Initiative, OSCI, SystemC, and Accellera Systems Initiative Inc. are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission. All other trademarks are the property of their respective owners.

Restricted Permission: This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement permits Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used only in accordance with a written agreement between Cadence and its customer.
2. The publication may not be modified in any way.
3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement.
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly outlined in such agreement, Cadence does not make, and expressly disclaims any representations or warranties as to the completeness, accuracy, or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third-party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from the use of such information.

Cadence is committed to using respectful language in our code and communications. We are also active in the removal and/or replacement of inappropriate language from existing content. This product documentation may however contain material that is no longer considered appropriate but still reflects long-standing industry terminology. Such content will be addressed at a time when the related software can be updated without end-user impact.

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

Contents

| | |
|---|----|
| 1 | 10 |
| The Tcl Command-Line Interface | 10 |
| Tcl Command Syntax | 10 |
| Tcl Command Rules | 10 |
| Command Description Conventions | 11 |
| Tcl Basics | 12 |
| Wildcards Characters in Tcl Commands | 16 |
| Limitations of Using Wildcard Characters | 18 |
| Hierarchy Separators in Tcl Commands | 18 |
| Example for Relative Paths | 19 |
| Example for Absolute Paths | 19 |
| Example for Setting the Debug Scope | 20 |
| Using \$uvm to Represent Logical UVM Pathnames | 21 |
| Syntax of a Logical Pathname | 22 |
| Strings in the UVM Path | 22 |
| The \$UVM Token Used as TCL Output | 23 |
| Executing UNIX Commands from the Tcl Interface | 24 |
| Command-Line Editing in Interactive Mode | 25 |
| Accessing VHDL Signal Attributes Using Tcl Commands | 25 |
| Displaying Debug Settings | 30 |
| Setting Variables | 30 |
| | 36 |
| Suppressing Assert Messages | 63 |
| Examples | 64 |
| 2 | 66 |
| Xcelium Simulator Tcl Commands | 66 |
| alias | 66 |
| Creating an Alias | 69 |
| Displaying the Definition of an Alias | 70 |
| Removing an Existing Alias Definition | 71 |
| analog | 72 |
| assertion | 73 |

| | |
|--|-----|
| Disabling and Enabling PSL/SVA and VHDL Assertions | 86 |
| Listing all the Assertions in a Given Scope | 89 |
| Controlling the PSL/SVA and VHDL Output Logs | 91 |
| Controlling the Level at which VHDL Assertions Stop the Simulation | 94 |
| Printing a Summary of PSL/SVA Assertion Statistics | 96 |
| Specifying an Assertion Directive and Setting the Fail/Finish Limits | 98 |
| attribute | 100 |
| call | 104 |
| call and assert Command Compositions | 106 |
| check | 107 |
| Checking for Bus Contention and Bus Float Conditions | 113 |
| constraint | 114 |
| coverage | 115 |
| cps | 119 |
| ctrandebug | 121 |
| database | 125 |
| Opening a Database | 140 |
| Setting a Default Database | 152 |
| Displaying Information about Databases | 152 |
| Deactivating a Database | 152 |
| Enabling a Database | 153 |
| Creating Incremental SHM Database Files | 153 |
| Closing a Database | 154 |
| deposit | 154 |
| Depositing to Verilog Objects | 173 |
| Depositing to VHDL Objects | 174 |
| Depositing to SystemC Objects | 176 |
| Depositing to IEEE 1801 Supply Nets | 177 |
| Debugging and Random Stability | 177 |
| describe | 178 |
| drivers | 187 |
| Report Formats for driver Command | 202 |
| Displaying the Drivers of Signals | 204 |
| dumpsaf | 204 |
| dumptcf | 232 |
| exit | 248 |
| fault | 248 |

| | |
|--|-----|
| find | 251 |
| Searching for Objects in a Specified Scope | 257 |
| Searching for Objects in All Scopes or a Set Number of Subscopes | 258 |
| finish | 258 |
| fmibkpt | 259 |
| force | 260 |
| force Command Examples | 263 |
| Forcing Verilog Objects | 270 |
| Forcing VHDL Objects | 271 |
| Forcing SystemC Objects | 272 |
| Listing the Forces to Reapply at Power-Up | 273 |
| Forcing the State and Voltage of a Supply Net | 274 |
| fs_strobe | 275 |
| Defining a Single Strobe List | 279 |
| Defining a Dual Strobe List | 280 |
| Setting a Stop On Condition | 282 |
| glitch | 283 |
| heap | 284 |
| help | 291 |
| history | 293 |
| ida_probe | 295 |
| input | 303 |
| logfile | 305 |
| loopvar | 308 |
| maptypes | 314 |
| memory | 315 |
| Memory Objects | 323 |
| Loading Memory | 325 |
| Dumping Memory | 328 |
| Specifying Memory File Format | 331 |
| omi | 332 |
| pause | 334 |
| pause Command Syntax | 335 |
| power | 339 |
| Displaying Information About a Power Domain | 348 |
| Displaying Information About a Specified Power State Table | 353 |
| Displaying a List of Power Domains Within a Specified Power Mode | 356 |

| | |
|--|-----|
| probe | 357 |
| Setting a Probe | 396 |
| Displaying Probe Information | 397 |
| Deactivating a Probe | 399 |
| Probing Different Simulation Objects | 399 |
| Enabling a Probe | 403 |
| Deleting a Probe | 404 |
| process | 405 |
| profile | 411 |
| release | 414 |
| reset | 416 |
| restart | 417 |
| run | 422 |
| Debugging SystemVerilog Randomization Constraint Failures | 428 |
| Stepping Through Lines of Code | 431 |
| save | 431 |
| scope | 437 |
| scverbosity | 448 |
| simtfile | 449 |
| simvision | 450 |
| sn | 452 |
| source | 453 |
| stack | 454 |
| status | 462 |
| stop | 462 |
| Creating a Breakpoint | 496 |
| Specifying Conditions in stop Command Using Tcl Expressions | 511 |
| Hierarchy Separators | 513 |
| Detecting Infinite Loops | 514 |
| Setting a Source Code Line Breakpoint | 516 |
| Setting an Object Breakpoint | 516 |
| Setting a Delta Breakpoint | 517 |
| Setting a VHDL Process Breakpoint | 517 |
| Setting a Subprogram Breakpoint | 517 |
| Deactivating, Enabling, Deleting, and Displaying Breakpoints | 517 |
| strobe | 518 |
| task | 524 |

| | |
|--|------------|
| tcheck | 528 |
| time | 539 |
| value | 544 |
| verisium | 557 |
| version | 558 |
| warn | 559 |
| where | 561 |
| xprof | 562 |
| xprop | 563 |
| ida_database | 565 |
| 3 | 568 |
| Managing the SHM Database | 568 |
| Opening a Database with \$shm_open | 569 |
| Probing Signals with \$shm_probe | 574 |
| Using \$recordvars and Related Tasks | 576 |
| \$recordvars | 577 |
| \$recordfile | 582 |
| \$recordsetup | 583 |
| \$recordon/\$recordoff | 586 |
| \$recordclose | 587 |
| \$recordabort | 587 |
| \$signalscan | 587 |
| 4 | 589 |
| Xcelium VWDB Probing | 589 |
| Commonly Used VWDB Probe Commands | 589 |
| Commonly Used VWDB Probe Examples | 592 |
| Dumping Commands for Xcelium VWDB Flow | 593 |
| \$xmlDumpfile | 593 |
| \$xmlDumpvars | 594 |
| \$xmlDumpMDA | 596 |
| \$xmlDumpMDA | 596 |
| \$xmlDumpon/\$xmlDumpoff | 597 |
| \$xmlDumpvarsByFile | 597 |
| \$xmlDumpMDAByFile | 598 |
| \$xmlSuppress | 599 |
| \$xmlDumpFinish | 600 |

| | |
|--|------------|
| \$xmDumpflush | 601 |
| \$xmDumpSVA | 601 |
| \$xmDumpLog | 602 |
| \$xmDumpRandDebug | 603 |
| Priority of Specifying VWDB Probe Options | 603 |
| Simulator Argument for Global Settings | 605 |
| 5 | 607 |
| The Value Change Dump (VCD) File | 607 |
| Generating a VCD File for a Verilog Design | 607 |
| Generating a VCD File with Tcl Commands | 608 |
| Generating a VCD File with VCD System Tasks | 609 |
| Example Source Description Containing VCD Tasks | 611 |
| Generating a VCD File for a VHDL Design | 612 |
| 6 | 616 |
| The Extended Value Change Dump (EVCD) File | 616 |
| Generating an EVCD File for a Verilog Design | 617 |
| Generating an EVCD File with Tcl Commands | 618 |
| Generating an EVCD File using the \$dumpports System Task | 626 |
| Generating an EVCD File for a VHDL Design | 630 |
| The \$dumpports System Task | 640 |
| The \$dumpports Compatibility | 645 |
| Listing Ports and Port Vector Ranges using \$dumpports | 646 |
| Altering Signal Identifiers | 646 |
| Using the format_flag Argument to Control \$dumpports Output | 646 |
| Using the \$dumpports_close System Task | 649 |
| \$dumpports Restrictions | 649 |
| Format of the EVCD File | 650 |
| Header Information Section | 650 |
| Node Information Section | 651 |
| Value Change Section | 653 |
| Port Names | 654 |
| Drivers | 655 |
| Port Value Character Mapping | 655 |
| Strength Mapping | 657 |
| Appendix A: Extensions of Tcl | 660 |
| Extensions to Tcl | 660 |

| | |
|--|-----|
| Value Substitution | 660 |
| The @ Character and Escaped Names | 661 |
| Example with Parameterized Class Names | 663 |
| Expression Evaluation | 663 |
| Verilog Expressions | 663 |
| Operators Added to Tcl | 664 |
| Concatenation Operators | 665 |
| Logical AND and OR Operators | 666 |
| Equality Operators | 666 |
| Conversion Functions | 667 |
| VHDL Expressions | 667 |
| Enumeration Literals | 668 |
| Enumeration Vectors | 669 |
| Standard Logic Type | 670 |
| VHDL operators | 671 |
| Tcl Functions for Type Conversion | 673 |
| Enabling Tk in the Simulator | 675 |

The Tcl Command-Line Interface

The Xcelium™ Simulator command language is based on the object-oriented Tool Command Language (Tcl). This means that an action to be performed on an object is supplied as a modifier to the command.

Tcl Command Syntax

```
xcelium> command [-modifier] [-options] [arguments]
```

Tcl Command Rules

Consider the following rules when using Tcl commands with Xcelium:

- Commands consist of a command name, which may be followed by *arguments* or a *-modifier*. The command name is always the first or left-most word on the command line. A *-modifier* may also have *-options*.

For example, in the following command, `database` is the object, and `-open` is a modifier:

```
xcelium> database -open
```

- Commands must be entered in lowercase. For instance:

```
xcelium> scope  
board  
xcelium> SCOPE  
xmsim: *E,TCLERR: invalid command name "SCOPE".
```

- Command modifiers can be entered in upper or lowercase.

```
xcelium> scope -history  
xcelium> scope -HISTORY
```

- Commands and command options can be abbreviated to the shortest unique string.

```
xcelium> scope -history
```

```
xcelium> sco -history
xcelium> sco -hi
```

You can enter more than one command on the command line. Use a semicolon to separate each command. However, because of the way that Tcl works, only the output from the last command in a script or on the command line is printed to the screen or to the log file. For instance, in the following example, Xcelium displays only the output of the `help` command.

```
xcelium> status; help
```

In addition to the commands implemented to help debug your design, you can also use any built-in Tcl command.

Command Description Conventions

The following conventions are used in the Tcl command topics:

| Conventions | Description |
|-----------------------|--|
| <code>literal</code> | <p>Nonitalic words using the courier font indicate keywords that you must enter literally. These keywords represent command, modifier, or option names.</p> <p>In the following examples, <code>run</code> and <code>-step</code> are command and modifier names, respectively.</p> <pre>xcelium> run xcelium> run -step</pre> |
| <code>argument</code> | <p>Words in italics using the courier font indicate user-defined arguments for which you must substitute a name or a value. Arguments are case sensitive if used in directory paths or if they refer to Verilog objects.</p> <p>In the following command, you must enter the name of an object for the <code>object_name</code> argument.</p> <pre>xcelium> value object_name</pre> |
| <code>[]</code> | <p>Square brackets denote optional arguments.</p> <p>In the following example, the <code>probe_name</code> argument is optional:</p> <pre>xcelium> probe -show [probe_name]</pre> |
| <code> </code> | <p>Vertical bars (OR-bars) separate possible choices for a single argument.</p> |

| | |
|-----|--|
| { } | <p>Curly braces are used with OR-bars and enclose a list of choices from which you must choose one.</p> <p>For example, the following syntax indicates that one of the three keywords (name, kind, or declaration) <i>must</i> be specified:</p> <pre>xcelium> scope -describe -sort {name kind declaration}</pre> |
| ... | <p>Three dots (...) indicate that you can repeat the previous argument. If they are used with brackets, you can specify zero or more arguments. If they are used without brackets, you need to specify at least one argument, but you can specify more. For example:</p> <pre>argument ... (Specify at least one, but more are possible) [argument ...] (You can specify zero or more)</pre> |

Related Topics

- [Tcl Basics](#)
- [Extensions to Tcl](#)
- [Xcelium Simulator Tcl Commands](#)

Tcl Basics

This topic provides basic information on the Tool Command Language (Tcl) that is used to control the execution of a simulation.

| Tcl Basics | Usage |
|------------|--|
| Comments | <p>Use the # character to create a comment in Tcl. For example:</p> <pre>xcelium> put test1; put test2 test1 test2 xcelium> # put test1; put test2 xcelium></pre> |

| | |
|-----------------------|--|
| Tcl Variables | <p>A simple Tcl variable has a name and a value. Both the name and the value can be arbitrary strings of characters. Variable names are case-sensitive.</p> <p>The value of a variable is always stored as a character string. Tcl variables can be used to represent many things, such as integers, real numbers, names, lists, and Tcl scripts, but they are always stored as strings. For example, if you use the <code>set</code> command to assign a value to a variable, as in:</p> <pre>xcelium> set a 140 140</pre> <p>it is critical to understand that the value of <code>a</code> is the character string <code>140</code>, not the integer <code>140</code>.</p> <p>Variables are created automatically when they are assigned values.</p> |
| Variable Substitution | <p>Placing an unquoted dollar sign character (\$) in front of a variable name triggers variable substitution. All characters following the \$ that are letters, digits, or underscores are treated as a variable name. The \$ and the name are replaced in the word by the value of the variable.</p> <p>In the following example, <code>\$a</code> is replaced with the value of <code>a</code>, the character string <code>140</code>. This value is then assigned to the variable <code>b</code>:</p> <pre>xcelium> set a 140 140 xcelium> set b \$a 140</pre> <p>Braces can be used to delineate the extent of the name. For example, the following command sets <code>c</code> to the value of <code>b</code> with <code>pounds</code> appended.</p> <pre>xcelium> set c \${b}pounds 140pounds</pre> |
| Tcl Commands | <p>A Tcl command consists of one or more <i>words</i>. The first word is the name of the command and additional words are arguments to that command. Words are separated by spaces or tabs.</p> <p>Most Tcl commands have a "result" that is a string. The result depends on how the command was invoked. For commands typed at the prompt, the result is printed on the screen.</p> <ul style="list-style-type: none">• The <code>set</code> Command: The <code>set</code> command is used to create, read, and modify variables. This command takes two arguments, and assigns the value of the second argument to the first argument. <pre>xcelium> set a 24 24 xcelium> set 42 b b</pre> <p>In the second example, a variable named <code>42</code> is created, and is assigned the value <code>b</code>.</p> |

- **The `expr` Command:** The `expr` command is used to evaluate arithmetic expressions. The argument must be an expression. The `expr` command returns the string value of the computed expression.
`xcelium> expr 24/3.2`
7.5

```
xcelium> expr (2+3) * 3  
15
```

- **Command Scripts:** Tcl commands can be combined into scripts by separating the commands with either a newline or a semicolon. For example, suppose that a file called `myscript.tcl` contains the following two commands:

```
set a 20  
set b 40
```

You can source this script with the `source` command, as follows:

```
xcelium> source myscript.tcl  
40
```

The following example shows the same two commands separated with a semicolon:

```
set a 20; set b 40  
xcelium> source myscript.tcl  
40
```

Notice that the result string of a command script is the result of the last command in the script.

Command Substitution

In addition to variable substitution (see [Variable Substitution](#)), Tcl also allows command substitution, which causes part or all of a word to be replaced with the result of a Tcl command.

Enclose the command in brackets to invoke command substitution.

```
xcelium> set inches 20  
20  
xcelium> set cm [expr $inches*2.54]  
50.8
```

The characters enclosed in brackets must be a valid Tcl script. The brackets and all of the characters between them are replaced with the result of the script.

| | |
|-----------------------------------|---|
| Backslash Substitution | <p>Backslash substitution is used in Tcl to insert special characters, such as newlines, [, #, and \$, without them being treated specially by the Tcl interpreter.</p> <p>In the following example, the value 24 is assigned to the variable <code>a</code>. In the second command, <code>\$a</code> is replaced by the value 24, and this is assigned to <code>b</code>. In the third command, however, the <code>\</code> prevents the <code>\$</code> from being treated specially (that is, triggering variable substitution), and therefore the string value of <code>\$a</code> is assigned to <code>b</code>.</p> <pre>xcelium> set a 24 24 xcelium> set b \$a 24 xcelium> set b \ \$a \$a xcelium> set b \#a #a</pre> |
| Quoting Words in a Command | <p>The following are the two ways to <i>quote</i> words in a command: with double quotes (" ") or with curly braces ({ }).</p> <ul style="list-style-type: none">• Using Double Quotes (" "): If you enclose a word in a command in double quotes:<ul style="list-style-type: none">◦ Spaces, tabs, newlines, and semicolons are treated as ordinary characters within the word.◦ Variable substitution, command substitution, and backslash substitution all occur as usual inside the double quotes. <p>For example, the following command sets <code>msg</code> to a string containing the name of a variable, and the value of a numeric expression:</p> <pre>xcelium> set msg "The value of \$b = [expr \$a/2]" The value of \$b = 12</pre> |

- **Using Braces ({ }):** If you enclose a word in a command in braces ({ }):
 - All characters, including spaces, tabs, newlines and semicolons, are treated as ordinary characters.
 - No substitutions are performed.

In the following example, the value of `msg` is the entire string, verbatim, that is enclosed by the braces:

```
xcelium> set msg {The value of \b = [expr $a/2]}  
The value of \b = [expr $a/2]
```

Braces are typically used to prevent the immediate processing of special characters by the Tcl parser. This is called *deferred evaluation*. Special characters are passed to the command procedure as part of its argument. The command procedure then processes the special characters itself, often by passing the argument back to the Tcl interpreter for evaluation.

In the following example, the `proc` command is used to create a procedure that adds two numbers:

```
xcelium> proc add {a1 a2} {  
> set sum [expr $a1+$a2]; return $sum  
> }  
xcelium> add 3 4  
7
```

Because the body of the procedure is enclosed in braces, it is passed verbatim to `proc`. The value of the variables `a1` and `a2` is not substituted when the `proc` command is parsed. This is necessary because a different value must be substituted for these variables each time the procedure is invoked.

Related Topics

- <https://www.tcl.tk/>
- <https://www.activestate.com/products/tcl/>
- [Extensions to Tcl](#)

Wildcards Characters in Tcl Commands

You can use wildcard characters in arguments to some Tcl commands. The wildcard characters are:

- The asterisk (*)

This character stands for any combination of zero or more characters. For example, the pattern `s*n` matches any object name, regardless of length, that starts with the letter `s` and that ends with the letter `n`. Possible matches include `sn`, `sun`, `son`, and `sudden`.

- The question mark (`?`)

This character stands for any single character. For example, the pattern `p?n` matches any three character object name that starts with the letter `p` and that ends with the letter `n`. Possible matches include `pun`, `pan`, and `pin`.

You can use wildcard characters in two types of arguments:

- In the names of objects that you create with commands such as `probe -create` or `stop -create`. When you create a probe or set a breakpoint, the simulator gives the probe or breakpoint either a default name or a name that you specify with the `-name` option. You can use wildcard characters to delete, disable, or enable these probes or breakpoints. For example:

```
xcelium> stop -delete *           ;# Deletes all breakpoints
```

```
xcelium> stop -disable br*        ;# Disables all breakpoints that have names that  
                                   begin with br.
```

```
xcelium> stop -enable break?      ;# Enables all breakpoints that have names that  
                                   begin with break and that have one additional  
                                   character.
```

- In the names of Verilog and VHDL objects in Tcl commands that can take multiple object names as their arguments. When used by itself, `*` matches every object name in the specified scope. If the scope is not specified, it matches every object in the current scope. For example:

```
xcelium> stop -create -object *    ;# Creates a breakpoint on all objects in the  
                                   current scope.
```

```
xcelium> describe mypin*z         ;# Describes all objects in the current scope  
                                   that have names that start with mypin and  
                                   that end with z.
```

```
xcelium> probe -create -shm top.dut.tes* ;# Creates a probe on all objects in  
                                           scope dut that have names that  
                                           begin with tes.
```

```
xcelium> value :process1:pin?z    ;# Displays the values of objects that have  
                                   names that have five characters and that
```

```
begin with pin and end with z in the scope  
:process1.
```

Limitations of Using Wildcard Characters

You cannot use wildcard characters in the following cases:

- In scope specifiers. Wildcards are allowed only in the final path element.

```
xcelium> probe -create -shm top.u*.sig1  
xmsim: *E,PNOOBJ: Path element could not be found :u*.
```

- Inside escaped names.

```
xcelium> stop -create -object :\_sig*\      ;# The wildcard character will be  
                                         considered to be part of the  
                                         escaped name.
```

- To specify array elements, array slice, signal attributes, and record elements. For example, the following commands are not valid:

```
xcelium> value ab*c'delayed  
xcelium> value ab*c[4]  
xcelium> deposit :data_rec_out1.dout1.my* = '0'  
xcelium> deposit :data_rec_in(0).d?n1 = '0' -after 10ns, '1' -after 20ns
```

Related Topics

- [Command Description Conventions](#)

Hierarchy Separators in Tcl Commands

The simulator supports hierarchical designs by allowing models to be embedded within other models. Levels of hierarchy in a design are called *scopes*. To create a scope, you nest objects within design units by instantiating them. Instantiation allows one design unit to incorporate a copy of another into itself.

Each scope in a design hierarchy has a unique hierarchical path name. Furthermore, many Tcl commands take a path to an object as an argument. This path can be one of the following two types:

- **Relative Path:** It assumes the current scope to be the top-level of the design.
- **Absolute Path:** You can use the standard Verilog or VHDL notation:

- If the top-level is Verilog, the path starts with the name of the top-level Verilog module.
- If the top-level is VHDL, the path starts with a colon.

With Xcelium, you can use the period (.), the colon (:), or the slash (/) interchangeably as the hierarchy separator.

Example for Relative Paths

The following examples assume that the current scope is the top-level of the design:

```
;/# Can use ., :, or / as hierarchy separator.
xcelium> scope vlog_dut.vlog_inst
xcelium> scope vlog_dut:vlog_inst
xcelium> scope vlog_dut/vlog_inst

xcelium> scope vlog_dut.vhdl_inst:vlog_inst
xcelium> scope vlog_dut:vhdl_inst/vlog_inst
xcelium> scope vhdl_dut/vhdl_inst/vlog_inst

xcelium> force vhdl_dut/vlog_inst/sig 4'b1111
xcelium> value %b vhdl_dut:vlog_inst.sig
4'b1111
xcelium> release vhdl_dut.vlog_inst:sig

xcelium> probe -shm vlog_dut/vlog_inst/vhdl_inst
Created default SHM database xcelium.shm
Created probe 1

xcelium> describe vlog_dut/vlog_inst/vhdl_inst/vlog_inst/sig
xcelium> describe vlog_dut.vlog_inst.vhdl_inst:vlog_inst/sig

xcelium> drivers -verbose vhdl_dut/vlog_inst/vlog_inst/sig
xcelium> drivers -verbose vhdl_dut:vlog_inst.vlog_inst/sig

xcelium> find -scope vhdl_dut/vlog_inst count*
```

Example for Absolute Paths

The following examples describe how to specify absolute paths for objects in a Tcl command.

Verilog

The path starts with the name of the top-level Verilog module. For example:

```
xcelium> scope vlog_top.vlog_or_vhdl_inst:vlog_or_vhdl_inst
xcelium> scope vlog_top:vlog_or_vhdl_inst.vlog_or_vhdl_inst
xcelium> scope vlog_top/vlog_or_vhdl_inst/vlog_or_vhdl_inst
```

VHDL

The path starts with a colon. For example:

```
xcelium> scope :vlog_or_vhdl_inst:vlog_or_vhdl_inst
xcelium> scope :vlog_or_vhdl_inst.vlog_or_vhdl_inst
xcelium> scope :vlog_or_vhdl_inst/vlog_or_vhdl_inst
```

For VHDL, the `:` that indicates the top level can be replaced with a slash. So, the following two commands are identical:

```
xcelium> force :vlog_or_vhdl_dut/vhdl_inst/sig {"1111"}
xcelium> force /vlog_or_vhdl_dut/vhdl_inst/sig {"1111"}
```

You can also use a language-neutral syntax in which an absolute path begins with a `/`.

- If the top level is Verilog, include the name of the top-level Verilog module after the slash.

```
xcelium> force /vlog_top/vlog_or_vhdl_inst/vlog_inst/sig 4'b1111
xcelium> value %b /vlog_top/vlog_or_vhdl_inst/vlog_inst/sig 4'b1111
xcelium> release /vlog_top/vlog_or_vhdl_inst/vlog_inst/sig
```

- If the top level is VHDL, include the name of the top-level VHDL entity after the slash.

```
xcelium> deposit /vhdl_top_entity/vlog_or_vhdl_inst/vhdl_inst/sig {"1111"}
xcelium> value /vhdl_top_entity/vlog_or_vhdl_inst/vhdl_inst/sig "1111"
xcelium> drivers /vhdl_top_entity/vlog_or_vhdl_inst/vhdl_inst/sig
:vlog_or_vhdl_inst:vhdl_inst:sig.....port : out
std_ulogic_vector(3 downto 0) = "1111"
...
...
```

Example for Setting the Debug Scope

The following examples show how you can traverse the model hierarchy by setting the scope to an instantiated object. using the Tcl command-line interface.

- If the current debug scope is the top level, and you want to scope down one level to a scope called `board`, use the following `scope -set` command:

```
xcelium> scope -set board
```

- If you are at the top level and want to scope down to a scope within `board` called `counter`, use the following command:

```
xcelium> scope -set board.counter
```

- You can also specify a full path name from any debug scope. For example, if the current scope is `board:counter`, you can scope up to the top level (module `top`) with the following command:

```
xcelium> scope -set top
```

- For VHDL, if the current scope is `:vending:coins`, you can scope up to the top level with the following command:

```
xcelium> scope -set :
```

- To set the scope to a subprogram that exists on a call stack, you can include the stack frame number with the scope name. For example:

```
xcelium> scope -set :\${PROCESS}_000[2]
```

- You can also set the scope to the process and then use the `stack -set` command to set the stack level, as shown in the following two commands:

```
xcelium> scope -set :\${PROCESS}_000  
xcelium> stack -set 2
```

Using \$uvm to Represent Logical UVM Pathnames

The top of the UVM hierarchy is called `uvm_top`, which is derived from the `uvm_root` class. The `uvm_top` instance contains one data declaration: `top_levels[$]`, which is a queue of UVM components that are considered to be the top levels of the UVM hierarchy.

The full logical path to a UVM object is: `uvm_pkg::uvm_top.top_levels[xx].leaf_name`, where `[xx]` is the index into the queue, and `leaf_name` is a string that holds the name of the most derived, or leaf-class of the UVM object.

When you make a direct reference to a top-level UVM component, use the `$uvm` token to reference the intermediate UVM hierarchy, instead of specifying the full logical pathname through the UVM design hierarchy to an object.

For example: `$uvm:{a."b"}`.

Syntax of a Logical Pathname

The syntax of a logical pathname using the `$uvm` token is:

```
path := ($uvm: UVMpath)? instpath
instpath := (delimiter instname)*
UVMpath := {string (.string)*}
```

where:

UVMpath is a logical path through the UVM design hierarchy.

instpath is a logical path through the non-UVM portion of the design (the Verilog, VHDL, or SystemC design components).

delimiter is one of the following, as appropriate for the language context:

- `.` (dot)
- `:` (colon)
- `::` (double colon)

instname is an instance name.

`{string.string}` is the target object(s).

The following symbols are used in syntax definition:

- The `?` symbol means zero or one occurrence. It is possible for you to specify an *instpath* with no `$uvm:{...}` component, which would reduce to the standard long form of the logical path.
- The `*` symbol means zero or more occurrences, which covers the case of multiple UVM objects in the path.
- Curly braces in `{string.string}` are required.

Strings in the UVM Path

The string variables in the UVM path are arbitrary names that may not correspond to the underlying SystemVerilog class names. For example, you could create three classes that are derived from `UVM_COMPONENT`:

```
class Class_A extends uvm_component;
    <...>
endclass
class Class_B extends uvm_component;
```

```
<...>
endclass
class Class_C extends uvm_component;
<...>
endclass
```

Then, you can create instances of these classes:

```
Class_A class_a = new( "a", NULL );
Class_B class_b = new( "b.c", NULL );
Class_C class_c = new( "c", NULL );
```

This allows you to use the following syntax to refer to the new objects:

```
$uvm:{a."b.c".c}
```

The quoted string "a.b" is how you encapsulate the dot (.), which is normally a delimiter, into an object specifier.

The path above is equivalent to:

```
uvm_pkg::uvm_top.uvm_top_level.class_a.class_b.class_c
```

Other examples of valid uses of the `$uvm` token are:

- `$uvm:{a."b.c".c}.vlog`: This represents a Verilog instance within a UVM logical path and is equivalent to: `uvm_pkg::uvm_top.top_level.a.b.c.vlog`
- `$uvm:{a}:vhdl1.vlog1.sysC1`: This represents a mixed-language reference and is equivalent to: `uvm_pkg::uvm_top.top_level.a:vhdl1.vlog1.sysC1`
- `$uvm:{uvm_test_top.uart_ctrl_tb0.apb0.bus_collector}`: [This](#) represents a logical path to a UVM component and is equivalent to:

```
uvm_test_top.test_2m_4s'(uvm_pkg::uvm_top.top_levels[0]).uart_ctrl_tb0.apb0.bus_collector
```

The \$UVM Token Used as TCL Output

TCL commands that produce fully hierarchical path references as output, display references into the UVM hierarchy in the following format:

```
$uvm:{UVM_logical_path}path_to_the_object
```

Some Tcl commands that display this output format are:

- **stop**: Creates breakpoints.
- **heap**: Provides information about objects on the heap.
- **scope**: Specifies design scope references.
- **probe**: Saves signal values to a database.
- **value**: Displays values of objects inside of logical paths.
- **describe**: Displays information about a simulation object.
- **force**: Sets objects to specific values.

Executing UNIX Commands from the Tcl Interface

To execute UNIX commands from within the simulator, enter them at the `xcelium>` prompt. You can do this in the command-line interface or the SimVision analysis environment. Command output goes to the log file. If you are using the analysis environment, output goes to the I/O region of the Console window.

For example:

```
xcelium> ls
xcelium> pwd
xcelium> which xmelab
```

You cannot use Wildcard characters for UNIX commands from within the simulator.

You can run UNIX commands in the background by including the ampersand character. For example:

```
xcelium> xterm &
```

You also can use the Tcl `exec` command to run UNIX commands. For example:

```
xcelium> exec xterm &
```

You cannot invoke a program that takes control of the window (such as the `vi` editor) from the `xcelium>` prompt.

Command-Line Editing in Interactive Mode

If you are running the simulation in the single-step invocation mode with *xrun*, you can include the `-e` option to enable command-line editing, which includes the following features:

- The right-arrow and left-arrow keys move the cursor right or left one character on the command line. You can insert new characters at the cursor position.
- The up-arrow and down-arrow keys scroll backward or forward in the command line history buffer.
- The Tab key provides filename completion functionality.
- The Backspace (or Delete) key deletes characters.
- Control-A moves the cursor to the beginning of the command line.
- Control-E moves the cursor to the end of the command line.

Command-line editing works only for interactive mode. Do not use the `-e` option if you are using a debugger. Use the `-e` option while working with the `-linedebug` option.

In the current release, the command line editing functionality is part of the Specman environment. You must have Specman installed, your `path` variable must include: `specman_install_directory/tools.$ARCH/bin`.

Accessing VHDL Signal Attributes Using Tcl Commands

The simulator Tcl interface recognizes nine signal attributes that are defined in the VHDL LRM.

There are two categories of signal attributes:

- **Signal Kind Attributes:** Signal kind attributes create new signals from the signals with which they are associated. These attributes create implicit signals, rather than explicit signals, which are created using signal declarations. These implicit signals return information about the signal to which the attribute is attached, such as whether the signal has been stable for a specified amount of time or when a transaction has occurred on the signal. Because signal kind attributes define new, implicit signals, these new signals can be used anywhere that normal signals are used, including sensitivity lists.

The signal kind attributes are:

- ``DELAYED [(time_spec)]`
- ``STABLE [(time_spec)]`
- ``QUIET [(time_spec)]`
- ``TRANSACTION`

See the VHDL LRM for a complete description of these attributes.

You can access a signal kind attribute only if it has been enabled on the prefix signal. A signal kind attribute is enabled on the prefix signal only if it has been used in the design. For example, if `clk'DELAYED(5 ns)` is used in the design, the signal `clk'DELAYED(5 ns)` can be accessed.

Each signal kind attribute on a particular signal must be explicitly used in the design. That is, using `clk'DELAYED(5 ns)` in the design does not enable `clk'DELAYED(10 ns)` or `clk'TRANSACTION`.

You can access signal kind attributes from the Tcl interface with the following commands:

- `value`
- `stop -object`
- `drivers`
- `describe`

Even though other Tcl commands, such as `probe`, `deposit`, and `force` take signal objects as arguments, you cannot set a probe or force signal kind attributes.

- **Function Signal Attributes:** Function signal attributes are functions that provide information about signals. These attributes return information about the behavior of signals. For example, you can use function signal attributes to report whether a signal has just changed value, the amount of time that has passed since the last event transition, or the previous value of a signal.

The function signal attributes are:

- ``EVENT`
- ``LAST_EVENT`
- ``ACTIVE`

- ``LAST_ACTIVE`
- ``LAST_VALUE`

You can access function signal attributes from the Tcl interface only if they have been enabled. Function signal attributes are enabled in two ways:

1. Function signal attributes that are used in the design are automatically enabled on the prefix signal. All five attributes are automatically enabled on the signal. For example, if `clk'EVENT` is used in the design, all function signal attributes are enabled for `clk`.
2. You can use the `attribute` command to enable function signal attributes for a specific signal(s). All five attributes are automatically enabled on the specified signal(s). The syntax of the `attribute` command is as follows:

```
attribute signal_name [signal_name ...]
```

For example, the following `attribute` command enables all function signal attributes for the signal `d_input`:

```
xcelium> attribute d_input
```

Because of an internal optimization for vector signals, attributes are enabled for all subelements of the vector. For example, suppose that you have the following signal declared in your VHDL code:

```
signal x1 : std_logic_vector (0 to 1);
```

The following `attribute` command enables the attributes for both bits of the vector, even though one subelement is specified.

```
xcelium> attribute :x1(0)
```

If you must enable the attributes only for specific subelements of the vector, elaborate the design with the `-expand` command-line option. The simulator will generate an error if you then use the `value` command on a subelement for which you have not enabled the attributes. For example:

```
xcelium> attribute :x1(0)
xcelium> value :x1(0)'event
FALSE
xcelium> value :x1(1)'event
xcelium: *E, NONFVA: function valued attributes are not enabled for this prefix.
```

Function signal attributes that have been enabled can only be used with the `value` command. For example, if you have enabled attributes for `d_input` using the `attribute` command shown above, you can then use the following `value` command to find out if an event has occurred on the signal during the current delta:

```
xcelium> value d_input 'EVENT'
```

If the `attribute` command is issued after simulation has begun, the attributes assume the default values that they would have had at the start of the simulation. The attribute values will be accurate only after the values have been updated during the simulation.

Function signal attributes that you have enabled on a signal using the `attribute` command remain enabled if you save the current simulation state with the `save` command and then load the saved snapshot with the `restart` command.

Function signal attributes that you have enabled on a signal using the `attribute` command are disabled if the simulation is reset using the `reset` command.

See [attribute](#) for details on the `attribute` command.

Example:

In the following example, two attributes are used: the signal kind attribute ``STABLE` is used on the signal `clock_ph2`, and the function signal attribute ``EVENT` is used on the signal `clock_ph1`.

```
entity attributes_signals is
    port (clock_ph1, clock_ph2 : in bit;
          A, B, C : in bit_vector(2 downto 0);
          Y : out bit_vector(2 downto 0));
end attributes_signals;

architecture RTL of attributes_signals is
begin
    process (clock_ph1, clock_ph2)
        variable S : bit_vector(2 downto 0);
    begin
        if (clock_ph1'event and clock_ph1 = '1') then
            S := A and B;
        end if;
        if (not clock_ph2'stable and clock_ph2 = '1') then
            Y <= S or C;
        end if;
    end process;
end RTL;

% xmvhdl -nocopyright attributes.vhd
% xmelab -nocopyright WORKLIB.ATTRIBUTES_SIGNALS:RTL
% xmsim -messages -tcl -nocopyright WORKLIB.ATTRIBUTES_SIGNALS:RTL
```

```
Loading snapshot worklib.attributes_signals:rtl ..... Done
xcelium> run 100
Ran until 100 FS + 0
xcelium>
# clock_ph2'delayed is not used in the design.
xcelium> value clock_ph2'delayed
xcelium: *E,BASGAT: Attribute not enabled for this signal.
xcelium>
# clock_ph2'stable is used in the design.
xcelium> value clock_ph2'stable
TRUE
xcelium> describe clock_ph2'stable
clock_ph2'stable...signal valued attribute : BOOLEAN = TRUE
xcelium> drivers clock_ph2'stable
clock_ph2'stable...signal valued attribute : BOOLEAN = TRUE
    '0' <- (:) [constant expression associated with port 'clock_ph2']
xcelium>
xcelium>
# clock_ph1'event is used in the design.
# All function signal attributes are enabled for clock_ph1.
xcelium> value clock_ph1'event
FALSE
xcelium> value clock_ph1'last_event
100 FS
xcelium> value clock_ph1'active
FALSE
xcelium> value clock_ph1'last_value
'0'
xcelium>
# clock_ph2'event is not used in the design.
xcelium> value clock_ph2'event
xcelium: *E,NOFVA: function valued attributes are not enabled for this prefix.
xcelium>
# Use the attribute command to enable function signal attributes for clock_ph2.
# This enables all function signal attributes for clock_ph2.
xcelium> attribute clock_ph2
xcelium> value clock_ph2'event
FALSE
xcelium> value clock_ph2'last_value
'0'
xcelium>
# Function signal attributes that you have enabled on a signal using the
# attribute command remain enabled after a save/restart, but are disabled after
```

```
# a reset.
xcelium> save snap1
Saved snapshot worklib.snap1:rtl
xcelium> reset
Loaded snapshot worklib.attributes_signals:rtl
xcelium> value clock_ph2'event
xcelium: *E, NONFVA: function valued attributes are not enabled for this prefix.
xcelium> restart snap1
Loaded snapshot worklib.snap1:rtl
xcelium> value clock_ph2'event
FALSE
```

Displaying Debug Settings

While debugging, you may open databases, set probes, set breakpoints, set aliases, and so on. To display your current debug settings:

- If you are using the Tcl command-line interface, use the appropriate modifier to display information. For most commands, this is the `-show` modifier. For example:

```
xcelium> database -show
xcelium> probe -show
xcelium> stop -show
```

Use the `alias` command without a modifier to display information about aliases you have set, as shown in the following example:

```
xcelium> alias
f2      finish 2
h       history
xcelium>
```

- If you are using the SimVision analysis environment, select *Simulation – Show*. For example, to view information on the breakpoints you have set, select *Simulation – Show – Breakpoints*.

Setting Variables

You can set Tcl variables to help you debug your design. In addition to user-defined variables, the simulator includes several predefined Tcl variables that you can use to control various simulator features.

You can:

- Set a variable or change the value of a variable with the built-in Tcl `set` command.

```
xcelium> set abc 10  
  
xcelium> set vlog_format %b  
  
xcelium> set pack_assert_off {std_logic_arith}
```

- Delete a variable, with the `unset` command.

```
xcelium> unset abc
```

- Display a list of predefined simulation variables and their current values, with the `help -variables` command.

```
xcelium> help -variables
```

- Display a list of all currently set variables, with the `info vars` command. This command does not display variable values.

```
xcelium> info vars
```

You can put variable definitions in an input file and then execute the commands in this file by using the `-input` option when you invoke the simulator. You can also execute these commands by using the `source` or `input` command after invoking the simulator.

The following table describes the predefined Tcl variables:

| Variable | Description |
|---|---|
| <code>assert_1164_warnings = value</code> | Controls the display of warnings from built-in functions from the <code>std_logic_1164</code> package. The <i>value</i> can be <code>yes</code> or <code>no</code> . If you set it to <code>no</code> , the simulator suppresses the warnings. This variable is initially set to <code>yes</code> . |

| | |
|--|---|
| <pre>assert_count_attempts = value</pre> | <p>Enables trace-based counting for assertions.</p> <p>For assertions, attempt-based counting is the default counting method for simulation. The initial value of this variable is 1. Assertion-language models will count successful assertion attempts.</p> <p>You can select trace-based counting by setting the value of this variable to 0. Assertion-language models will count successful traces.</p> <pre>set assert_count_attempts 0</pre> <p>You can also override the default attempt-based counting with the <code>xmsim (or xrun) -assert_count_traces</code> option.</p> <div style="border: 1px solid #fde725; padding: 10px; margin-top: 10px;"> <p>The <code>assert_count_attempts</code> variable has no effect on SystemC PSL assertions. You must use the <code>-assert_count_traces</code> option to enable trace-based counting.</p> </div> |
| <pre>assert_output_stop_level = value</pre> | <p>Specifies the assertion state(s) that will stop the simulation.</p> <p>The <code>value</code> can be set to <code>inactive</code>, <code>finished</code>, <code>failed</code>, <code>none</code>, or <code>all</code>. This variable is initially set to <code>failed</code>.</p> <p>If you do not want to break at any assertion transitions, use the <code>none</code> keyword. Use the <code>all</code> keyword for all states.</p> <p>See <i>Assertion Checking in Simulation</i>, Chapter 3, "Simulating a Design with Assertions" for more information.</p> |
| <pre>assert_report_incompletes = value</pre> | <p>The value of this variable can be set to 1 or 0. The value is initially set to 0.</p> <p>By default, SystemVerilog assertions (SVA) use "weak" semantics for assertion reporting. If the enabling condition of a sequential assertion has been satisfied, but it is incomplete at the end of simulation, the simulator will not report it as a failure.</p> <p>If you set this variable to 1, strong semantics are turned on for both SVA and PSL.</p> |


```
assert_report_level = value
```

Sets the global minimum severity level at which VHDL and PSL assertion messages will be reported. 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, severity, and the name of the design unit in which the assertion occurred.

This variable is initially set to `note`.

You can override the global report level set with the `assert_report_level` variable by using the `assertion -logging` command. For example, the following command logs output for VHDL assertions in scope `:driver1` that have severity level `error` (or higher). The reporting behavior of VHDL assertions in scope `:driver1` will be controlled by the value of the command-line option. This localized value overrides the global report level set with the `assert_report_level` variable.

```
xcelium> assertion -logging -vhdl :driver1 -severity  
error -redirect assert.log
```

| | |
|--------------------------------------|---|
| <pre>assert_stop_level = value</pre> | <p>Specifies the minimum severity level for which VHDL assertions cause the simulation to stop.</p> <p>The <i>value</i> can be <code>note</code>, <code>warning</code>, <code>error</code>, <code>failure</code>, or <code>never</code>.</p> <p>This variable is initially set to <code>error</code>.</p> <p>If the severity level specified in the assertion statement is at or above the severity level specified by this variable, the simulator stops and returns the Tcl prompt.</p> <div style="border: 1px solid #f9e79f; padding: 10px; margin: 10px 0;"> <p>By default, when you invoke the simulator with the <code>-input</code> option, all of the commands in the Tcl script are executed sequentially, just as if each command had been entered at the simulator prompt. The simulator exits after the last command in the input file has been executed. In other words, the simulator will stop if an assertion that is at, or above, the severity level specified with the <code>assert_stop_level</code> variable; but the simulator, by default, will exit at that point only if there are no other commands in the input file. You can change this default behavior by setting the <code>tcl_runerror_exit</code> variable.</p> </div> <p>You can override the global stop level set with the <code>assert_stop_level</code> variable by using the <code>assertion -simstop</code> command. For example, the following command sets the minimum severity level for which VHDL assertions in scope <code>:driver1</code> cause simulation to stop. The <code>-simstop -severity warning</code> option sets a localized stop level for the scope. This overrides the global stop level set with the <code>assert_stop_level</code> variable.</p> <pre>xcelium> assertion -vhdl -simstop -severity warning :driver1 -depth all</pre> |
| <pre>assert_stop_reason</pre> | <p>Identifies which assertion caused a breakpoint in simulation.</p> <p>The <code>assert_stop_reason</code> Tcl variable is intended for use in scripts, to make it easy to determine the reason for an assertion stop. It returns a list containing:</p> |

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

An `expect` statement is 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 SVA, or when there is no PSL report message.

For example, the following PSL 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}}
```

There are several ways to use this variable in a script. For example, you might set a breakpoint on specific assertions of interest, and print the value of the variable when one of the specified assertions fails.

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

See "Interrogating Assertions" in the chapter "Running an Assertion Simulation" in *Assertion Checking in Simulation* for more information about the `assert_stop_reason`

| | |
|------------------------------|---|
| | <p>variable and how it can be used.</p> |
| <pre>autoscope = value</pre> | <p>The value of this variable can be <code>yes</code> or <code>no</code>.</p> <ul style="list-style-type: none">• <code>yes</code>: Set the debug scope to the scope of the current execution point (if any) when the simulator stops.• <code>no</code>: Do not automatically set the debug scope to the scope of the current execution point (if any) when the simulator stops. <p>This variable is initially set to <code>yes</code>.</p> |
| <pre>clean = value</pre> | <p>The value of this variable can alternate between <code>1</code> and <code>0</code> as the simulation runs:</p> <ul style="list-style-type: none">• <code>1</code>: The simulation is in a clean state, and you can use the <code>save -simulation</code> command to create a snapshot of the current simulation state.• <code>0</code>: The simulation is not in a clean state. If you want to save the simulation state, use the <code>run -clean</code> command to run the simulation to the next point at which the <code>save</code> command will work. |

| | |
|---------------------------------------|--|
| <code>display_unit = value</code> | <p>The value of this variable is the time unit used to display time values throughout the user interface.</p> <ul style="list-style-type: none"> • <code>auto</code>: Use the largest time unit in which the time can be expressed as an integer. • <code>module</code>: Use the timescale of the module that is the current debug scope. • <code>FS, 10FS, 100FS, ...</code> • <code>xlstyle</code>: Print time values using the same formatting rules that Verilog-XL uses. XL follows any <code>\$timeformat</code> that is in effect, and, if there is none, formats time values to the smallest <code>`timescale</code> precision. Setting <code>display_unit</code> to <code>xlstyle</code> can make it easier to compare simulation results from the two simulators. Setting the value to <code>xlstyle</code> affects the formatting of time values only. It does not affect the format of messages. <p>The default value is <code>auto</code> unless you have invoked the simulator (<i>xmsim</i>) with the <code>-xlstyle_units</code> command-line option, in which case the default value is <code>xlstyle</code>.</p> |
| <code>heap_garbage_check value</code> | <p>Specifies which simulation events trigger garbage collection. You can specify the following events:</p> <ul style="list-style-type: none"> • <code>SVH_CHK_NEW</code>: On any <code>new</code> operation. • <code>SVH_CHK_ASSIGNALL</code>: On any assignment to a handle. • <code>SVH_CHK_ASSIGN</code>: On any assignment to a handle that decrements a reference count. • <code>SVH_CHK_DEREF</code>: On any dereference operation. <p>For example:</p> <pre>xcelium> set heap_garbage_check SVH_CHK_NEW</pre> <p>The default value is <code>SVH_CHK_ASSIGN</code>. A value of <code>0</code> disables these checks.</p> |

| | |
|--|---|
| <code>heap_garbage_size value</code> | <p>Triggers garbage collection when the size of the heap has increased since the last garbage collection.</p> <p>A positive value specifies an increase in bytes. For example, the following command triggers garbage collection when the heap has increased by five bytes:</p> <pre>xcelium> set heap_garbage_size 5</pre> <p>A negative value specifies an increase in percentage. For example, the following command triggers garbage collection when the heap has increased by 5%.</p> <pre>xcelium> set heap_garbage_size -5</pre> <p>The value of this variable is initially set to <code>-200</code>. Setting this variable to <code>0</code> will disable the check.</p> <p>To see the current value for the <code>heap_garbage_size</code> variable, display the heap system parameters using the <code>heap -show</code> command.</p> |
| <code>heap_garbage_time value</code> | <p>Specifies the wall time, in seconds, to wait before triggering the next garbage collection. For example, the following command triggers garbage collection every <code>7</code> seconds.</p> <pre>xcelium> set heap_garbage_time 7</pre> <p>The default value is <code>0</code>. A value that is less than or equal to <code>0</code> will disable the check.</p> <p>To see the current value for the <code>heap_garbage_time</code> variable, display the heap system parameters using the <code>heap -show</code> command.</p> |
| <code>intovf_severity_level = value</code> | <p>The value of this variable determines whether an integer overflow violation is reported as an error, which stops the simulation, as a warning, or if the violation is ignored.</p> <p>The value of the variable can be set to <code>error</code>, <code>warning</code>, or <code>ignore</code>. By default, the value is set to <code>error</code>.</p> <pre>xcelium> set intovf_severity_level ignore</pre> |

| | |
|---|--|
| <pre>pack_assert_off = value</pre> | <p>The value of this variable specifies the package name(s) from which you want to suppress the display of VHDL assert messages. The value is:</p> <pre>{ [library.]package [[library.]package ...] }</pre> <p>If you specify more than one package, use a space (not a comma) to separate the package names. For example,</p> <pre>xcelium> set pack_assert_off {std_logic_arith numeric_std}</pre> <p>You can specify a library name if there are packages with the same name in different libraries and you want to turn off messages only in a package in a particular library. If you do not specify a library name, assert messages are turned off in all packages with the specified name.</p> <p>Set the <code>severity_pack_assert_off</code> variable to specify the severity level(s) of the messages that you want to suppress.</p> |
| <pre>probe_exclude_patterns = value</pre> | <p>Forces the Tcl <code>probe</code> command to exclude signals or scopes that match the pattern(s) specified in the value argument.</p> <p>In some cases (for example, in the integration between xeDebug (emulation run-time control software) and <i>xmsim</i> under the IXCOM flow), the tools may be using a modified design netlist, which is similar to the original netlist, but which also has additional instrumented objects, like nets or even complete sub-hierarchies. When you probe a hierarchy, you want to filter out all instrumented objects in the hierarchy because they are of no use and only clutter the waveform. These instrumented objects are easy to identify because their name always follows a well-defined pattern (for example, they may all have the prefix <code>_zy</code>). You can set the <code>probe_exclude_patterns</code> variable to exclude these objects when you probe a hierarchy.</p> <p>The <code>value</code> argument is a list of patterns that you want to exclude from subsequent <code>probe</code> commands. The patterns in the list are separated by a space. The two wildcard characters asterisk (*) and question mark (?) are supported.</p> |

For example:

```
set probe_exclude_patterns { _zy* _sd?ag* }
```

The patterns cannot have hierarchical names. Patterns such as the following are not supported with the `probe_exclude_patterns` variable:

```
a.bx*  
a.b.c.xy*  
abc.y*  
abc.y*.c.xyz
```

Setting this variable will:

- Filter out all objects whose name matches the specified pattern regardless of where it is located in the hierarchy.
- Filter out all objects in a scope whose name matches the pattern, and also all objects that are located in nested hierarchies.

For example:

```
set probe_exclude_patterns { _zy* }  
probe -create A.B.C -depth all -vcd
```

This will filter out nets with the prefix `_zy` in scope `A.B.C` and all of its subscopes:

```
A.B.C._zyn1  
A.B.C._zyn12  
A.B.C.D._zyn2  
...
```

This will also filter out the nets in all scopes with a name beginning with `_zy` and all of their subscopes:

```
A.B.C._zycell11.n3  
A.B.C.D._zycell12.n4  
A.B.C.D._zycell12.E.F.mysig  
A.B.C.D._zycell12.E.F._zysig  
...
```

Objects or scopes that match the specified patterns are excluded from all subsequent `probe` commands until the variable is disabled. To disable the variable, set the value to `{ }`.


```
set probe_exclude_patterns {}
```

In the current release, if an object specified on the `probe` command using a full path name matches a pattern specified with the `probe_exclude_patterns` variable, an error is generated and the object is not probed. You can probe the object by resetting the variable first. For example:

```
set probe_exclude_patterns { _zy* _abc* }  
probe -shm -create top._zywirl           ;# Results in an  
error.
```

```
set probe_exclude_patterns { }  
probe -shm -create top._zywirl           ;# Will probe  
top._zywirl
```

probe_screen_format

You can use a Tcl `probe -screen` command to monitor the value changes on the specified object(s) and display the values on the screen. When using this command with multiple signals, a discrete line of output is generated for each signal by default.

Set the value of the `probe_screen_format` variable to 1 if you want to generate multiple signal events on the same line.

```
xcelium> set probe_screen_format 1
```

In the following example, two signals are probed.

```
xcelium> scope top.u1
xcelium> probe -screen a4 a32
xcelium> run 35 ns
```

The following shows the output format of the `probe -screen` command when the `probe_screen_format` variable is set to 0:

```
Time: 0 FS: top.u1.a4 = 4'h0
Time: 0 FS: top.u1.a32 = 32'h00000000
Time: 10 NS: top.u1.a4 = 4'hf
Time: 10 NS: top.u1.a32 = 32'hffffffff
Time: 20 NS: top.u1.a4 = 4'ha
Time: 20 NS: top.u1.a32 = 32'haaaaaaaa
Time: 30 NS: top.u1.a4 = 4'h5
Time: 30 NS: top.u1.a32 = 32'h55555555
```

The following shows the output format of the command with the `probe_screen_format` variable set to 1:

```
Time: 0 FS: top.u1.a4 = 4'h0; top.u1.a32 =
32'h00000000;
Time: 10 NS: top.u1.a4 = 4'hf; top.u1.a32 =
32'hffffffff;
Time: 20 NS: top.u1.a4 = 4'ha; top.u1.a32 =
32'haaaaaaaa;
Time: 30 NS: top.u1.a4 = 4'h5; top.u1.a32 =
32'h55555555;
```

| | |
|---|---|
| <pre>rangecnst_severity_level = value</pre> | <p>By default, a VHDL simulation generates an error, which stops simulation, if a range constraint violation is detected. You can use this variable to specify that simulation should continue.</p> <p>The <code>rangecnst_severity_level</code> variable can have one of three values:</p> <ul style="list-style-type: none">• <code>error</code>: Generate an error and stop the simulation if a range constraint error is detected. This is the default.• <code>warning</code>: Generate a warning and continue the simulation.• <code>ignore</code>: Do not generate a warning and continue the simulation. <p>Example:</p> <pre>xcelium> set rangecnst_severity_level warning</pre> |
| <pre>real_precision = valutex</pre> | <p>The value of this variable determines the number of digits to the right of the decimal point to include when formatting values of type REAL. The value of this variable is initially set to 6.</p> |

```
relax_path_name = value
```

The value of this variable determines the string that is returned when the VHDL ``PATH_NAME` attribute is used.

The VHDL LRM states that the result of the ``PATH_NAME` attribute is "A string describing the hierarchical path starting at the root of the design hierarchy and descending to the named entity, excluding the name of the instantiated design entities."

The value of this variable is initially set to 0. This prints the name of the VHDL top-level entity. That is, the root of the design hierarchy is reported as `:entity_name`. In addition, for a mixed-language design, the result includes the module name and the instance identifier for each Verilog unit. For example, the hierarchical path returned by the ``PATH_NAME` attribute might look as follows:

```
:vhdl_top_test:top:u1:subtractor:u1:vlog_test
```

where:

- `:vhdl_top_test` is the top-level entity name.
- `subtractor` is the Verilog module name for `top:u1`.

If you set the value of the `relax_path_name` variable to 1, the VHDL top-level is reported as `:`, instead of `:entity_name`, and Verilog module names do not appear in the output representing the hierarchy anywhere below the top level. The Verilog module name is printed only if the top-level of the hierarchical path is Verilog. In addition, the language-specific hierarchy separators are used (`.` for Verilog, and `:` for VHDL), and case is preserved for Verilog instance identifiers. For example, if you set the value of the variable to 1, the path shown above is reported as follows:

```
:top:u1.U1:vlog_test
```

Setting the value of this variable to 1 prints hierarchical path strings that match the format required by, or returned by, Tcl commands (for example, the strings match the output of the Tcl `describe` command) and that are more useful with other Xcelium tools.

| | |
|---|---|
| <pre>severity_pack_assert_off = value</pre> | <p>The value of this variable specifies the severity level of the VHDL assert messages that originate from IEEE or user-defined packages that you want to suppress in the simulation run. The value can be one or more of the following:</p> <ul style="list-style-type: none">• failure• error• warning• note <p>The value of this variable is initially set to {warning note}.</p> <p>If you specify more than one severity level, separate the levels with a space. For example,</p> <pre>xcelium> set severity_pack_assert_off {error warning note}</pre> <p>After you have specified the severity level of the assert messages that you want to suppress, you must specify the package names by setting the <code>pack_assert_off</code> variable.</p> |
| <pre>show_force = value</pre> | <p>Enables the display of objects that have been forced to a value.</p> <p>The value of this variable must be set to 1 at the time that the forces are applied to enable the display of forces with a subsequent Tcl <code>force -show</code> command or in the SimVision GUI. Normally, this variable is initialized to 0. However, the variable is automatically set to 1 if you have:</p> <p>Elaborated the design with the <code>-show_forces</code> option (<code>xmelab -show_forces</code>)</p> <p>Invoked the simulator with the <code>-gui</code> option</p> <p>It is recommended that you set this variable to 1 at the beginning of simulation if you intend to use <code>force -show</code> commands.</p> |

| | |
|---|---|
| <code>simvision_attached = value</code> | <p>The value of this variable indicates whether SimVision is being used to control the simulator, either because the simulator was run in GUI mode or because SimVision was attached subsequently with a walk-up connection. A value of 1 indicates that SimVision is being used to control the simulator; a value of 0 indicates that it is not.</p> <p>This variable is read-only. You cannot change its value.</p> |
| <code>snapshot = value</code> | <p>The value of this variable is the name of the currently loaded simulation snapshot. This variable is read-only.</p> |
| <code>strobeFmt = format</code> | <p>The value of this variable specifies the format in which to print the value of objects when using the Tcl <code>strobe</code> command.</p> <p>By default, the simulator prints the values of the specified objects using the default format of the <code>value</code> command. You can specify a format for individual objects by enclosing the object-format pair in curly braces. For example, the following command prints the value of <code>data</code> in binary.</p> <pre>xcelium> strobe -time 100 clk {data %b}</pre> <p>You can also set the <code>strobeFmt</code> variable to specify a global format. For example:</p> <pre>xcelium> set strobeFmt %b</pre> <p>If a strobe is already set, you must reset the strobe in order for the change to take effect.</p> <p>See the strobe Command Examples for an example.</p> |

`strobeHeader = value`

The value of this variable determines if the header information is printed in the tabular output of the Tcl `strobe` command.

The value is initially set to 0. If you interrupt or stop the simulation (with `CTRL-C`, an assert statement, or by running the simulation for a specified period of time, for example), and then continue the simulation, the simulator does not print the header in the tabular output. This is done so that if you send the output to a file with the `-redirect` option, the header does not appear in the output file every time you continue the simulation.

However, if you are sending output to the screen, you might want to redisplay the header. To display the header again, set the `strobeHeader` variable to 1, as follows:

```
xcelium> set strobeHeader 1
```

See the [strobe Command Examples](#) for an example.

| | |
|------------------------------------|--|
| <pre>strobeTimeWidth = value</pre> | <p>The value of this variable controls how much space is used to print the simulation time in the output of a Tcl <code>strobe</code> command. The value is initially set to 15. To change this, set the <code>strobeTimeWidth</code> variable before creating the strobe. For example:</p> <pre>xcelium> set strobeTimeWidth 25</pre> <p>If a strobe is already set, you must reset the strobe in order for this change to take effect.</p> <pre>xcelium> strobe -time 100 clk clr data q Setting up strobe time - '100' xcelium> run 200 ns Time clk clr data q ----- 100 NS 1'h1 1'h1 4'h0 4'hx 200 NS 1'h1 1'h1 4'h1 4'h0 Ran until 200 NS + 0 xcelium> xcelium> set strobeTimeWidth 25 25 xcelium> strobe -time 100 clk clr data q Setting up strobe time - '100' xcelium> run 200 ns Time clk clr data q ----- 300 NS 1'h1 1'h1 4'h2 4'h1 400 NS 1'h1 1'h1 4'h3 4'h2 Ran until 400 NS + 0 xcelium></pre> |
| <pre>tcl_debug_level = value</pre> | <p>Indicates whether to display Tcl output when sourcing scripts with the <code>source</code> command. The value of this variable is initially set to 0 (no display when sourcing a script). Setting the value to 1, as shown in the following command, displays output for the Tcl <code>run</code>, <code>stop -create</code> and <code>stop -show</code> commands.</p> <pre>xcelium> set tcl_debug_level 1</pre> |

| | |
|----------------------------------|--|
| <code>tcl_prompt1 = value</code> | <p>The value of this variable is the command that generates the main Tcl prompt. The default is:</p> <pre>puts -nonewline "xcelium> "</pre> <p>The following command changes the prompt to <code>verilog></code>:</p> <pre>xcelium> set tcl_prompt1 {puts -nonewline "verilog> "}</pre> |
| <code>tcl_prompt2 = value</code> | <p>The value of this variable is the command that generates the prompt you get if you press the <code>Return</code> key before completing a Tcl command. The default is:</p> <pre>puts -nonewline "> "</pre> <p>The following command changes the prompt to <code>Give me more></code>:</p> <pre>xcelium> set tcl_prompt2 {puts -nonewline "Give me more> "}</pre> |

| | |
|--|---|
| <p><code>tcl_relaxed_literal value</code></p> | <p>Enables the use of <i>radix#number</i> syntax:</p> <ul style="list-style-type: none"> For the value in Tcl <code>force</code> and <code>deposit</code> commands, as shown: <pre> force sig 2#1011 Binary radix force sig 8#17 Octal radix force sig 10#8 Decimal radix force sig 16#c Hexadecimal radix </pre> <div style="border: 1px solid #f0e68c; padding: 10px; text-align: center; margin: 10px 0;"> Base 10 is not supported for VHDL. </div> In equality expressions of the form <i>signal == radix#number</i>. For example: <pre> stop -condition {#/top/sig == 2#1011} stop -condition {#/top/sig == 8#11} stop -condition {#/top/sig == 16#b} stop -condition {#/top/sig == 10#11} Not supported if sig is a VHDL signal. stop -condition {#%b/top/sig == 2#1011} stop -condition {#%d/top/sig == 8#11} stop -condition {#%x/top/sig == 8#13} </pre> In X and Z comparisons using <i>signal === radix#number</i> syntax. For example: <pre> stop -create -condition {#tb.d1.din === 1'bx} - continue stop -create -condition {#tb.d1.din === 2#x} - continue </pre> <p>This variable is initially set to 0. Set it to 1 to enable the functionality.</p> |
| <p><code>tcl_runcmd_interrupt = value</code></p> | <p>This TCL variable can assume only two values, "<i>exit_block</i>" and "<i>next_command</i>," and acquires the default value as "<i>next_command</i>."</p> <p>The run command inside an if-block gets interrupted when a <i>Ctrl+C</i> signal is passed, commands after the run command inside the if-block get skipped, and the tool</p> |

eventually returns to the Xcelium prompt if set to *"exit_block"*.

test.sv

```
module loop_test();
reg a,b;
always @(*) begin
    a = b;
    a = 1'b0;
end
always @(*) begin
    b = a;
    b = 1'b1;
end
initial begin
    #1 b = 1;
    #1
    $finish;
end
endmodule
```

sim.tcl

```
set a 10
puts $tcl_runcmd_interrupt
set tcl_runcmd_interrupt "exit_block"
puts $tcl_runcmd_interrupt
if {$a == 10} {
    run;
    puts "Hello1"
}
puts "Hello2"
exit
```

Scenario 1:

Run the test case using the following command:

```
xrun -clean test.sv -input sim.tcl -access +rwc -
line
```

Pass a *Ctrl+C* signal after some time.

Output:

```
xcelium> set a 10
10
xcelium> puts $tcl_runcmd_interrupt
next_command
xcelium> set tcl_runcmd_interrupt "exit_block"
exit_block
xcelium> puts $tcl_runcmd_interrupt
exit_block
xcelium> if {$a == 10} {
>     run;
>     puts "Hello1"
> }
^CSimulation interrupted at 1 NS + 0
./test.sv:9    b = 1'b1;
xcelium>
```

If the TCL variable `tcl_runcmd_interrupt` is set to "exit_block", then, as the *Ctrl+C* signal is passed, the run command inside the if-block gets terminated and eventually returns to the Xcelium prompt. Otherwise, the TCL command specified after the run command inside the if-block runs next.

Scenario 2:

Below is an example of a TCL file *sim1.tcl* and an HDL file *test.sv*, both of which run indefinitely, the same as the above example.

sim1.tcl

```
set a 10
puts $tcl_runcmd_interrupt
set tcl_runcmd_interrupt "next_command"
puts $tcl_runcmd_interrupt
if {$a == 10} {
    run;
    puts "Hello1"
}
puts "Hello2"
exit
```

Run the test case using the following command:

```
xrun -clean test.sv -input sim1.tcl -access +rwc -
```

line

Pass a *Ctrl+C* signal after some time.

Output:

```
xcelium> set a 10
10
xcelium> puts $tcl_runcmd_interrupt
next_command
xcelium> set tcl_runcmd_interrupt "next_command"
next_command
xcelium> puts $tcl_runcmd_interrupt
next_command
xcelium> if {$a == 10} {
>     run;
>     puts "Hello1"
> } ^CHello1
xcelium>
```

`tcl_runerror_exit = value`

Specifies whether the simulator, when invoked with a Tcl script (using the `-input` option), should exit after encountering a condition that would stop a simulation, or whether the simulator should continue to execute commands in the input file after encountering the condition.

By default, when you invoke the simulator with the `-input` option, all of the commands in the Tcl script are executed sequentially, as if each command had been entered at the simulator prompt. If an error or some other condition that would stop the simulation is encountered, the appropriate message is issued, and subsequent commands in the file are executed. If there are no other commands, the simulator exits.

For example, suppose that you have set the

`assert_stop_level` variable to `failure` to specify the severity level of assertions that will cause the simulator to stop. The Tcl input file is as follows:

```
set assert_stop_level failure
run
```

When you invoke the simulator, both commands are executed. The simulator stops when it encounters the assertion failure (at time 3 ps in this example). The simulator then exits because there are no other commands to execute.

```
% xmsim -input input.tcl worklib.testb
xcelium> set assert_stop_level failure
failure
xcelium> run
ASSERT/FAILURE (time 3 PS) from process
:inst:$PROCESS_000
(architecture worklib.test:behave)
FIFO blah blah
Assertion at 3 PS + 0
./assert.vhd:27          assert (dummy >= tDS)
xcelium> exit
%
```

If there are subsequent commands in the input file, those commands will be executed. The simulator does not exit until the last command has been executed. For example:

```
set assert_stop_level failure
run
run 10 ps
run 10 ps
```

```
% xmsim -input input.tcl worklib.testb
xcelium> set assert_stop_level failure
failure
xcelium> run
ASSERT/FAILURE (time 3 PS) from process
:inst:$PROCESS_000 (architecture
```

```
worklib.test:behave)
FIFO blah blah
Assertion at 3 PS + 0
./assert.vhd:27          assert (dummy >= tDS)
xcelium> run 10 ps
Ran until 13 PS + 0
xcelium> run 10 ps
ASSERT/FAILURE (time 15 PS) from process
:inst:$PROCESS_000 (architecture
worklib.test:behave)
FIFO blah blah
Assertion at 15 PS + 0
./assert.vhd:27          assert (dummy >= tDS)
xcelium> exit
%
```

The `tcl_runerror_exit` variable is initially set to `false`. Set the variable to `true` to force the simulator to exit when it encounters a condition that stops the simulation, ignoring subsequent commands in the Tcl input file.

`tcl_simcmderror = value`

This variable is read-only. The initial value at the start of the simulation is `0`, and the value is updated after the execution of every simulation command (for example, `run`, `value`, `scope`, and so on). The variable is set to `1` when there is a simulation error.

This predefined variable can be used in scripts to check if commands executed successfully, or if some appropriate error handling is required.

Only simulation commands change the value of the variable. Standard Tcl commands and Tcl errors do not change the value. For example:

```
xcelium> scope -show
Directory of scopes at current scope level:
    module (m16), instance (counter)
    module (m555), instance (clockGen)

Current scope is (board)
Highest level modules:
board
xcelium> puts $tcl_simcmderror
0
xcelium> sceop -set dwe
xcelium: *E,TCLERR: invalid command name "sceop".
xcelium> puts $tcl_simcmderror
0
xcelium> scope -set dwe
xcelium: *E,PNOOBJ: Path element could not be found:
dwe.
xcelium> puts $tcl_simcmderror
1
```


| | |
|--|--|
| <code>textio_severity_level = value</code> | <p>By default, a VHDL simulation generates an error, which stops simulation, if a <code>TEXTIO</code> error is detected. You can use this variable to specify that simulation should continue.</p> <p>The <code>textio_severity_level</code> variable can have one of three values:</p> <ul style="list-style-type: none">• <code>error</code> – Generate an error and stop the simulation if a <code>textio</code> error is detected. This is the default.• <code>warning</code> – Generate a warning and continue the simulation.• <code>ignore</code> – Do not generate a warning and continue the simulation. <p>Example:</p> <pre>xcelium> set textio_severity_level warning</pre> |
| <code>time_unit = value</code> | <p>The value of this variable is the unit to be used in time or cycle specifications that do not contain an explicit unit. The value can be: <code>FS</code>, <code>10FS</code>, <code>100FS</code>, ... , <code>SEC</code>, <code>MIN</code>, <code>HR</code>, <code>DELTA</code>, or <code>module</code>. For example, if <code>time_unit = NS</code>, the following command runs the simulation for 100 ns.</p> <pre>xcelium> run 100</pre> <p>This variable is initially set to <code>module</code>, which uses the timescale of the module that is the current debug scope.</p> |
| <code>time_scale = value</code> | <p>The value of this variable is the timescale of the current debug scope. This variable is read-only.</p> |

| | |
|--|--|
| <code>vcd_compact_mode</code> | <p>When the <code>vcd_compact_mode</code> is set to 1, wires will always be dumped in the standard VCD format irrespective of the presence of <code>-expand</code> option during elaboration or any internal implementation considerations, as shown below:</p> <pre>wire [0:5] w;</pre> <p>VCD File contents:</p> <pre>\$scope module top \$end \$var wire 6 ! w [0:5] \$end \$dumpvars b111010 !</pre> <p>This new functionality will be supported only when TCL variable "<code>vcd_compact_mode</code>" is set to 1, before any of the VCD probing related TCL commands are executed.</p> <div style="border: 1px solid #fde725; padding: 10px; margin-top: 10px;"> <ul style="list-style-type: none"> • TCL variables are case sensitive. Hence, <code>vcd_compact_mode</code> will only be supported when specified in small letters. • This is not applicable on VHDL signal/variable. </div> |
| <code>verisium_attached = value</code> | <p>The value of this variable indicates whether Verisium Debug is being used to control the simulator. A value of 1 indicates that Verisium Debug is being used to control the simulator; a value of 0 indicates that it is not.</p> <p>This variable is read-only. You cannot change its value.</p> |
| <code>vhdl_format = value</code> | <p>The value of this variable is the format for the output of VHDL values in describe output, stop point messages, and expression results. The value can be set to <code>%h</code>, <code>%x</code>, <code>%d</code>, <code>%o</code>, or <code>%b</code>, or <code>%v</code>. This variable is initially set to <code>%v</code>.</p> |

`vhdl_vcdmap` *value*

The value of this variable is a user-defined mapping of VHDL `std_logic` values to the four states for VCD (1, 0, X, Z). By default, the nine values of `STD_LOGIC` (U, X, 0, 1, Z, W, L, H, -) are mapped to (X, X, 0, 1, Z, X, 0, 1, X).

You can set the `vhdl_vcdmap` variable to define your own mapping. The value is a string of nine valid VCD values. For example:

```
xcelium> set vhdl_vcdmap XXXXZ1111
```

The specified mapping will be used for all VCD databases that you open.

You can also specify a user-defined mapping by using the `-vcd -vcdmap` option when you open a VCD database. For example:

```
xcelium> database -open myvcd.vcd -vcd -vcdmap  
XXXXZ1111
```

If a VCD mapping is defined by setting the Tcl variable and by specifying the `database -vcd -vcdmap` option, the mapping defined by the `database` command is used.

`vital_timing_checks_on value`

The value of this variable controls whether or not VITAL timing checks are run.

You can use the `vital_timing_checks_on` variable to turn VITAL timing checks on or off at any point during the simulation.

The *value* of the `vital_timing_checks_on` variable can be:


- 1: Run VITAL timing checks. The variable is initially set to 1, which means that VITAL timing checks are on.

```
xcelium> set vital_timing_checks_on 1
```

- 0: Do not run VITAL timing checks.

```
xcelium> set vital_timing_checks_on 0
```

The `vital_timing_checks_on` variable affects both accelerated and unaccelerated VITAL cells.

 If you have elaborated the design with the `-notimingchecks` option (`xmelab -notimingchecks`), all timing checks are disabled. You cannot use the `vital_timing_checks_on` variable to re-enable the VITAL timing checks. The simulator will generate an error if you elaborate with `-notimingchecks` and then try to enable VITAL timing checks with:

```
xcelium> set vital_timing_checks_on 1
```

| | |
|---|--|
| <pre>vlog_code_show_force = value</pre> | <p>When the <i>value</i> of this variable is set to 1, then the output of a Tcl <code>force -show</code> command will include any objects that have been forced by a Verilog procedural force continuous assignment.</p> <div style="border: 1px solid #f9e79f; padding: 10px; margin-top: 10px;"> <p>The <code>vlog_code_show_force</code> variable has been deprecated, and is currently maintained only for backwards compatibility with legacy code. For all supported releases, the Tcl <code>show_force</code> variable must be set to 1 at the time the forces are applied to enable the display of forces, whether from the Verilog code or from other sources.</p> </div> |
| <pre>vlog_format = value</pre> | <p>The <i>value</i> of this variable is the format for the output of Verilog values in describe output, stop point messages, and expression results. The value can be set to <code>%h</code>, <code>%x</code>, <code>%d</code>, <code>%o</code>, or <code>%b</code>.</p> <p>This variable is initially set to <code>%h</code>.</p> |
| <pre>- vhdl_forgen_loopindex_enum_pos</pre> | <p>This variable displays the enumeration position in place of the enumeration literal in the for-generate loop index in the output of the find TCL command.</p> <p>You can enable this functionality by running the following command on TCL:</p> <pre>set vhdl_forgen_loopindex_enum_pos 1</pre> <p>The following Tcl commands are affected when using this variable:</p> <ul style="list-style-type: none"> • <code>find</code>: Display enum position in place of enum literal value in the output • <code>scope -desc</code>, <code>scope -set</code>, <code>probe -screen</code>, <code>probe -create -shm</code>, and <code>stop -create -object</code>: Take enum position in place of enum literal in the input <p>For example:</p> <pre>type Example_T is (DUMMY1A, DUMMY2A, Link1, Link2, Link3, Link4, Link5, Link6, Link7, Link8,</pre> |

```
DUMMY1B, DUMMY2B);  
subtype SubTypeEx_T is Example_T range Link1 to  
Link8;  
  
BeginLabelEx:  
for i in SubTypeEx_T generate  
GenLabelEx:  
vhdl_ent  
port map (  
CLK(ConstantEx_C(i).Num-1), D(ConstantEx_C(i).Num-  
1), Q(ConstantEx_C(i).Num-1)  
);  
end generate;
```

TCL command:

```
find -instances * -recursive all
```

Current Output:

```
vhdl_top_inst:BeginLabelEx(Link1):GenLabelEx  
vhdl_top_inst:BeginLabelEx(Link2):GenLabelEx  
vhdl_top_inst:BeginLabelEx(Link3):GenLabelEx  
vhdl_top_inst:BeginLabelEx(Link4):GenLabelEx  
vhdl_top_inst:BeginLabelEx(Link5):GenLabelEx  
vhdl_top_inst:BeginLabelEx(Link6):GenLabelEx  
vhdl_top_inst:BeginLabelEx(Link7):GenLabelEx  
vhdl_top_inst:BeginLabelEx(Link8):GenLabelEx
```

Updated Output:

```
vhdl_top_inst:BeginLabelEx(2):GenLabelEx  
vhdl_top_inst:BeginLabelEx(3):GenLabelEx  
vhdl_top_inst:BeginLabelEx(4):GenLabelEx  
vhdl_top_inst:BeginLabelEx(5):GenLabelEx  
vhdl_top_inst:BeginLabelEx(6):GenLabelEx  
vhdl_top_inst:BeginLabelEx(7):GenLabelEx  
vhdl_top_inst:BeginLabelEx(8):GenLabelEx  
vhdl_top_inst:BeginLabelEx(9):GenLabelEx
```

Suppressing Assert Messages

When simulating a VHDL design, you may want to turn off assert messages that come from IEEE or user-defined packages, especially assert messages with a severity level of warning and note, and assert warning messages from built-in operators. You can do this using the Tcl `set` command to `set severity_pack_assert_off` and `pack_assert_off` variables.

1. Specify the severity level of the messages you want to suppress by setting the `severity_pack_assert_off` variable. The syntax is as follows:

```
set severity_pack_assert_off {severity_level}
```

The `severity_level` value can be one or more of the following:

- failure
- error
- warning
- note

If you specify more than one severity level, separate the levels with a space. For example:

```
xcelium> set severity_pack_assert_off {note}
xcelium> set severity_pack_assert_off {warning}
xcelium> set severity_pack_assert_off {error warning note}
```

The value of `severity_pack_assert_off` is initially set to `{warning note}`.

You cannot turn off different types of messages for different packages.

2. Specify the package name(s) from which you want to suppress the display of VHDL assert messages by setting the `pack_assert_off` variable. The syntax is as follows:

```
set pack_assert_off {package_specification}
```

The `package_specification` value is:

```
[library.]package [[library.]package ...]
```

If you specify more than one package, use a space (not a comma) to separate the package

names.

You can specify a library name if there are packages with the same name in different libraries and you want to turn off messages only in a package in a particular library. If you don't specify a library name, assert messages are turned off in all packages with the specified name.

Examples

```
xcelium> set pack_assert_off {numeric_std}
xcelium> set pack_assert_off {std_logic_arith numeric_std}
```

The simulator also has a predefined Tcl variable called `assert_1164_warnings`. This variable controls the display of warnings from built-in functions from the `std_logic_1164` package. This value of this variable is initially set to yes, which tells the simulator not to suppress these warnings. If you set the value to no, the simulator suppresses the warnings from this particular package.

Some packages include other packages. To turn off assertions that come from functions in an included package, you must specify the included package. To determine what package(s) the messages that you want to suppress come from, look at the actual assert messages generated in the `xmsim.log` file.

The following UNIX command is useful for determining which packages to filter:

```
% grep ieee xmsim.log | cut -d'@' -f2 | cut -d',' -f1 | sort | uniq
```

Examples

The following commands turn off assert warning messages from package `numeric_std`. The first command is optional if the `severity_pack_assert_off` variable is already set to warning.

```
xcelium> set severity_pack_assert_off {warning}
xcelium> set pack_assert_off {numeric_std}
```

The following commands turn off assert warning messages from packages `numeric_std` and `std_logic_arith`.

```
xcelium> set severity_pack_assert_off {warning}
xcelium> set pack_assert_off {numeric_std std_logic_arith}
```

The following commands turn off assert warning messages from packages `numeric_std`, `std_logic_arith`, and `mytypes`, which is in the library `mypack`.

```
xcelium> set severity_pack_assert_off {warning}
xcelium> set pack_assert_off {numeric_std std_logic_arith mypack.mtypes}
```

Xcelium Simulator Tcl Commands

The following simulator commands are available to help debug your design:

| | | | |
|----------------------------|------------------------------|-----------------------------|--------------------------|
| alias | find | omi | sn |
| analog | finish | pause | source |
| assertion | fmibkpt | power | stack |
| attribute | force | probe | status |
| call | heap | process | stop |
| check | help | profile | strobe |
| constraint | history | release | task |
| coverage | ida_database | reset | tcheck |
| database | ida_probe | restart | time |
| deposit | input | run | value |
| describe | xprof | save | verisium |
| drivers | logfile | scope | version |
| dumpsaif | loopvar | scverbosity | warn |
| dumptcf | maptypes | simtfile | where |
| exit | memory | simvision | xprop |

alias

The Tcl `alias` command defines handles that you can use as command short-cuts.

The following are common tasks that you can perform with this command:

- [Creating an alias](#)
- [Displaying the definition of an alias](#)

- [Removing an existing alias definition](#)

alias Command Syntax

```
alias [-set] [<alias_name> [<alias_definition>]]  
      -unset <alias_name>
```

 The `alias` command, with no modifiers or arguments, prints all alias definitions.

alias Command Options

This section describes the options that you can use with the Tcl `alias` command.

```
-set [<alias_name>  
     [<alias_definition>]]
```

Creates a command alias using a short-cut handle and definition. The `-set` modifier is optional.

```
-unset <alias_name>
```

Removes an alias definition. You must include the `alias_name` argument to specify the name of the alias that you want to remove. This is equivalent to the UNIX `unalias` command.

alias Command Examples

The following are examples of the various tasks you can perform with the `alias` command.

Creating an Alias

The following command creates an alias called `h`, using the `-set` option, and defines it as the `history` command.

```
xcelium> alias -set h history
```

Use the `h` alias to show the Tcl command history, as illustrated below.

```
xcelium> h  
1 alias -set h history  
2 alias bp stop -show  
3 alias go {run 10;value count}  
4 alias  
5 alias bp  
6 go  
7 h
```

The following command creates an alias called `bp` and defines it as the `stop -show` command. In this

case, the `-set` option is not specified.

```
xcelium> alias bp stop -show
```

The following command creates an alias called `go` and defines it as `{run 10;value count}` .

```
xcelium> alias go {run 10;value count}
```

Now, the `go` alias advances the simulation and shows the value of `count` .

```
xcelium> go
      5 count= x
4'hx
```

Be aware that Xcelium does not support creating an alias with the same name as a predefined Tcl command. For instance, the `puts` command is already a Tcl command so the following command will result in an error.

```
xcelium> alias puts value
xmsim: *E,ALNORP: cannot create an alias for Tcl command puts.
```

Displaying Alias Definitions

The following command prints the definition of the alias `bp` .

```
xcelium> alias bp
bp      stop -show
```

The following command prints all alias definitions.

```
xcelium> alias
bp      stop -show
go      run 10;value count
h       history
```

Removing an Alias

The following command deletes the alias `bp` .

```
xcelium> alias -unset bp
```

Creating an Alias

Use the `-set` option with the `alias` command to define a handle that you can use as a Tcl command short-cut.

Syntax

This section lists the command syntax to create an `alias`.

```
alias [-set] [alias_name [alias_definition]]
```

Options

| | |
|---|--|
| <code>-set [alias_name] [alias_definition]</code> | Creates a command alias using a short-cut handle and definition. The <code>-set</code> modifier is optional. |
|---|--|

Examples

The following command creates an alias called `h`, using the `-set` option, and defines it as the `history` command.

```
xcelium> alias -set h history
```

The following command creates an alias called `bp` and defines it as the `stop -show` command. In this case, the `-set` option is not specified.

```
xcelium> alias bp stop -show
```

The following command creates an alias called `go` and defines it as `{run 10;value count}`.

```
xcelium> alias go {run 10;value count}
```

Xcelium does not support creating an alias with the same name as a predefined Tcl command. For instance, the `puts` command is already a Tcl command so the following command results in an error.

```
xcelium> alias puts value  
xmsim: *E,ALNORP: cannot create an alias for Tcl command puts.
```

Displaying the Definition of an Alias

Use the `alias` command and include a particular alias name on the Tcl command line to print the definition of that alias.

Using this command with no modifiers or arguments prints all alias definitions.

Syntax

This section lists the command syntax to display the definition of an `alias`.

```
alias [alias_name]
```

Options

| | |
|----------------------------------|---|
| <code>[<i>alias_name</i>]</code> | Specifies the particular alias name for which you want to print a definition. |
|----------------------------------|---|

Examples

The following command prints the definition of the alias `bp`.

```
xcelium> alias bp  
bp      stop -show
```

The following command prints all alias definitions.

```
xcelium> alias  
bp      stop -show  
go      run 10;value count  
h       history
```

Removing an Existing Alias Definition

Use the `-unset` option with the `alias` command to remove an alias definition.

Syntax

This section lists the command syntax to remove an `alias` definition.

```
alias -unset alias_name
```

Options

`-unset <alias_name>`

Removes an alias definition. You must include the *alias_name* argument to specify the name of the alias that you want to remove. This is equivalent to the UNIX `unalias` command.

Examples

The following command deletes the alias `bp`.

```
xcelium> alias -unset bp
```

analog

The Tcl `analog` command controls the analog solver during a mixed-signal simulation.

analog Command Syntax

```
analog
  -iabstol <value>
  -off
  -on
  -reltol <value>
  -show
  -stop <time_spec>
  -tcl
  -vabstol <value>
```

analog Command Options

This section describes the options that you can use with the Tcl `analog` command.

| | |
|--------------------------------------|---|
| <code>-ahdllint_report</code> | Prints the dynamic linter report. |
| <code>-iabstol <value></code> | Changes the absolute tolerance for current to the specified <i>value</i> in the next analog time step. |
| <code>-off</code> | Skips the analog simulation starting from this digital timepoint. |
| <code>-on</code> | Starts the analog simulation starting from this digital timepoint. |
| <code>-reltol <value></code> | Changes the relative tolerance of the analog engine to the specified <i>value</i> in the next analog time step. |
| <code>-show</code> | Prints the tolerance values that will be set for the next analog time step using the <code>analog</code> command, if any. |
| <code>-stop <time_spec></code> | Changes the analog stop time to the specified <i>time_spec</i> . The value of the time specification is numeric and must be greater than the current analog time. |
| <code>-tcl</code> | Switches from Xcelium interactive Tcl to the Spectre FX interactive Tcl system. |
| <code>-vabstol <value></code> | Changes the absolute tolerance for voltage to the specified <i>value</i> in the next analog time step. |

Related Topic

- [Appendix B: Tcl-Based Debugging](#)

assertion

The Tcl `assertion` command enables you to control various PSL and SystemVerilog assertion-based verification features and VHDL `assert` messages.

The following are common tasks that you can perform with this command:

- [Disabling and enabling PSL/SVA and VHDL assertions](#)
- [Listing all the assertions in a given scope](#)
- [Controlling how Xcelium logs PSL/SVA and VHDL assertion output](#)
- [Controlling the level at which VHDL assertions cause the simulation to stop](#)
- [Printing a summary report of PSL/SVA assertion statistics](#)
- [Specifying one or more assertion directives and control each by setting fail/finish limits](#)

assertion Command Syntax

```
assertion
  [-psl | -vhdl | -sva]
  -asrctl on
  -counter <coverage_counter>
  -directive {<directive> | {directive_list} | none | all}}
    [-failure_limit <fail_limit>]
    [-finish_limit <finish_limit>]
    [-global_failure_limit <global_fail_limit>]
  -list <scope>
    [-depth {<levels> | all | to_cells}]
    [-directive <directive> | {directive_list} | none | all}}
    [-failed]
      [-new]
    [-multiline]
    [-nosort]
    [-permoff]
    [-signals <assertion_name>]
    [-uncovered]
  -logging [<scope_name>]
    [-all]
    [-append]
    [-cellname <vhdl_cellname>]
    [-depth {<levels> | all | to_cells}]
    [-error {on | off}]
    [-redirect <filename>]
    [-severity {note | warning | error | failure | never | global}]
```

```
        [-only {failure | error | warning | note }]
[-state {<state> | {state_list} | none | all}]
[-nostatechange {<state> | {state_list} | all}]
-off|-on [<scope_name>]
    [-all]
    [-always]
    [-cellname <vhdl_cellname>]
    [-concurrent]
    [-depth {<levels> | all | to_cells}]
    [-immediate]
    [-onfailure [<fail_limit>]]
-record coverall
-report opt
-resetcounter
-simstop
    [-all]
    [-cellname <vhdl_cellname>]
    [-depth {<levels> | all | to_cells}]
    -severity {note | warning | error | failure | never | global}
        [-only {failure | error | warning | note}]
-strict {on | off}
-style
    [-multiline | -oneline]
    [-statement | -unit]
-summary [instance_name]
    [-byfailure | -byname]
    [-extend_immediate]
    [-final]
    [-nosort]
    [-redirect <filename>]
    [-show {<counter> | {counter_list} | none | all}]
```

assertion Command Options

This section describes the options that you can use with the Tcl `assertion` command.

| | |
|---|--|
| <code>-psl</code> | Applies the <code>assertion</code> command to just the PSL assertions in the design. By default, an <code>assertion</code> command applies to PSL, SVA, and VHDL assertions when you disable assertions (<code>-off</code>), enable assertions (<code>-on</code>), or log assertions (<code>-logging</code>). |
| <code>-vhdl</code> | Applies the <code>assertion</code> command to just the VHDL assertions in the design. |
| <code>-sva</code> | Applies the <code>assertion</code> command to just the SystemVerilog assertions in the design. |
| <code>-asrctl on</code> | <p>Prints a note in the simulation log file with detailed hierarchical information about the assertion being turned on or off using the assert control commands. The information in the note is printed in the following format.</p> <pre>xmsim: *N, ASRCTL(<path to file with \$asserton/\$assertoff at this time>, <line#>): (time <timestamp>) assert_control_type:\$asserton(), scope_or_assertion:<path to assert control> Hierarchical_depth:all</pre> |
| <code>-counter</code> <code><coverage_counter></code> | Prints the value of the specified assertion statistics counter for the given assertion. For all assertions, the values of the following counters can be displayed: <code>Checked</code> , <code>Disabled</code> , <code>Finished</code> , and <code>Failed</code> . For SystemVerilog assertions, the following additional counter values can be displayed: <code>Pass</code> , <code>Vacuous</code> , and <code>Attempts</code> . |
| <code>-depth</code> <code>{<levels> </code> <code>all </code> <code>to_cells}</code> | <p>Specifies how many scope levels in the instance hierarchy to descend when searching for assertions. The <code>-depth</code> option applies to a specified scope. This option has no effect if a scope is not specified.</p> <p>You must specify one of the following arguments:</p> |

- **<levels>**: Descends the specified number of scopes. For example, `-depth 1` means include only the given scope, `-depth 2` means include the given scope and its subscopes, and so on. The default is 1.
- **all**: Includes all scopes in the hierarchy below the specified scope.
- **to_cells**: Includes all scopes in the hierarchy below the specified scope, but stops at cells (Verilog modules with `celldefine` or VITAL entities with `VITAL Level0` attribute).

This option is used when you want to disable (`-off`), enable (`-on`), list (`-list`), and log (`-logging`) assertions at specific scope levels of a hierarchy.

```
-directive
{<directive> |
{<directive_list>} |
none | all}}
```

Specifies the assertion directive for which the command applies.

This option applies to PSL and SVA assertions only.

Use the following options to control the assertion directive:

- **-failure_limit <fail_limit>**: Turns off the assert/assume directive after their `fail_limit` failures.
- **-finish_limit <finish_limit>**: Turns off the cover directive after their `finish_limit` finishes.
- **-global_failure_limit <global_fail_limit>**: Turns off all the assertions after their total `global_fail_limit` failures.

```
-list <scope>
```

Lists all the assertions in a given scope.

You can use the following options with assertion `-list`:

- **-depth {<levels> | all | to_cells}**: Specifies how many scope levels in the instance hierarchy to descend when searching for assertions. This option applies to the specified scope and has no effect if a scope is not specified.

- **-directive** {<directive> | {directive_list} | none | all}: Specifies the assertion directive for which the command applies.

This option applies to PSL and SVA assertions only. Currently, the supported directives are `assert`, `assume`, `cover`, and `restrict`. To specify more than one directive, you can use the option multiple times or specify a list containing directives. You can use `all` for all directives. To turn off applying the command to all directives, you can use an empty list or specify `none`.

- **-failed**: Lists all the failed assertions in a given scope.
- **-new**: Lists all the assertions failed from the last invocation of the `assertion -list -failed` command.
- **-multiline**: Shows each assertion in a separate line.
- **-nosort**: Does not sort assertions.
- **-permoff**: Lists all assertions which have been permanently turned off in the given scope.
- **-signals** <assertion_name>: Lists down an assertion with fan-ins.
- **-uncovered**: Lists down all the assertions which are uncovered at the time of access.

`-logging`
[<scope_name>]

Controls the logging of assertion output.

You can use the following options with `assertion -logging`:

- **-all**: Applies the `assertion` command to all assertions in the design. Use this option when you disable assertions (`-off`), enable assertions (`-on`), log assertions (`-logging`), or control the stop level of VHDL assertions (`-simstop`).
- **-cellname** <vhdl_cellname>: Applies the assertion command to all VHDL assertions in the complete hierarchy of the specified VHDL cell. This option applies to VHDL assertions only.

Use this option when you disable assertions (`-off`), enable assertions (`-on`), log assertions (`-logging`), or control the stop level of VHDL assertions (`-simstop`).

- **-depth** {<levels>|all|to_cells}: Specifies how many scope levels in the instance hierarchy to descend when searching for assertions. This option applies to the specified scope and has no effect if a scope is not specified.

- **-error {on|off}**: Controls whether assertion failures contribute to the error total when logging assertions with `-logging`. The default is `on`, which means that the global simulation error count is updated when an assertion fails. If you specify `-error off`, assertion failures do not contribute to the global error count. This is because the default severity for assertion failures is changed from Error to Note.
- **-redirect <filename>**: Redirects assertion output to a file with the specified name. This option is used with the `-logging` option. It disables the default logging to the standard output device and the simulation log file.

The `-redirect` option is not available for SVA immediate assertions.

- **-append**: Appends the results of another `assertion -logging` command to the file specified with `-redirect`. You can accumulate assertion output from multiple simulations by using this option.

The log file is not locked. Using `-append` for parallel simulations might result in overwritten output.

- **-severity {global|never|failure|error|warning|note}**: Sets the minimum severity level for reporting assertion messages. With `-logging` this option is not required; however, when specified it applies to VHDL assertions only. A VHDL assertion message is logged if the message is at or higher than the set severity level.

If you include the `-only` option, only the assertions at the specified level are logged.

- **-only {failure|error|warning|note}**: Specifies an explicit severity level at which to log assertions.
- **-state {<state>|{state_list}|none|all}**: Specifies which assertion state transition(s) to log. This option applies to PSL/SVA assertions only.

Xcelium can log assertions when they transition to one of the following states: `inactive`, `active`, `failed`, `finished`, `disabled`, `suspended`, `off`, or `flushed`. To specify more than one state transition, use the option multiple times or specify a list of state transition values. Use `-state all` to log all state transitions.

To turn off logging for any transition, you can use an empty list or `none`.

- **-nostatechange** {<state>|{state_list}|all}: Specifies which successive assertion state transitions to suppress.

You can suppress assertions state transitions to one of the following states: `inactive`, `active`, `failed`, `finished`, `disabled`, `suspended`, `off`, or `flushed`. To specify more than one state transition, use the option multiple times or specify a list containing state transition values. Alternatively, you can use `all` to suppress all state transitions. After suppressing a state transition, you cannot revert the transition.

`-off|-on`

Turns the selected assertions off or on.

You can use the following options with `assertion -off` and `assertion -on`:

- **-all**: Applies the assertion command to all assertions in the design. Use this option when you disable assertions (`-off`), enable assertions (`-on`), log assertions (`-logging`), or control the stop level of VHDL assertions (`-simstop`).
- **-always**: When specified with `-off`, it turns off the selected assertions permanently that are not permanently turned on. Such assertions are not evaluated and cannot be turned on later by any other assertion control.

When specified with `-on`, it turns on the selected assertions permanently that are not permanently turned off. Such assertions cannot be turned off by any other assertion control.

Note: A permanently turned on assertion can be suspended using the Low Power control assertion commands.

- **-cellname** <vhdl_cellname>: Applies the assertion command to all VHDL assertions in the complete hierarchy of the specified VHDL cell. This option applies to VHDL assertions only.

Use this option when you disable assertions (`-off`), enable assertions (`-on`), log assertions (`-logging`), or control the stop level of VHDL assertions (`-simstop`).

- **-concurrent**: Applies the `assertion` command to concurrent assertions in the design.
- **-depth** {<levels>|all|to_cells}: Specifies how many scope levels in the instance hierarchy to descend when searching for assertions. This option applies to the specified scope and has no effect if a scope is not specified.
- **-immediate**: Applies the `assertion` command to immediate assertions in the design.

The `-onfailure` option is explicit to `assertion -off` and applies to PSL and SVA assertions only:

- **`-onfailure [<fail_limit>]`**: Specifies the number of times that a PSL or SVA assertion can fail before it is disabled.

This option applies a failure count to each instance of an assertion separately, so an assertion in one place in the design could be disabled, while the same assertion in a different place in the design could still be enabled. A `reset` command resets the assertion failure count to zero. Without this option, the default behavior is to disable the assertions after the next failure.

If you save the simulation, the failure count is saved. After a restart, the number of times an assertion is allowed to fail before being disabled depends on how many times the assertion failed before the save, in addition to the `-onfailure` limit.

| | |
|-------------------------------|--|
| <code>-record coverall</code> | Controls the assertion recording mode. This option has one supported argument: <code>coverall</code> . Use it to enable recording of all finish counts for cover directives. |
| <code>-report opt</code> | Enables counter details for assertions. |
| <code>-resetcounter</code> | Resets all the assertions present in the design. Applicable in IXCOM mode only. |
| <code>-simstop</code> | <p>Controls stopping a simulation with VHDL assertions in the design.</p> <p>This option applies to VHDL assertions only.</p> <p>You can use the following options with assertion <code>-simstop</code>:</p> <ul style="list-style-type: none"> • <code>-all</code>: Applies the assertion command to all assertions in the design. Use this option when you disable assertions (<code>-off</code>), enable assertions (<code>-on</code>), log assertions (<code>-logging</code>), or control the stop level of VHDL assertions (<code>-simstop</code>). • <code>-cellname <vhdl_cellname></code>: Applies the assertion command to all VHDL assertions in the complete hierarchy of the specified VHDL cell. This option applies to VHDL assertions only. <p>Use this option when you disable assertions (<code>-off</code>), enable assertions (<code>-on</code>), log assertions (<code>-logging</code>), or control the stop level of VHDL assertions (<code>-simstop</code>).</p> <ul style="list-style-type: none"> • <code>-depth {<levels> all to_cells}</code>: Specifies how many scope levels in the instance hierarchy to descend when searching for assertions. This option applies to the specified scope and has no effect if a scope is not specified. |

- **-severity {global|never|failure|error|warning|note}**: Sets the minimum severity level for reporting assertion messages. With `-simstop`, this option is required. When specified, a VHDL assertion message stops the simulation and returns to the Tcl prompt (or exits if not in interactive mode) if the message is at or higher than the set severity level.

If you include the `-only` option, only the assertions at the specified level stop the simulation.

- **-only {failure|error|warning|note}**: Specifies an explicit severity level at which assertions stop the simulation. The simulation stops only if an assertion's severity matches the level specified with this option.

`-strict {on|off}`

Controls the simulation of SystemVerilog assertions.

The Xcelium simulator runs the pass statement of an SVA 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 this option to control simulation checking of SystemVerilog assertions. The default is `off`.

You must use this command before starting the simulation.

When this option is `on`, the simulation:

- Executes both the pass and fail action blocks when a concurrent assertion passes and fails at the same time, which overlapping assertions can do. The property is considered to have failed.
- Counts vacuous pass conditions (the left operand of an implication is not met).
- Runs action blocks on vacuous pass conditions.
- Counts simultaneous finish and fail conditions.

`-style`

Modifies the style of the output assertion log messages.

This option applies to PSL/SVA assertions only.

You can use the following options with assertion `-style`:

- **-oneline**: Prints log messages on a single line. This is the default format.
- **-multiline**: Prints log messages on multiple lines.
- **-statement**: Prints source information for the assertion statement directly. This is the default source information setting.

- **-unit**: Prints source information for the parent unit of the assertion, rather than for the assertion statement itself. This affects the assertion message itself, but not the sub-expression level debug information that is printed with some messages.

-summary

Prints a summary report of assertion statistics.

This option applies to PSL/SVA assertions only.

You can use the following options with **-summary**:

- **-byfailure**: Sorts assertions by the number of failures.
- **-byname**: Sorts assertions by hierarchical name. This is the default.
- **-extend_immediate**: Displays the full hierarchical name for labeled immediate assertions. By default, full names are not printed in the summary text for labeled immediate assertions.
- **-final**: Defers the summary report until the end of the simulation.
- **-nosort**: Does not sort assertions.
- **-redirect <filename>**: Prints the summary output to the specified file.
- **-show {<counter>|{counter_list}|none|all}**: 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.

Use `all` to print all counters. To omit all counters, use an empty list or specify `none`.

assertion Command Examples

Consider a VHDL design `test.vhd` which includes an assertion for `MYPROP`, which defines the following behavior:

```
-- pragma property MYPROP = always { LO_HI_X_SEQ };
-- pragma assert MYPROP;
```

When running the simulation, the behavior for `MYPROP` must always evaluate to true, otherwise the assertion will fire, generating an error, and will cause the simulation to stop.

```
% xrun -compile -v93 -assert -work cells ./CELLS.src/x_tri_v.vhd -top X_TRI
...
VITAL LEVEL 0 ENTITY Check Successful
WIRE DELAY BLOCK Check Successful
VITALBEHAVIOR PROCESS Check Successful (./CELLS.src/x_tri_v.vhd, 61)
VITAL ARCHITECTURE Check Successful

% xrun -elaborate -v93 -assert -access +rwc test.vhd -top test
...
xmclab: *W,ARCMRA: Elaborating the WORKLIB.TEST:BENCH, MRA (most recently analyzed)
architecture.
    Elaborating the design hierarchy:
    Top level design units:
        :test(bench):
    Building instance specific data structures.
    Design hierarchy summary:
        Instances  Unique
        Components:   5      2
        Processes:   11      2
        Signals:     14      9
        Assertions:   1      1

    Writing initial simulation snapshot: WORKLIB.TEST:BENCH

% xrun -R -s
...
Loading snapshot worklib.test:bench ..... Done
...
xcelium> run
--      { {not(to_x01(0)); to_x01(0)} ; {to_x01(0); (to_x01(0) = 'X')}} };
      |
xmsim: *E,ASRTST (./test.vhd,40): (time 0 FS) Assertion :MYPROP has failed (1 cycles,
starting 0 FS)
0 FS + 0 (Assertion output stop: :MYPROP = failed)
```

Disabling Assertions

Disabling assertions allows you to continue with the simulation which can be helpful in those cases where you need more information for debug.

To disable all PSL and SVA and VHDL assertions in the design:

```
xcelium> assertion -off -all
```

To disable all VHDL assertions only, add the `-vhdl` option:

```
xcelium> assertion -off -vhdl -all
```

To disable the assertion directive `assert`:

```
xcelium> assertion -off -directive assert  
xcelium> run
```

```
Test Started ... (gross tests only - run comparescan for timing checks!)
```

```
enabling output driver ...
```

```
enabling driver1 ...
```

```
  cycling through all 9 values of std_ulogic on input ...
```

```
disabling driver1 ...
```

```
  cycling through all 9 values of std_ulogic on input ...
```

```
enabling driver2 ...
```

```
  cycling through all 9 values of std_ulogic on input ...
```

```
disabling driver2 ...
```

```
  cycling through all 9 values of std_ulogic on input ...
```

```
enabling driver1 and driver2 ...
```

```
  cycling through all 9 values of std_ulogic on inputs ...
```

```
disabling output driver ...
```

```
Test Completed.
```

```
xmsim: *W,RNQUIE: Simulation is complete.
```

```
xcelium> exit
```

Enabling Assertions

Once the debug process is complete, you can enable previously disabled assertions.

To enable all PSL and SVA and VHDL assertions in the design:

```
xcelium> assertion -on -all
```

To enable all VHDL assertions only, add the `-vhdl` option:

```
xcelium> assertion -on -vhdl -all
```

To enable the assertion directive `assert`:

```
xcelium> assertion -on -directive assert
```

Controlling the Assertions Log

You can use the `describe -verbose` command to display the assertion status for a scope. For example, in the following sequence of commands, a `describe` command is issued after the snapshot is loaded. The report level for VHDL assertions is controlled by the `assert_report_level` variable, which is initially set to `note`. The stop level is controlled by the `assert_stop_level` variable, which is initially set to `error`.

An `assertion -logging -severity` command is then issued to set the report level for scope `:driver1` to `error`, and an `assertion -simstop -severity` command is issued to set the stop level for scope `:driver1` to `warning`. A `describe -verbose` command is then used to display the assertion status for the scope `:driver1`.

Here, the command `xmsim` simulates the design instead of `xrun`.

```
% xmsim -nocopyright -messages -tcl test:bench
Loading snapshot worklib.test:bench ..... Done
xcelium> describe -verbose :driver1
:driver1...direct instantiation of entity X_TRI(X_TRI_V)
      VHDL assertion properties
      -----
      Local settings:
          status.....enable
          Logging.....default
      Global settings:
          assert_report_level...note
          assert_stop_level....error
xcelium> assertion -logging -vhdl :driver1 -severity error
xcelium> assertion -vhdl -simstop -severity warning :driver1
xcelium> describe -verbose :driver1
:driver1...direct instantiation of entity X_TRI(X_TRI_V)
      VHDL assertion properties
      -----
      Local settings:
          status.....enable
          report_level.....error  (Overrides assert_report_level value)
          stop_level.....warning  (Overrides assert_stop_level value)
          Logging.....default
      Global settings:
          assert_report_level...note
          assert_stop_level....error
xcelium> exit
```

Disabling and Enabling PSL/SVA and VHDL Assertions

Use the `-off` option with the `assertion` command to turn off the selected assertions. Use the `-on` option to enable previously disabled assertions. These options disable and enable a specified assertion directive, all assertions within a scope or VHDL cell, or all assertions in the design. It disables assertion reporting, breakpoints, failures in the probe waveform, and incrementing the failure count for the specified property.

Additionally, you can specify the number of scope levels in the instance hierarchy to descend when searching for assertions using the `-depth` option.

i For low-power simulation, specifying a Tcl `assertion -off` or `assertion -on` command overrides `create_assertion_control` behavior. Alternatively, when declaring assertions in your HDL source, you can add the `CDNS_UPF_ASSERT_CAC_IGNORE` or `CDNS_UPF_ALWAYS_ON_ASSERT_CAC_IGNORE` SystemVerilog attribute. This is synonymous with `assertion -on always`.

Syntax

This section lists the `assertion` command syntax to turn off or turn on selected assertions.

```
assertion -off|-on [scope_name] [-psl | -vhdl | -sva] [-all]
               [-always] [-cellname <vhdl_cellname>] [-concurrent]
               [-depth {<levels> | all | to_cells}] [-immediate]
               [-onfailure [fail_limit]]
```

Options

`-off|-on`

Turns the selected assertions off or on.

You can use the following options with `assertion -off` and `assertion -on`:

- **-all**: Applies the assertion command to all assertions in the design. Use this option when you disable assertions (`-off`), enable assertions (`-on`), log assertions (`-logging`), or control the stop level of VHDL assertions (`-simstop`).

- **-always**: When specified with `-off`, it turns off the selected assertions permanently that are not permanently turned on. Such assertions are not evaluated and cannot be turned on later by any other assertion control.

When specified with `-on`, it turns on the selected assertions permanently that are not permanently turned off. Such assertions cannot be turned off by any other assertion control.

Note: A permanently turned on assertion can be suspended using the Low Power control assertion commands.

- **-cellname <vhdl_cellname>**: Applies the assertion command to all VHDL assertions in the complete hierarchy of the specified VHDL cell. This option applies to VHDL assertions only.

Use this option when you disable assertions (`-off`), enable assertions (`-on`), log assertions (`-logging`), or control the stop level of VHDL assertions (`-simstop`).

- **-concurrent**: Applies the `assertion` command to concurrent assertions in the design.
- **-depth {<levels>|all|to_cells}**: Specifies how many scope levels in the instance hierarchy to descend when searching for assertions. This option applies to the specified scope and has no effect if a scope is not specified.
- **-immediate**: Applies the `assertion` command to immediate assertions in the design.

The `-onfailure` option is explicit to `assertion -off` and applies to PSL and SVA assertions only:

- **-onfailure [<fail_limit>]**: Specifies the number of times that a PSL or SVA assertion can fail before it is disabled.

This option applies a failure count to each instance of an assertion separately, so an assertion in one place in the design could be disabled, while the same assertion in a different place in the design could still be enabled. A `reset` command resets the assertion failure count to zero. Without this option, the default behavior is to disable the assertions after the next failure.

If you save the simulation, the failure count is saved. After a restart, the number of times an assertion is allowed to fail before being disabled depends on how many times the assertion failed before the save, in addition to the `-onfailure` limit.

Examples

Disabling Assertions

To disable the assertion directive `assert`:

```
xcelium> assertion -off -directive assert
```

To disable a list of assertion directives, including `assert` and `assume`:

```
xcelium> assertion -off -directive {assert assume}
```

To disable assertions in scope `:driver1` and all subscopes:

```
xcelium> assertion -off :driver1 -depth all
```

To disable SVA assertions in scope `:driver1` after a set failure limit of 3:

```
xcelium> assertion -off :driver1 -sva -onfailure 3
```

To disable VHDL assertions in the VHDL cell `CELLS.X_TRI:X_TRI_V`:

```
xcelium> assertion -off -cellname CELLS.X_TRI:X_TRI_V
```

To disable all PSL and SVA and VHDL assertions in the design:

```
xcelium> assertion -off -all
```

To disable all VHDL assertions only, add the `-vhdl` option:

```
xcelium> assertion -off -vhdl -all
```

Enabling Previously Disabled Assertions

To enable VHDL assertions in the VHDL cell `CELLS.X_TRI:X_TRI_V`:

```
xcelium> assertion -on -cellname CELLS.X_TRI:X_TRI_V
```

To enable assertions in scope `:driver1` and its subscopes (explain about the depth option also):

```
xcelium> assertion -on :driver1 -depth 2
```

To enable all PSL and SVA and VHDL assertions in the design:

```
xcelium> assertion -on -all
```

To enable all VHDL assertions only, add the `-vhdl` option:

```
xcelium> assertion -on -vhdl -all
```


Listing all the Assertions in a Given Scope

Use the `-list` option with the `assertion` command to list all the assertions in a given scope. You can filter the generated list by assertion directive, failed assertions, and those assertions which have been permanently turned off. Additionally, you can specify the number of scope levels in the instance hierarchy to descend when searching for assertions using the `-depth` option.

Syntax

This section lists the `assertion` command syntax to list all the assertions in a given scope.

```
assertion -list <scope> [-depth {<levels> | all | to_cells}]  
                [-directive <directive> | {directive_list} | none | all]  
                [-failed] [-new] [-multiline] [-nosort] [-permoff]  
                [-signals <assertion_name>] [-uncovered]
```

Options

`-list <scope>`

Lists all the assertions in a given scope.

You can use the following options with `assertion -list`:

- **-depth {<levels> | all | to_cells}**: Specifies how many scope levels in the instance hierarchy to descend when searching for assertions. This option applies to the specified scope and has no effect if a scope is not specified.

- **-directive {<directive> | {directive_list} | none | all}**: Specifies the assertion directive for which the command applies.

This option applies to PSL and SVA assertions only. Currently, the supported directives are `assert`, `assume`, `cover`, and `restrict`. To specify more than one directive, you can use the option multiple times or specify a list containing directives. You can use `all` for all directives. To turn off applying the command to all directives, you can use an empty list or specify `none`.

- **-failed**: Lists all the failed assertions in a given scope.
- **-new**: Lists all the assertions failed from the last invocation of the `assertion -list -failed` command.
- **-multiline**: Shows each assertion in a separate line.
- **-nosort**: Does not sort assertions.
- **-permoff**: Lists all assertions which have been permanently turned off in the given scope.

- **-signals** *<assertion_name>*: Lists down an assertion with fan-ins.
- **-uncovered**: Lists down all the assertions which are uncovered at the time of access.

Examples

To list the assertions in the scope `bench` of the design `test.vhd`:

```
xcelium> assertion -list bench  
:MYPROP
```

Controlling the PSL/SVA and VHDL Output Logs

Use the `-logging` option with the `assertion` command to control the logging of assertion output. You can specify which state transitions are reported, how to redirect the output, and whether assertions contribute to the total error count of a running simulation. Additionally, you can specify the number of scope levels in the instance hierarchy to descend when searching for assertions using the `-depth` option and/or the style preferences for the messages using the `-style` option.

Syntax

This section lists the `assertion` command syntax to control the logging of output.

```
assertion -logging [scope_name] [-psl | -vhdl | -sva] [-all] [-append]
          [-cellname <vhdl_cellname>] [-depth {<levels> | all | to_cells}]
          [-error {on|off}] [-redirect <filename>]
          [-severity {note | warning | error | failure | never | global}]
          [-only {failure | error | warning | note}]
          [-state {<state> | <state_list> | none | all}]
          [-nostatechange {<state> | <state_list> | all}]
```

Options

`-logging`
`[<scope_name>]`

Controls the logging of assertion output.

You can use the following options with `assertion -logging`:

- **-all**: Applies the `assertion` command to all assertions in the design. Use this option when you disable assertions (`-off`), enable assertions (`-on`), log assertions (`-logging`), or control the stop level of VHDL assertions (`-simstop`).
- **-cellname <vhdl_cellname>**: Applies the assertion command to all VHDL assertions in the complete hierarchy of the specified VHDL cell. This option applies to VHDL assertions only.

Use this option when you disable assertions (`-off`), enable assertions (`-on`), log assertions (`-logging`), or control the stop level of VHDL assertions (`-simstop`).

- **-depth {<levels>|all|to_cells}**: Specifies how many scope levels in the instance hierarchy to descend when searching for assertions. This option applies to the specified scope and has no effect if a scope is not specified.

- **-error {on|off}**: Controls whether assertion failures contribute to the error total when logging assertions with `-logging`. The default is on, which means that the global simulation error count is updated when an assertion fails. If you specify `-error off`, assertion failures do not contribute to the global error count. This is because the default severity for assertion failures is changed from Error to Note.
- **-redirect <filename>**: Redirects assertion output to a file with the specified name. This option is used with the `-logging` option. It disables the default logging to the standard output device and the simulation log file.

The `-redirect` option is not available for SVA immediate assertions.

- **-append**: Appends the results of another `assertion -logging` command to the file specified with `-redirect`. You can accumulate assertion output from multiple simulations by using this option.

The log file is not locked. Using `-append` for parallel simulations might result in overwritten output.

- **-severity {global|never|failure|error|warning|note}**: Sets the minimum severity level for reporting assertion messages. With `-logging` this option is not required; however, when specified it applies to VHDL assertions only. A VHDL assertion message is logged if the message is at or higher than the set severity level.

If you include the `-only` option, only the assertions at the specified level are logged.

- **-only {failure|error|warning|note}**: Specifies an explicit severity level at which to log assertions.
- **-state {<state>|{state_list}|none|all}**: Specifies which assertion state transition(s) to log. This option applies to PSL/SVA assertions only.

Xcelium can log assertions when they transition to one of the following states: `inactive`, `active`, `failed`, `finished`, `disabled`, `suspended`, `off`, or `flushed`. To specify more than one state transition, use the option multiple times or specify a list of state transition values. Use `-state all` to log all state transitions.

To turn off logging for any transition, you can use an empty list or `none`.

- **-nostatechange** {<state>|{state_list}|all}: Specifies which successive assertion state transitions to suppress.

You can suppress assertions state transitions to one of the following states: *inactive*, *active*, *failed*, *finished*, *disabled*, *suspended*, *off*, or *flushed*. To specify more than one state transition, use the option multiple times or specify a list containing state transition values. Alternatively, you can use *all* to suppress all state transitions. After suppressing a state transition, you cannot revert the transition.

Examples

To specify that all assertions should be logged in the file `assert.log`:

```
xcelium> assertion -logging -all -redirect assert.log
```

To log output for PSL/SVA assertions when they transition to the *finished* or *failed* state:

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

To log output of VHDL assertions in the VHDL cell `CELLS.X_TRI:X_TRI_V`.

```
xcelium> assertion -logging -cellname CELLS.X_TRI:X_TRI_V
```

Controlling the Level at which VHDL Assertions Stop the Simulation

Use the `-simstop` option with the `assertion` command to control stopping a simulation if the design includes VHDL assertions.

With this option, you can specify both the minimum severity level and the VHDL assert statements which will cause the simulation to stop and return to the Tcl prompt (or exit if not in interactive mode). To specify the severity level, you must use the `-severity` option. Additionally, you can specify the number of scope levels in the instance hierarchy to descend when searching for assertions using the `-depth` option.

Syntax

This section lists the `assertion` command syntax to control stopping a simulation with VHDL assertions.

```
assertion -vhdl -simstop [-all] [-cellname <vhdl_cellname>]
           [-depth {<levels> | all | to_cells}]
           -severity {note | warning | error | failure | never | global}
           [-only {note | warning | error}]
```

Options

`-simstop`

Controls stopping a simulation with VHDL assertions in the design.

This option applies to VHDL assertions only.

You can use the following options with `assertion -simstop`:

- **-all**: Applies the assertion command to all assertions in the design. Use this option when you disable assertions (`-off`), enable assertions (`-on`), log assertions (`-logging`), or control the stop level of VHDL assertions (`-simstop`).
- **-cellname <vhdl_cellname>**: Applies the assertion command to all VHDL assertions in the complete hierarchy of the specified VHDL cell. This option applies to VHDL assertions only.

Use this option when you disable assertions (`-off`), enable assertions (`-on`), log assertions (`-logging`), or control the stop level of VHDL assertions (`-simstop`).

- **-depth {<levels>|all|to_cells}**: Specifies how many scope levels in the instance hierarchy to descend when searching for assertions. This option applies to the specified scope and has no effect if a scope is not specified.

- **-severity {global|never|failure|error|warning|note}**: Sets the minimum severity level for reporting assertion messages. With **-simstop**, this option is required. When specified, a VHDL assertion message stops the simulation and returns to the Tcl prompt (or exits if not in interactive mode) if the message is at or higher than the set severity level.

If you include the **-only** option, only the assertions at the specified level stop the simulation.

- **-only {failure|error|warning|note}**: Specifies an explicit severity level at which assertions stop the simulation. The simulation stops only if an assertion's severity matches the level specified with this option.

Examples

To stop the simulation at the set severity level of `warning` for VHDL assertions in scope `:driver1`:

```
xcelium> assertion -vhdl -simstop -severity warning :driver1 -depth all
```

The parameters **-simstop -severity warning** set a localized stop level for the specified scope. This overrides any global stop level set with the `assert_stop_level` variable. To return control of the stop level to the `assert_stop_level` variable, use **-severity global**:

```
xcelium> assertion -vhdl -simstop -severity global :driver1 -depth all
```

The stop level for VHDL assertions in scope `:drivers1` is now controlled by the value of the `assert_stop_level` variable, rather than by the severity level set with the **-severity** option in the previous command.

Printing a Summary of PSL/SVA Assertion Statistics

Use the `-summary` option with the `assertion` command to print a summary report of assertion statistics. The summary report is a plain text listing of the information displayed in the Assertion Browser: the assertions in your design and their checked, finished, and failed counts.

Optionally, you can choose to start the summary at a specified instance name so that only those assertions in that module and below are included in the report.

Syntax

This section lists the `assertion` command syntax to print a summary of assertion statistics.

```
assertion -summary [instance_name] [-psl | -sva] [-byfailure | -byname]
          [-extend_immediate] [-final] [-nosort] [-redirect <filename>]
          [-show {<counter> | {counter_list} | none | all}]
```

Options

`-summary`

Prints a summary report of assertion statistics.

This option applies to PSL/SVA assertions only.

You can use the following options with `-summary`:

- **-byfailure**: Sorts assertions by the number of failures.
- **-byname**: Sorts assertions by hierarchical name. This is the default.
- **-extend_immediate**: Displays the full hierarchical name for labeled immediate assertions. By default, full names are not printed in the summary text for labeled immediate assertions.
- **-final**: Defers the summary report until the end of the simulation.
- **-nosort**: Does not sort assertions.
- **-redirect <filename>**: Prints the summary output to the specified file.

- **-show {<counter>|{counter_list}|none|all}**: 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.

Use `all` to print all counters. To omit all counters, use an empty list or specify `none`.

Examples

To print a summary report of assertion statistics:

```
xcelium> assertion -summary -byfailure -final -redirect assert.log
```

In the report, assertions are sorted by the number of failures. The `-final` option specifies that the report is generated at the end of simulation.

Specifying an Assertion Directive and Setting the Fail/Finish Limits

Use the `-directive` option with the `assertion` command to specify the assertion directive for which the command applies.

In this Xcelium release, the supported directives are `assert`, `assume`, `cover`, and `restrict`. To specify more than one directive, you can use the option multiple times or specify a list containing directives. You can use `all` for all directives. To turn off applying the command to all directives, you can use an empty list or specify `none`.

Syntax

This section lists the `assertion` command syntax to specify an assertion directive.

```
assertion -directive {<directive> | {<directive_list>} | none | all}
           [-failure_limit <fail_limit>] [-finish_limit <finish_limit>]
           [-global_failure_limit <global_fail_limit>]
```

Options

```
-directive
{<directive> |
{<directive_list>} |
none | all}}
```

Specifies the assertion directive for which the command applies.

This option applies to PSL and SVA assertions only.

Use the following options to control the assertion directive:

- **-failure_limit <fail_limit>**: Turns off the `assert/assume` directive after their *fail_limit* failures.
- **-finish_limit <finish_limit>**: Turns off the `cover` directive after their *finish_limit* finishes.
- **-global_failure_limit <global_fail_limit>**: Turns off all the assertions after their total *global_fail_limit* failures.

Examples

To disable the assertion directive `assert`:

```
xcelium> assertion -off -directive assert
```

To disable the assertion directive `assert` after a set limit of 3 failures:

```
xcelium> assertion -off -directive assert -failure_limit 3
```

To disable a list of assertion directives, including `assert` and `assume`:

```
xcelium> assertion -off -directive {assert assume}
```

To disable all assertions after a set global limit of 3 failures:

```
xcelium> assertion -off -all -global_failure_limit 3
```

attribute

The Tcl `attribute` command enables VHDL function signal attributes for certain signals by default so that they can then be accessed from the Tcl interface with the `value` command. Xcelium supports the following function signal attributes when using this command:

- ``EVENT`
- ``LAST_EVENT`
- ``ACTIVE`
- ``LAST_ACTIVE`
- ``LAST_VALUE`

When you use this command to enable attributes on a signal, all the five supported function signal attributes are enabled on that signal automatically. This command cannot enable signal kind attributes, such as ``DELAYED`, ``STABLE`, ``QUIET`, and ``TRANSACTION`. These signal attributes are enabled only if they exist in the design.

Once you have enabled the function signal attributes for a signal, you cannot disable them.

For a complete description of these attributes, see the latest VHDL LRM.

Optionally, you can also use this command to create user-defined attributes. User-defined attributes use VHDL semantics to map a specified value-bearing object to a design object that holds the attribute.

attribute Command Syntax

```
attribute <signal_name> [<signal_name> ...]  
    -create  
        -attr <attribute_name>  
        -value <path_to_value_object> <design_object>  
    -show  
        -all  
        -attr <attribute_name> <design_object>
```

attribute Command Options

This section describes the options that you can use with the Tcl `attribute` command.

| | |
|----------------------------------|---|
| <code><signal_name></code> | Specifies the name of a signal to which you want to enable attributes. This command supports both single and multiple name values. |
|----------------------------------|---|

| | |
|----------------------|---|
| <code>-create</code> | <p>Creates a user-defined attribute. Once created (registered), the attribute becomes available to the Tcl <code>probe</code> and <code>value</code> commands.</p> <p>The following sub-options are required when creating a user-defined attribute:</p> <ul style="list-style-type: none">• <code>-attr <attribute_name></code>: Specifies the name of the attribute.• <code>-value <path_to_value_object> <design_object></code>: Specifies the path of a value-bearing object that is associated with a design object. The design object holds the attribute. |
| <code>-show</code> | <p>Displays registered user-defined attributes.</p> <p>You can control the output by using one of the following sub-options:</p> <ul style="list-style-type: none">• <code>-all</code>: Specifies that all registered user-defined attributes are shown.• <code>-attr <attribute_name> <design_object></code>: Specifies that only the explicitly named user-defined attribute is shown. |

attribute Command Examples

The following are examples of some of the tasks you can perform with the `attribute` command.

Enabling Function Signal Attributes

The following VHDL source code is used for the examples in this section.

```
library ieee;
use ieee.std_logic_1164.all;

entity d_flop is
    generic (setup_time, hold_time : time );
    port (d, clk : in std_logic;
          q : out std_logic);

begin
    setup_check : process (clk)
    begin
        if (clk = '1') and (clk'event) then
            assert (d'last_event <= setup_time)
            report "setup violation"
            severity error;
        end if;
    end process setup_check;
```

```
hold_check : process (clk'delayed(hold_time))
begin
    if (clk'delayed(hold_time) = '1') and
        (clk'delayed(hold_time)'event) then
        assert (d'last_event = 0 ns) or
            (d'last_event < hold_time)
        report "hold violation"
        severity error;
    end if;
end process hold_check;
end d_flop;

architecture d_flop_behave of d_flop is
begin
    dff_process : process (clk)
    begin
        if (clk = '1') and (clk'event) then
            q <= d;
        end if;
    end process dff_process;
end d_flop_behave;

% xmvhdl -nocopyright d_flop.vhd
% xmelab -nocopyright -access +rwc -generic "setup_time => 10 fs"
    -generic "hold_time => 10 fs" WORKLIB.D_FLOP:D_FLOP_BEHAVE
% xmsim -nocopyright -tcl -messages WORKLIB.D_FLOP:D_FLOP_BEHAVE
Loading snapshot worklib.d_flop:d_flop_behave ..... Done

xcelium> run 400
Ran until 400 FS + 0
```

In the code example shown above, `clk'event` and `d'last_event` are used in the design. Therefore, all function signal attributes are automatically enabled for the signals `clk` and `d`.

```
xcelium> value clk'event
FALSE
xcelium> value clk'last_event
400 FS
xcelium> value clk'last_value
'U'
xcelium> value d'last_event
400 FS
xcelium> value d'last_value
'U'
```

In the code example, no attributes are used on the `q` signal. Use the `attribute` command to enable function signal attributes for this signal. The attributes assume the default value that they would have had

at the start of the simulation. The accurate value of the attributes is only available after the simulation has been advanced and the values have been updated.

```
xcelium> attribute q
xcelium> run 400
Ran until 800 FS + 0
xcelium> value q'event
FALSE
xcelium> value q'last_active
800 FS
```

Use `-linedebug` option with *xmvhdl*.

Signal-valued attributes (``DELAYED`, ``STABLE`, ``QUIET`, ``TRANSACTION`) are enabled only if you have used them in the design. You cannot enable these attributes by using the `attribute` command.

```
xcelium> attribute q
xcelium> value q'quiet
xmsim: *E,BASGAT: Attribute not enabled for this signal.
```

Using Attributes for Subelements of a Vector

Function signal attributes are enabled for all subelements of a vector in Xcelium because of an internal optimization for vector signals. For instance, consider the following signal declared in some VHDL code:

```
signal x1 : std_logic_vector (0 to 1);
```

The Tcl `attribute` command below enables the attributes for both bits of the vector, even though one subelement is specified.

```
xcelium> attribute :x1(0)
```

If you must enable the attributes only for specific subelements of the vector, elaborate the design with the `-expand` command-line option. The simulator generates an error if you then use the `value` command on a subelement for which you have not enabled the [VHDL function signal attributes](#). For example:

```
xcelium> attribute :x1(0)
xcelium> value :x1(0)'event
FALSE
xcelium> value :x1(1)'event
xmsim: *E,NONFVA: function valued attributes are not enabled for this prefix.
```

If you issue the Tcl `attribute` command after the simulation has begun, the attributes assume the default values that they would have had at the start of the simulation. The attribute values are accurate only after the values are updated during the simulation.

Related Topic

- [Accessing Signal Attributes Using Tcl Commands](#)

call

The Tcl `call` command invokes a user-defined C-interface function, or a Verilog user-defined PLI system task or function, or a SystemC function from the command line.

call Command Syntax

```
call [-systf | -predefined] task_or_function_name [arg1 [arg2 ...]]
```

call Command Options

This section describes the options that you can use with the Tcl `call` command.

`-systf`
`[task_or_function_name]`

Looks for the specified task or function name only in the table of user-defined PLI system tasks and functions.

This option is available because the `call` command is also used to invoke functions from the VHDL C-interface, and there may be a user-defined C-interface function with the same name as a PLI system task or function. The `-systf` option causes the lookup in the C-interface task list to be skipped.

This option must appear before the task or function name on the command line.

You cannot use this option with the `-predefined` option.

The command `call -systf` with no task or function, the name argument displays a list of all registered user-defined system tasks and functions.

`-predefined`
`[function_name]`

Looks for the specified task or function name only in the table of predefined CFC library functions.

You cannot use the `-predefined` option when calling a user-defined system task or function.

This option must appear before the CFC function name on the command line.

You cannot use this option with the `-systf` option.

The command `call -predefined` with no function, the name argument displays a list of all predefined C function names.

call Command Arguments

Arguments to the system task or function can be either literals or names.

Literals can be:

- **Integers**

```
xcelium> call mytask 5
xcelium> call mytask 5 7
```

- **Reals**

```
xcelium> call mytask 3.4
xcelium> call mytask 22.928E+10
```

- **Strings**

Strings must be enclosed in double quotes. Enclose strings in curly braces or use the backslash character to escape quotes, spaces, and other characters that have special meaning to Tcl. For example:

```
xcelium> call mytask {"hello world"}
xcelium> call mytask \"hello\\ world\\\"
```

- **Verilog literals, such as 8'h1f**

Names can be full or relative path names of instances or objects. Relative pathnames are relative to the current debug scope (set by the `scope` command). Object names can include a bit select or part select. For example:

```
xcelium> call mytask top.u1
xcelium> call mytask top.u1.reg[3:5]
```

Expressions that include operators or function calls are not allowed. For example, the following two commands result in an error:

```
xcelium> call \\$mytask a+b
xcelium> call \\$mytask {func a}
```

However, literals can be created using Tcl's `expr` command. For example, if the desired argument is the expression `(a+b)`, use the following:

```
xcelium> call \\$mytask [expr #a + #b]
```

The result of the expression `(a+b)` is substituted on the command line and then treated by the `call` command as a literal.

The `expr` command cannot evaluate calls to Verilog functions.

If you are calling a user-defined system function, the result of the `call` command is the return value from the system function. Therefore, user-defined system functions can be used to generate literals for other commands. For example:

```
xcelium> call task [call func arg1 ...]
xcelium> force a = [call func arg1 ...]
```

call Command Examples

The following Verilog module contains a call to a user-defined system task and to a system function. The task and function can also be invoked from the command line.

```
module test();
initial
begin
    $hello_task();
    $hello_task($hello_func());
end
endmodule
```

The following command invokes the `$hello_task` system task:

```
xcelium> call \ $hello_task
```

This task can also be invoked with any of the following:

```
xcelium> call hello_task
xcelium> call {$hello_task}
xcelium> call {hello_task}
```

The `$hello_func` function can be invoked with any of the following commands:

```
xcelium> call \ $hello_func
xcelium> call hello_func
xcelium> call {$hello_func}
xcelium> call {hello_func}
```

In the following command, the `call` command calls the `$hello_task` system task with a call to the system function `$hello_func` as an argument.

```
xcelium> call hello_task [call hello_func]
```

The following command displays a list of all registered user-defined system tasks and functions.

```
xcelium> call -sysf
```

Related Topic

- [call and assert Command Compositions](#)

call and assert Command Compositions

When using the `assert` command in the `call` command, the command compositions of the `call` command are:

```
xcelium> call [assert_cmd("level","scope_inst")]
```

The `assert` command string has the following three compositions:

- **`assert_cmd`:** Represents a superset of strings of the supported assert system/task, namely, `$asserton`, `$assertoff`, and `$assertkill`.
For example:

```
xcelium> call \"$asserton(3,\"instance.hierarchy.here\")\"  
xcelium> call {$asserton(3,\"instance.hierarchy.here\")}
```
- **`level`:** Represents the depth of assert system/task with respect to the scope of the given scope instance/`assert_name`, 0 denotes all depths. `Level` is a number allowed only when presented by decimal digit(s).
- **`scope_inst`:** The `scope_inst` can be module identifier, assertion identifier, or hierarchical identifier.

Currently, the `scope_inst` can only support the usage with single scope, where only one module identifier, or one assertion identifier, or one hierarchical identifier is allowed. For example:

```
xcelium> call \"$asserton(3,\"instance.hierarchy.here\")\"
```

check

The Tcl `check` command checks for bus contention and bus float conditions for specified VHDL bus signals (a signal that has multiple drivers). Checks can be applied only on VHDL objects and you can use this command only on `std_logic` bus signals.

The simulator issues a warning message if you attempt to use the `check` command on:

- Verilog objects
- Signals that are declared in a VITAL Level0 scope

Checks are performed if any of the drivers of the bus signal change and at the end of the simulation. If the simulator detects a bus contention or bus float, it issues an error message with the following information:

- For bus contention, the message includes the name of the bus signal on which the contention was detected, the names of the drivers and their current values, the time at which the contention started, and the time at which the time window was exceeded.
- For bus float detection, the message includes the bus signal name, the time at which the float started, and the time at which the time window was exceeded.

check Command Syntax

check

```
{-contention|-float <signal_specifier>
  [-delay <check_time_limit>]
  [-name <check_name>] |
[-delay <time_limit>]}
-delete <check_name> [<check_name>...]
-disable <check_name> [<check_name>...]
-enable <check_name> [<check_name>...]
-show [<check_name>...]
```

check Command Options

This section describes the options that you can use with the Tcl `check` command.

`-contention|-float`
`<signal_specifier>`

Specifies bus contention detection or bus float detection.

The `signal_specifier` argument can be:

- An explicit signal name or a list of signal names, using spaces to separate each name value.
- **-all**: Specifies checking on all bus signals in the current scope.
- **-depth all | n <scope_name>**
 - **-depth all**: Specifies checking on all bus signals in the specified scope and in all scopes in the hierarchy below the specified scope.
 - **-depth n**: Specifies checking on all bus signals in the specified scope and in the specified number of sub-scopes. For example, **-depth 1** means only the specified scope, **-depth 2** means the specified scope and its sub-scopes, and so on. The default is 1.

The following options are also supported when specifying `-contention` or `-float` using the Tcl `check` command:

- **-delay <check_time_limit>**: Specifies a time limit for the check. If you do not specify a time limit for a `check -contention` or `check -float` command, the value is set using a previous `check -delay` command. If you have not specified a time limit using a `check -delay` command, the default value is 1 fs.
- **-name <check_name>**: Specifies a name for the check. You can then use the name that you assign to the check with the `-disable`, `-enable`, `-delete`, and `-show` modifiers. By default, checks are numbered sequentially.

| | |
|--|---|
| <code>-delay <time_limit></code> | <p>Specifies a default time limit for bus contention or bus float checks.</p> <p>The default bus contention or bus float time limit cannot be smaller than, or more precise than, the unit of simulation.</p> <p>If you do not specify a time limit in the <code>check -contention</code> or <code>check -float</code> command, the time value is the value set with a previous <code>check -delay</code> command. If you have not specified a time limit using a <code>check -delay</code> command, the default value is 1 fs.</p> <p>Changing the default time limit using a <code>check -delay</code> command does not affect existing checks.</p> |
| <code>-delete <check_name></code> <code>[<check_name>...]</code> | <p>Deletes the check that has the specified name. If you specify more than one check, separate the names with a space.</p> <p>You can use wildcard characters (* or ?) in a <code>check -delete</code> command.</p> |
| <code>-disable <check_name></code> <code>[<check_name>...]</code> | <p>Disables the check that has the specified name. If you specify more than one check, separate the names with a space.</p> <p>You can use wildcard characters (* or ?) in a <code>check -disable</code> command.</p> <p>To resume checking, use the <code>-enable</code> modifier.</p> |
| <code>-enable <check_name></code> <code>[<check_name>...]</code> | <p>Enables a previously disabled check so that checking is resumed. If you specify more than one check, separate the names with a space.</p> <p>You can use wildcard characters (* or ?) in a <code>check -enable</code> command.</p> |
| <code>-show [<check_name>...]</code> | <p>Displays information about the check that has the specified name. If you specify more than one check, separate the names with a space.</p> <p>If you do not include a <code>check_name</code> argument, the <code>check -show</code> command displays information on all checks.</p> |

check Command Examples

The following VHDL source file is used for the examples in this section:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
use std.textio.all;
entity drvio is
    port ( dr : inout std_logic_vector(1 to 3));
end drvio;
architecture drv_arch of drvio is
    signal a,b,c : std_logic := 'Z';
begin
    dr(2) <= b;
    dr(2) <= c;
```

```
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity drvo is
    port ( dr : out std_logic_vector(1 to 3));
end drvo;
architecture drv_arch of drvo is
    signal c : std_logic := 'Z';
begin
    dr(2) <= c, 'H' after 5 ns ;
    dr(2) <= c, 'L' after 5 ns;
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.vital_timing.all;
use ieee.vital_primitives.all;
use ieee.std_logic_textio.all;
use std.textio.all;
entity trace is
end trace;
architecture arch of trace is
    component drvio
        port( dr : inout std_logic_vector(1 to 3));
    end Component;
    component drvo
        port( dr : out std_logic_vector(1 to 3));
    end Component;
    for all: drvio use entity work.drvio(drv_arch);
    for all: drvo use entity work.drvo(drv_arch);

    signal a, b, c, d, k : std_logic := 'Z';
begin
    a <= d ;
    vitalident(a,b);
    U1 : drvo port map (dr(2) => a);
    U2 : drvio port map (dr(2)=>a);
    a <= 'L' , 'Z' after 2 ns;
    b <= '1' after 4 ns ;
    d <= 'H' after 1 ns, 'Z' after 2.5 ns, '0' after 4 ns;
    k <= b;
    k <= c;
    process
        begin
            wait for 20 ns;
```

```
        assert false severity failure;
    end process;
end;
```

In the following sequence of commands:

- The `check -delay` command sets a default time limit of 10 fs for bus contention and bus float checks.
- The next command creates a bus contention check on bus signal `:a`. If more than one driver of `a` is driving a non-Z value for more than 10 fs, the bus contention messages is displayed.
- The third command creates a bus float check on bus signal `:k`. The `-delay` option is included to set a time limit of 20 fs for this check. If no driver is driving `k` for more than 20 fs, a bus float message is displayed.

```
% xmvhdl -nocopyright test.vhd
% xmelab -nocopyright -access rwc worklib.trace:arch
% xmsim -nocopyright -tcl -messages worklib.trace:arch
Loading snapshot worklib.trace:arch ..... Done
xcelium> check -delay 10 fs
xcelium> check -contention :a
Created check 1
xcelium> check -float :k -delay 20 fs
Created check 2
xcelium> run
2 NS: Contention detected on signal :a(test.vhd,45) at 1000010 FS from 1 NS between the
following drivers -
value --> 'L'  (:)a <= 'L' , 'Z' after 2 ns [File: test.vhd, Line: 51]
value --> 'H'  (:)a <= d [File: test.vhd, Line: 47]
4 NS: Float detected on signal :k(test.vhd,45) at 20 FS from 0 FS
5 NS: Contention detected on signal :a(test.vhd,45) at 4000010 FS from 4 NS between the
following drivers -
value --> '1'  (:)vitalident(a,b) [File: test.vhd, Line: 48]
value --> '0'  (:)a <= d [File: test.vhd, Line: 47]
ASSERT/FAILURE (time 20 NS) from process :$PROCESS_007 (architecture WORKLIB.trace:arch)
Assertion violation.
Assertion at 20 NS + 0
./test.vhd:59          assert false severity failure;
```

The following reset command reloads the snapshot. The `check -show` command shows that the checks are still enabled.

```
xcelium> reset
Loaded snapshot worklib.trace:arch
xcelium> check -show
1          Enabled          10 FS          -contention :a

2          Enabled          20 FS          -float :k
```

The following command deletes all checks:

```
xcelium> check -delete *
Deleted Check 1
Deleted Check 2
xcelium> check -show
No checks set
```

The following command specifies bus contention detection on bus signals `:a` and `:u1:dr(2)`. The `-name` option is included to specify a user-defined name for the check.

```
xcelium> check -contention :a :u1:dr(2) -name contention_check
Created check contention_check
```

The following command specifies bus float detection on all bus signals in the current scope and its sub-scopes. The `-name` option is included to specify a user-defined name for the check.

```
xcelium> check -float -depth 2 : -name float_check
xmsim: *W,CHSBPR: check can not be applied on :U1:DR(1).
xmsim: *W,CHSBPR: check can not be applied on :U1:DR(3).
xmsim: *W,CHSBPR: check can not be applied on :U2:DR(1).
xmsim: *W,CHSBPR: check can not be applied on :U2:DR(3).
Created check float_check
```

- The following command displays information on the check named `contention_check`.

```
xcelium> check -show contention_check
contention_check      Enabled          10 FS      -contention :a
                      :U1:dr(2)
```

- The following command disables the check named `contention_check`.

```
xcelium> check -disable contention_check
Disabled Check contention_check
xcelium> check -show
float_check      Enabled          10 FS      -float -depth 2 :
contention_check Disabled          10 FS      -contention :a
                      :U1:dr(2)

xcelium> run
4 NS: Float detected on signal :k(test.vhd,45) at 10 FS from 0 FS
4 NS: Float detected on signal :a(test.vhd,45) at 2500010 FS from 2500 PS
5 NS: Float detected on signal :U1:dr(2)(test.vhd,18) at 10 FS from 0 FS
ASSERT/FAILURE (time 20 NS) from process :$PROCESS_007 (architecture WORKLIB.trace:arch)
Assertion violation.
Assertion at 20 NS + 0
./test.vhd:59      assert false severity failure;
xcelium> exit
20 NS: Float detected on signal :U2:dr(2)(test.vhd,6) at 10 FS from 0 FS
```


Checking for Bus Contention and Bus Float Conditions

The simulator lets you check for bus contention and bus float conditions in your design. You can perform these two design checks only on VHDL `std_logic` bus signals (a signal that has multiple drivers). Checks cannot be applied on signals that are declared in a VITAL Level0 scope.

- **Bus contention detection:** Bus contention checking detects bus fights on nodes that have multiple drivers. If more than one driver of a `std_logic` bus signal drives a non-Z value for more than a specified time limit, the condition is reported as a bus contention.

If the simulator detects a bus contention, it issues an error message. The message includes the name of the bus signal on which the contention was detected, the names of the drivers and their current values, the time at which the contention started, and the time at which the time window was exceeded.

- **Bus float detection:** Bus float checking detects nodes that are in the high impedance state for a time that exceeds a specified time limit. If no driver of a `std_logic` bus signal is driving a non-Z value for more than a specified time limit, the condition is reported as a bus float.

If the simulator detects a bus float, it issues an error message. The message includes the bus signal name, the time at which the float started, and the time at which the time window was exceeded.

Use the Tcl `check` command to check for bus contention/bus float conditions on a specified VHDL bus signal(s). Use the `-contention` option to create a bus contention check, or use the `-float` option to create a bus float check.

The following example shows how to use the `check` command to create a bus contention check. If more than one driver of bus signal `sig1` is driving a non-Z value and if `sig1` is active for more than 10 ns, the simulator reports a bus contention message. If the duration of the bus contention is 10 ns or less, the condition is not reported.

```
xcelium> check -contention sig1 -delay 10 ns
```

The following example shows how to use the `check` command to create a bus float check. If no driver is driving bus signal `sig1` for more than 10 ns, the simulator reports a bus float message. If the duration of the bus float is 10 ns or less, the condition is not reported.

```
xcelium> check -float sig1 -delay 10 ns
```

You can set the time limit for bus contention or bus float checks in two ways:

- By including the `-delay` option on a `check -contention` or a `check -float` command, as shown in the previous examples.
- By issuing a `check -delay` command. This sets a default time limit for subsequent checks. In the following example, the `check -delay` command sets a default time limit of 10 ns for the subsequent bus contention check.

```
xcelium> check -delay 10 ns
xcelium> check -contention sig1
```

If you do not specify a time limit using a `check -delay` command or in the `check -condition` or `check -float` command, the default time limit value is 1 fs.

Changing the default time limit using a `check -delay` command does not affect existing checks.

The bus contention or bus float time limit cannot be smaller than, or more precise than, the unit of simulation.

The `check` command checks for bus contention and bus float conditions at the following times:

- If any of the drivers of the bus signal change.
- At the end of the simulation. This is to take care of a situation when a contention or float occurs on a bus signal sometime during the simulation, and after that, none of the drivers of this bus signal change. If the check was done only on a driver change, a bus contention or bus float on such a bus signal would go undetected.

constraint

The Tcl `constraint` command allows you to add a new SystemVerilog randomization constraint to the class `randomize` call that you are debugging.

The SystemVerilog built-in `randomize()` function returns the value 1 for success or 0 for failure. A failure occurs because there are conflicts in the collection of constraints to solve or because a variable is over-constrained.

There are several commands you can use to debug these failures:

- `stop -randomize`: Sets a breakpoint in `randomize()` method calls.
- `deposit -constraint_mode`: Enables/disables a specific constraint.
- `deposit -rand_mode`: Enables/disables a specific random variable.
- `run -rand_solve`: Executes the current `randomize()` call again.

Use the `constraint` command to add a new constraint to the class `randomize` call that you are debugging. The constraint is added to the current set of constraints of the class.

A constraint can be added only to class `randomize` calls. You cannot add a constraint to a scope `randomize` call.

The command creates a persistent constraint. That is, the constraint remains in effect even after you complete the debugging session by continuing the simulation with the `run` command.

constraint Command Syntax

```
constraint "constraint_expression"
```

constraint Command Option

This section describes the options that you can use with the Tcl `constraint` command.

constraint_expression The "*constraint_expression*" argument must be in the following form:

operand_1 operator operand_2

where:

- *operand_1* and *operand_2* are either unsigned integers or variables. If they are variables, they must be random variables that are declared in the class of the current `randomize()` call, and they cannot be hierarchical references.
- The *operator* must be one of the following: `==`, `!=`, `>`, `>=`, `<`, or `<=`.

The *constraint_expression* must be enclosed in double quotes if there are spaces between the operands and the operator. For example:

```
constraint intl==200  
constraint "intl == 200"
```

coverage

The Tcl `coverage` command enables you to set up global coverage variables and generate coverage data for a simulation run. The Xcelium simulator supports code coverage analysis at all levels of design abstraction for Verilog, VHDL, and mixed-language.

You must elaborate the design with the `xmelab -coverage` option to enable coverage.

coverage Command Syntax

```
coverage
```

```
-setup  
    [-design <design>]  
    [-dut <dut_instance>]  
    [-scope <scopename>]  
    [-testname <testname>]  
    [-workdir <workdir>]  
  
-code -reset  
    [-after <time> [-absolute | relative]]
```

```
[-instance <instances>
[-depth <n> | all | to_cells]]

-fsm -reset
[-after <time> [-absolute | -relative]]

-toggle -reset
[-after <time> [-absolute | relative]]

-functional -select <assertion_selector>
[-cover] [-assert] [-assume] [-boolean] [-immediate]
[-depth {<n> | all | to_cells} [-filter <wildcard>]]
-list

-functional -deselect <assertion_selector>
[-cover] [-assert] [-assume] [-boolean] [-immediate]
[-depth {<n> | all | to_cells} [-filter <wildcard>]]
-list

-off
-dump <test>
-analyze
```

coverage Command Options

This section describes the options that you can use with the Tcl `coverage` command.

`-setup`

Sets up the global coverage variables.

The `coverage -setup` command has the following sub-options:

- **-design**: Specifies an alternate directory for storing coverage design files. By default, the design files are stored in the `./cov_work/scope` directory.
- **-dut**: Defines the hierarchy reference for reporting. This option tells the simulator to store scored coverage in the hierarchy below `<dut_instance>` with paths that start at `<dut_instance>`. This makes databases insensitive to any hierarchy above that level.
- **-scope**: Specifies an alternate directory for storing coverage model files. By default, the model files are stored in the `./cov_work/scope` directory.
- **-testname**: Specifies the test directory name for the current run. By default, the coverage data files for the current run are stored in `./cov_work/scope/test`.

- **-workdir**: Specifies an alternate directory for coverage output files. By default, the coverage files are stored in the `cov_work` directory created in the current working directory.

`-code -reset`

Resets the code coverage counts to 0.

The `coverage -code -reset` command has the following sub-options:

- **-after**: Resets the counts:
 - *At* the specified `<time>` if the `-absolute` option is used.
 - *After* the specified `<time>` if the `-relative` option is used.

If `-absolute` or `-relative` is not specified, then by default, `-relative` is assumed.
- **-instance <instances>**: Specifies the instance scope to be reset.
- **-depth**: Specifies the scope levels to descend for resetting coverage counts. It can be:
 - `<n>`: An integer value that specifies the number of scope levels to descend. For example, 1 indicates that only the specified scope is included, 2 indicates that the specified scope and its sub-scopes are included, and so on. The default value is 1.
 - `all`: selects all scopes in the hierarchy below the specified scope.
 - `<to_cells>`: Includes all scopes except the modules marked with ``celldefine` or VITAL entities with the VITAL Level0 attribute.

```
-fsm -reset  
  [-after <time>  
  [-absolute | -relative]]
```

Resets the FSM coverage counts to 0.

See [code coverage options](#) for syntax description.

```
-toggle -reset  
  [-after <time>  
  [-absolute | relative]]
```

Resets the toggle coverage counts to 0.

See [code coverage options](#) for syntax description.

`-functional -select`

Selects the functional coverage options while simulating the design.

The `coverage -functional -select` command has the following sub-options:

- **<assertion_selector>**: Specifies the scope path or the assertion for functional coverage analysis. When specifying an assertion, valid values include the assertion path, the assertion name, or a wildcard.

- **-cover**: Selects the `cover` directive.
- **-assert**: Selects the `assert` directive.
- **-assume**: Selects the `assume` directive. By default, all the directives get selected as functional coverage points.
- **-boolean**: Selects boolean type `assert` and `assume` directives.
- **-immediate**: Enables scoring of SVA immediate assertions. SVA immediate assertions are not scored inside classes.
- **-depth**: Specifies the scope levels to descend for selecting/deselecting assertions. See [code coverage options](#) for details on this option.
- **-filter**: Enables selection of directives that match `<wildcard>`. This option is used along with the `-depth` option and if used, should follow immediately after the `-depth` option.
- **-list**: Lists the coverage points being processed for generating the functional coverage data. It is helpful to list points to score after doing complex selection and deselection. You cannot list the coverage points related to SystemVerilog CoverGroup using this option.

`-functional -deselect`

Omits or deselect coverage points at any time during the functional coverage analysis process. See [select functional coverage options](#) for syntax description.

Selection and deselection commands are sequentially applied and build the final set of coverage points to be scored and stored.

`-off`

Turns off storing of coverage results immediately.

`-dump <test>`

Dumps all the currently scored coverage into `<workdir>/<scope>/<test>` immediately. If `<test>` exists, an error is reported and database dumping fails.

`-analyze`

By default, the above command dumps coverage data and launches *Integrated Metrics Center* (IMC) as the coverage analysis tool. If IMC is not found in the path from where NC is launched, an error is reported. Currently, IMC can be launched only from `<install_dir>/bin`. For more details on IMC, see the *Integrated Metrics Center User Guide*.

For more details on coverage analysis with Enterprise Manager, see the *Enterprise Manager Analyzing Metrics* guide available in the Enterprise Manager (EMGR) installation.

coverage Command Examples

To set the coverage output files locations as `mywork/mydesign/test1`, use:

```
xcelium> coverage -setup -workdir mywork -scope mydesign -test test1
```

Option, `-design` will be deprecated and unsupported from next release. Use option `-scope`.

To select all the scopes in `Monitor_ABV` and with coverage point names starting with `m`, use:

```
xcelium> coverage -functional -select Monitor_ABV -depth all -filter m*
```

You should use wildcards to match objects and not the scope. As a result, a wildcard specified with an intention of selecting scopes does not work.

The following command does not match any scopes and hence selects nothing:

```
xcelium> coverage -functional -select top.* -depth all
```

To include all of the scopes in the module `top`, use:

```
xcelium> coverage -functional -select top -depth all
```

To dump all of the currently scored coverage to `test1`, use:

```
xcelium> coverage -dump test1
```

cps

The Tcl `cps` command allows you to collect CPS information for a specific period. If this option is not specified, the CPS information is collected for the whole simulation period.

The `-cps` elaboration command option must be specified to use the `cps` Tcl command option.

cps Command Syntax

```
cps  
    [-on | -off]  
    -clear  
    -dump <filename> -overwrite <filename>
```

cps Command Options

This section describes the options that you can use with the Tcl `cps` command.

| | |
|-------------------------|--|
| <code>-on</code> | Starts collecting the CPS data. |
| <code>-off</code> | Stops collecting the CPS data. |
| <code>-clear</code> | Clears the collected CPS data. |
| <code>-dump</code> | Changes the argument of <code>-concpslog</code> simulation command-line option to <code><-filename></code> . |
| <code>-overwrite</code> | Overwrites the existing report with a new report using the specified file name. |

Important

- The default state of `cps` is `-on`.
- Multiple `cps -on` commands will retain the CPS data only from the latest `cps -on` time.
- For `cps -dump`, the `cps` must not be in the `-off` state. A report is dumped at the exit only if `cps` is not in the `-off` state.
- For `cps -dump`, if you do not specify the *filename*, there will be no change in the `-concpslog` argument. In this case, an output file with the default name, `cps_sim_log`, is dumped at the exit.

cps Command Examples

```
cps -dump cps_report.out
```

```
cps -dump -overwrite cps_sim_newlog
```

The following contents of Tcl file shows various scenarios of using the `cps` Tcl command:

```
cps -dump cps_data.out # cps data is dumped in cps_data.out file as default state of cps is -on.
```

```
run 10
```

```
cps -on
```

```
run 20
```

```
cps -dump cps_data1.out #cps_data1.out will contain cps information only from latest cps -on command.
```

```
run 30
```

```
cps -clear #clear will clear all the previous cps data
```

```
cps -dump cps_data2.out #cps data in -cps_data2.out will be empty
```

```
cps -off
```

```
cps -dump cps_data3.out # -dump command with cps in -off state will cause TCL error
```

```
cps -on
```



```
run 40
cps -dump cps_data4.out
run 50
cps -dump -overwrite cps_data4.out # Same filename without -overwrite will cause TCL error
run 100
exit # A cps report at exit is dumped if cps is not in -off state. Filename will be -concpslog
<arg>/cps_sim_log.
```

Related Topics

- [-cps](#)
- [-concpslog](#)

ctrandebug

The Tcl `ctrandebug` command invokes the real-time TRAN network debugging utility that allows you to gather debugging information about all the TRAN connected nets in a design.

To enable this `ctrandebug` Tcl command, you must specify the `xrun/xmsim/xmelab` option, `-ctrandebug` during both elaboration and simulation. This option qualifies all the TRAN network to CTRAN, irrespective of size, and disables some TRAN optimizations.

This utility is not supported with MSIE and Fault simulation.

ctrandebug Command Syntax

```
ctrandebug
[-connections] [<netpath>]
-collision
-detailed
-ctranid [<netpath>]
-resvalue [<netpath>]
-tranlist [<netpath>]
-value [<netpath>]
```

ctrandebug Command Options

This section describes the options that you can use with the Tcl `ctrandebug` command.

| | |
|---------------------------|---|
| <code>-connections</code> | Prints all the active connections of the net. |
|---------------------------|---|

| | |
|--|---|
| <code>-collision</code> | Reports the nets which are not HiZ. You must specify the <code>-connections</code> option also with the <code>-collision</code> option, else the tool displays an error. |
| <code>-detailed</code> | Prints the tranlist for each reported net. You must specify the <code>-connections</code> option also with the <code>-detailed</code> option, else the tool displays an error. |
| <code>-ctranid [<netpath>]</code> | Prints the CTRAN network ID and net ID if the specified net is part of the TRAN network. These IDs are unique for each net. |
| <code>-resvalue [<netpath>]</code> | Prints the CTRAN pre-evaluation value of the specified net. This value is derived from non-TRAN drivers. The net that does not have a driver (non-TRAN) gives the value as HiZ. |
| <code>-tranlist [<netpath>]</code> | Prints the trans directly connected to the specified net. |
| <code>-value [<netpath>]</code> | Prints the CTRAN evaluated value of the specified net. This is the final value of a net evaluated by the TRAN engine. |

ctrandebug Command Examples

This section gives examples of the output for the various options for the following design:

top.sv

```
module tran_top();
    reg r;
    wire w;
    wire wx;
    assign w1 = r;
    assign #1 w2 = !r;
    assign #2 cntrl = r;
    assign #4 cntrl2 = cntrl;
    assign cntrl3 = 1'b1;
    assign wx = 1'bx;

    initial
    begin
        #5
            r = 1'b0;
        #50
            $finish();
    end

    always
    begin
        #10
            r = ~r;
    end

    tranif1(w1, wt1, cntrl);
    tranif0(w1, wt2, cntrl2);
    rtran(wt1, wt2);
    tran(wt1, w2);
    tranif0(wt1, wx, cntrl3);
endmodule
```

The following example shows the output of the `-ctranid` option. It lists the unique CTRAN and net ID of the net, `wt1`. This also indicates that the net is part of the TRAN network.

```
xcelium> ctrandebg -ctranid tran_top.wt1
```

```
CTRAN_ID:0, NET_ID:1
```

The following example shows that the resvalue of net, `wt1` is `HiZ`, which is derived from non-TRAN drivers.

```
xcelium> ctrandebg -resvalue tran_top.wt1
```

```
HiZ
```

The following example shows the final evaluated value of the net, `wt1`.

```
xcelium> ctrandebug -value tran_top.wt1
```

StX

The following is the output of the `-tranlist` option, which lists all the trans (active and non-active) that were connected to net, `wt1`. The output shows the first pin resvalue, second pin resvalue, control pin value, and the pin position of the net, `wt1`. It also shows the instance where it was connected (`tran_top`) and the design file and line number where the net is connected to the TRAN.

```
xcelium> ctrandebug -tranlist tran_top.wt1

tran HiZ,St0,St1 PIN_1 (tran_top) ./top.v:29
rtran HiZ,HiZ,St1 PIN_1 (tran_top) ./top.v:28
tranif0 HiZ,StX,St1 PIN_1 (tran_top) ./top.v:30
tranif1 St1,HiZ,St1 PIN_2 (tran_top) ./top.v:26
```

The following example shows the output of the `-connections` option. It lists all the nets that are currently connected through the active trans to the net, `tran_top.wt1`.

```
xcelium> ctrandebug -connections tran_top.wt1

Dumping connections for ctid:0, netid:1
value:StX resvalue:HiZ netid:1 (tran_top) Expr:wt1 ./top.v:26
value:StX resvalue:St0 netid:3 (tran_top) Expr:w2 ./top.v:6
value:PuX resvalue:HiZ netid:4 (tran_top) Expr:wt2 ./top.v:27
value:StX resvalue:St1 netid:0 (tran_top) Expr:w1 ./top.v:5
```

The following example shows the output of the `-collision` option, which lists only the nets having a non-HiZ resvalue.

```
xcelium> ctrandebug -connections -collision tran_top.wt1

Dumping connections for ctid:0, netid:1
value:StX resvalue:St0 netid:3 (tran_top) Expr:w2 ./top.v:6
value:StX resvalue:St1 netid:0 (tran_top) Expr:w1 ./top.v:5
```

The following example shows the output of `-detailed` option, which lists the tranlist of the reported nets. As shown in the example, you must specify the `-connections` option also with the `-detailed` option.

```
xcelium> ctrandebug -connections -detailed tran_top.wt1

Dumping connections for ctid:0, netid:1
value:StX resvalue:HiZ netid:1 (tran_top) Expr:wt1 ./top.v:26
tran HiZ,St0,St1 PIN_1 (tran_top) ./top.v:29
rtran HiZ,HiZ,St1 PIN_1 (tran_top) ./top.v:28
tranif0 HiZ,StX,St1 PIN_1 (tran_top) ./top.v:30
tranif1 St1,HiZ,St1 PIN_2 (tran_top) ./top.v:26
value:StX resvalue:St0 netid:3 (tran_top) Expr:w2 ./top.v:6
tran HiZ,St0,St1 PIN_2 (tran_top) ./top.v:29
value:PuX resvalue:HiZ netid:4 (tran_top) Expr:wt2 ./top.v:27
rtran HiZ,HiZ,St1 PIN_2 (tran_top) ./top.v:28
```

```
tranif0 St1,HiZ,St1 PIN_2 (tran_top) ./top.v:27
value:StX resvalue:St1 netid:0 (tran_top) Expr:w1 ./top.v:5
tranif0 St1,HiZ,St1 PIN_1 (tran_top) ./top.v:27
tranif1 St1,HiZ,St1 PIN_1 (tran_top) ./top.v:26
```

The following example shows the output when you use both the `-collision` and `-detailed` options.

```
xcelium> ctrandebug -connections -collision -detailed tran_top.wt1

Dumping connections for ctid:0, netid:1
value:StX resvalue:St0 netid:3 (tran_top) Expr:w2 ./top.v:6
tran HiZ,St0,St1 PIN_2 (tran_top) ./top.v:29
value:StX resvalue:St1 netid:0 (tran_top) Expr:w1 ./top.v:5
tranif0 St1,HiZ,St1 PIN_1 (tran_top) ./top.v:27
tranif1 St1,HiZ,St1 PIN_1 (tran_top) ./top.v:26
```

Related Topic

- [Compiler Options](#)

database

The Tcl `database` command controls an SHM, Value Change Dump (VCD), or Extended Value Change Dump (EVCD) database. The database can contain SystemC objects that support the `probe` command. However, only an SHM database is supported by the Tcl `probe` command applied to SystemC objects. Options `-vcd` and `-evcd` are not supported by SystemC.

With the `database -open` option, you can open an SHM, VCD, EVCD database for Verilog, VHDL, or mixed-language. See [Opening a Database](#).

database Command Syntax

```
database [-open] <dbase_name>
        [-compress]
        [-default]
        [-evcd [direction]]
        [-mti]
        [-timescale <time_spec>]
        [-event]
        [-gzip]
        [-into <filename>]
        [-maxsize <max_size>]
        [-statement]
        [-shm]
        [-create_common_hierarchy]
        [-incfiles <number_of_incremental_files>]
        [-incsize <incremental_file_size>]
        [-inctime <time_spec>]
        [-use_common_hierarchy <path>]
        [-vcd]
        [-mti]
        [-timescale <time_value>]
        [-vcdmap <mapping_value>]
        [-change <shm_dbase_names>]
        [-close <dbase_names>]
        [-disable <dbase_names>]
        [-enable <dbase_names>]
        [-setdefault <dbase_names>]
        [-show <dbase_names>]
```

database Command Options

This section describes the options that you can use with the Tcl `database` command:

| | |
|---|--|
| <code>-open <dbase_name></code> | Creates a new database with the name specified by the <code>dbase_name</code> argument. The <code>-open</code> modifier is optional. |
| You can use the following options with <code>-open</code> : | |

- **-compress**: Compresses a database to reduce its size.

For VCD or EVCD databases, this option generates a compressed database file with a `.z` file extension. Use the `uncompress` or `gzip -d` command to uncompress the file.

You can also compress a VCD or EVCD database with the `-gzip` option.

Signal transition data is always compressed. The default level of compression requires no additional memory, and the compression time is minimal. The time spent in compression is, in almost all cases, made up by decreased I/O time since less data is written to disk. For SHM databases, data is stored in the database by signal, and the data for each signal is compressed independently. This allows data for arbitrary signals to be loaded and viewed without uncompressing an entire file. Only the data that is being loaded is uncompressed, and this is done automatically as it is read.

The `-compress` option enables maximum compression. This requires additional memory and takes more compression time but results in a smaller database.

The amount of time spent in compression, and the resulting size of the database, is highly dependent on the characteristics of the data. For example, with the `-compress` option, data that is essentially random may take extra time to write as the program unsuccessfully searches for patterns in the data, and the database may not use significantly less disk space. Without the `-compress` option, random data does not require any additional time other than the time to write that data to disk.



Important: The algorithm for compression allows complex patterns to be detected, with the potential to greatly reduce the database size. However, if the design does not contain complex patterns, the resulting database may actually be larger than an uncompressed database because the memory space is not used as effectively. If this is the case for your design, then you should use the default `database` setting, which only looks for simple patterns, such as clock signals. The default setting is designed to work best for most designs.

The `-compress` option may also not decrease the database size significantly if all of the data consists of very simple patterns that are found using the fast compression enabled by default. Most of the time, however, the data consists of a mixture of simple patterns, more complex patterns, and un-compressible data, and `-compress` produces a smaller database at the cost of some additional time and memory.

There is no penalty when reading the database if `-compress` was used to write the data. In fact, reading is faster if the compression was effective because there is less data that must be read from disk.

Because the size of the trade-off is highly dependent on the data, you might want to try both levels of compression. In general, however, if you want the highest writing speed or lowest memory use, use the default level of compression. If you want minimum database size, use `-compress`.

- **-default**: Specifies that this is the default database for all signal tracing of the same kind (SHM, VCD, or EVCD).
- **-evcd [direction] [-mti] [-timescale <timespec>]**: Specifies that this is an EVCD database.

By default, the simulator does not dump port direction information. The keyword port is dumped for all ports. For example:

```
$scope module top $end

$scope module m10 $end
$var port 1 <0 io $end
$var port [1:0] <1 vctrl $end
$upscope $end

$upscope $end

$enddefinitions $end
```

Include the direction argument if you want to dump port direction in the node information section of the output file. For example:

```
$scope module top $end

$scope module m10 $end
$var inout 1 <0 io $end
$var input [1:0] <1 vctrl $end
$upscope $end

$upscope $end

$enddefinitions $end
```

Use the **-timescale** option to set the `$timescale` value in the EVCD file to the specified timescale. This option lets you output a different timescale in the EVCD file than the timescale being used during simulation. In the output file, the times that are shown for the signal changes reflect the simulation times at the precision that you specify with the **-timescale** option.

The `timescale_value` argument can be:

- fs, 10fs, 100fs
- ps, 10ps, 100ps
- ns, 10ns, 100ns
- us, 10us, 100u
- ms, 10ms, 100ms

For example:

```
xcelium> database -open test_mp -evcd -timescale 10ps
```

The **-mti** option is a compatibility switch that changes the default format for VHDL for-generate or if-generate constructs in the header of the EVCD file. By default, the index is in parentheses. For example:

```
$scope begin ADDR_GEN(5) $end
```


If you use the `-mti` option, the format is:

```
$scope begin ADDR_GEN_5 $end
```

By default, vector ports are dumped as vectors, not as individual elements. For VHDL, you can dump individual elements of vector ports by including the `-evcd -mode lfcompat` option on the `probe` command. See [probe](#) for details.

To generate a compressed EVCD output file, include the `-compress` option on the `database` command line

- **-event**: Dumps all value changes to the database.

By default, when probing to an SHM database, the simulator discards multiple value changes for an object during one simulation time and dumps only the final value at the end of that simulation time. Use `-event` if you want to dump all value changes to the SHM database. You can then use the SimVision waveform viewer to view glitches by expanding a single moment of simulation time to show the sequence of value changes that occurred at that time.

This option is not turned on for a database when a `probe` command causes the automatic creation of a default database.

This option has no effect on VCD or EVCD databases. The simulator always dumps all value changes to a VCD or EVCD database.

If you are doing transaction-based verification, you must use the `-event` option to enable transaction recording.

- **-gzip**: Compresses a VCD or EVCD database to reduce its size.

For VCD or EVCD databases, this option generates a compressed database file with a `.gz` file extension. Use the `gzip -d` command to uncompress the file.

You can also compress a VCD or EVCD (or SHM) database with the `-compress` option.

- **-into <filename>**: Specifies the physical filename for the database. By default, the filename for an SHM database is `dbase_name.shm`, the filename for a VCD database is `dbase_name.vcd`, and the filename for an EVCD database is `dbase_name.evcd`. Use the `-into` option to override these defaults.

- **-maxsize <max_size>**: Sets a limit on the size of the database. By default, there is no size limit.

The *max_size* argument is a positive integer, which can be followed by a qualifier to indicate that the value is in bytes, KB, MB, or GB. The qualifier can be:

- B or b (Size value is in bytes. This is the default.)
- K or k (Size value is in KB.)
- M or m (Size value is in MB.)
- G or g (Size value is in GB.)

The qualifier can be separated from the value with a blank space.

For example:

```
xcelium> database -open -shm -maxsize 4000000 shmddb
(Default is bytes)
xcelium> database -open -vcd -maxsize 400K vcddb
xcelium> database -open -evcd -maxsize 4 m evcddb
```

For VCD and EVCD databases, this option specifies a limit on the number of bytes that the simulator can dump to the VCD or EVCD file. This option has the same effect as the Verilog `$dumplimit` system task for VCD. If the size of the VCD or EVCD file reaches the specified limit, the simulator inserts a comment (`dump limit reached`) into the file, and dumping stops.

In most cases, the VCD or EVCD output file size is no more than a few hundred bytes over the specified limit. However, because the header and initial values are always dumped regardless of the limit and because the limit is not checked until after the header and initial values are written, the size of the database could far exceed the limit specified using the `-maxsize` option if you have a large design in which the number of objects that you are probing is very high.

SHM (SST2) databases exist as chunks that vary in size from about 2 MB to about 4 MB. Thus, the minimum possible size of the database is somewhere between 2 MB and 4 MB. If you set the maximum size to less than this approximate range of sizes, the resulting database size can still be up to about 4 MB.

When the maximum size that you specified with the `-maxsize` option is about to be exceeded, the simulator maintains the size limit by discarding an entire chunk (2-4 MB) of the earliest recorded values. This means that if the maximum size that you specify is greater than 4 MB, the database size is below the limit, but it can be up to 4 MB below the limit.

When the size limit is exceeded, the waveform window displays no value for objects from the beginning of the simulation to the time of the first undiscarded value.

If a `probe` command automatically creates a default database, the `-maxsize` option has no effect.

- **-shm**: Specifies that this is an SHM database. This is the default.

- **-create_common_hierarchy**: Specifies the common hierarchy database to use for common signals in a shared database mode.
- **-incfiles <number_of_incremental_files>**: Specifies the maximum number of incremental files that are kept for an SHM database.

The `-incfiles` option can be used only when opening an SHM database. This option is not supported for VCD or EVCD databases.

Incremental files are created:

- Through the `-incsize` option
- By a database `-change` command
- By `save`, `restart`, or `reset` commands

By default, all incremental files are preserved. Use the `-incfiles` option to set a limit on the number of incremental files that are kept for the database.

The `number_of_incremental_files` argument is an integer that specifies the number of incremental files to keep. When more than the specified number of incremental files have been written, the oldest files are deleted automatically.

Examples:

The following command specifies that no more than four incremental files are to be kept for the database.

```
xcelium> database -open -shm shmdb -incfiles 4
```

The following command specifies that the size of the incremental files is 1 MB, and that no more than four incremental files are to be kept for the database.

```
xcelium> database -open -shm shmdb -incsize 1M -incfiles 4
```

- **-incsize incremental_file_size**: Specifies the incremental file size for the SHM database.

Use the `-incsize` option only when opening an SHM database. This option is not supported for VCD or EVCD databases.

By default, there is no limit on the size of an SHM database. Because a database for a large simulation can be very big, you may want to break up the signal transition information (the `.trn` file) and, if you are tracing statement data, the statement trace information (the `.stc` file) into multiple files. These files correspond to a range of simulation time and are called *incremental files*.

Breaking up a large database file into incremental files can make the simulation results more manageable. You can open just one incremental file, or any subset of the files, in SimVision so that you can view the waveforms for the time range(s) corresponding to that file or set of files. This can improve viewer performance and memory usage. Incremental files can also be used to ensure that database files are kept under some specified size (for example, 2 GB), and files corresponding to uninteresting time ranges can be deleted to save disk space.

Use the `-incsize` option to specify a size limit for the SHM database file. When the current SHM database file size approximates the size specified, a new incremental file is started automatically.

The *incremental_file_size* argument is an integer, which can be followed by a qualifier to indicate that the value is in MB or GB. The qualifier can be:

- M or m (Size value is in MB. This is the default.)
- G or g (Size value is in GB.)

The qualifier can be separated from the value with a blank space.

For example:

```
xcelium> database -open -shm shmdb -incsize 4 (Default is MB)
xcelium> database -open -shm shmdb -incsize 4M
xcelium> database -open -shm shmdb -incsize 4 M
xcelium> database -open -shm shmdb -incsize 1G
xcelium> database -open -shm shmdb -incsize 1 g
```

The actual size of the incremental files can vary from the specified size because the file size is checked on simulation time changes, and the buffered data is subjected to further compression.

`xcelium> database -open -shm xmsim -incsize 2M`The initial database file and all incremental files are stored in the database directory. The incremental files have a number included in the filename. For example, the following command opens a database called `xmsim`.

The initial database file is called `xmsim.trn`. The incremental files are called `xmsim-1.trn`, `xmsim-2.trn`, and so on.

The same incremental file number is used for both `.trn` and `.stc` files, and if both are being written, both files are changed at the same time.

You can include the `-incfiles` option to set a limit on the number of incremental files that are kept for the database.

If you specify both the `-incsize` and `-maxsize` options when opening an SHM database, the `-maxsize` option is ignored.

- **-inctime <timespec>**: Specifies that incremental database files are to be dumped at specified time intervals.

Use the `-inctime` option only when opening an SHM database. This option is not valid for analog simulation.

By default, there is no limit on the simulation time stored in an SHM database. Because the database for some simulations can be extremely large, you may want to break up the signal transaction information (the `.trn` file) into multiple files called incremental files. Use the `-inctime` option to specify that the incremental files are to be dumped at regular time intervals.

The `time_spec` argument specifies the periodic time interval at which to start creating a new incremental database file. The argument can include a time unit. If a time unit is not specified, the default simulation time unit is used.

```
xcelium> database -open -shm shmdb -inctime 1000
xcelium> database -open -shm shmdb -inctime 1000 ns
```

You can include the `-incfiles` option to set a limit on the number of incremental files that are kept for the database. By default, all incremental files are preserved.

```
xcelium> database -open -shm shmdb9 -inctime 1000 ns -
incfiles 10
```

The initial database file and all incremental files are stored in the database directory. The incremental files have a number included in the filename. For example, the following command opens a database called `xmsim`.

```
xcelium> database -open -shm xmsim -inctime 200 ns
```

The initial database file is called `xmsim.trn`. The incremental files are called `xmsim-1.trn`, `xmsim-2.trn`, and so on.

You cannot use `-inctime` with the `-incsize` option.

- **-use_common_hierarchy <path>**: Specifies the common hierarchy database to use for common signals in a shared database mode

- **-statement**: Specifies that statement trace data should be written to the SHM database. After writing statement trace data to the SHM database, you can use the *Explore - Go To - Cause* command in SimVision to help you track down the cause of a signal transition with either the Source Code Browser or the Trace Signals sidebar. See [Finding the Cause of a Signal Transition](#)

For Verilog designs, you can also use the `$recordvars` system task to dump statement trace information.

You must compile design units with the `-linedebug` option to record statement trace data.

You can only write statement trace data into an SHM database. The simulator generates an error if you use the `-vcd` or `-evcd` option with the `-statement` option.

Using the `-statement` option creates an SHM database with the `-event` option. All value changes are dumped to the database.

You can probe specific objects, inputs, outputs, or ports to a database opened with `-statement`. However, no statement trace data are recorded unless you specify one or more scopes as the argument to the `probe` command. All statements within the specified scope(s) are traced. Statements are not traced if you specify an object or use the `-inputs`, `-outputs`, or `-ports` options.



Recording statement trace information can have a severe performance impact, and the simulator issues a warning message (DBSPER) to this effect when you open a database with the `-statement` option.

- **-vcd**: Specifies that this is a VCD database.

You can specify the following:

- **-mti**: Use the `-mti` option to change the default format for VHDL for-generate or if-generate constructs in the header of the VCD file. By default, the index is in parentheses. For example:

```
$scope begin ADDR_GEN(5) $end
```

If you use the `-mti` option, the format is:

```
$scope begin ADDR_GEN_5 $end
```

To generate a compressed VCD output file, include the `-compress` option on the database command line.

- **-timescale timescale_value**: Use the `-timescale` option to set the `$timescale` value in the VCD file to the specified timescale. This option lets you output a different timescale in the VCD file than the timescale being used during simulation. In the output file, the times that are shown for the signal changes reflect the simulation times at the precision

that you specify with the `-timescale` option.

The `timescale_value` argument can be:

- fs, 10fs, 100fs
- ps, 10ps, 100ps
- ns, 10ns, 100ns
- us, 10us, 100us
- ms, 10ms, 100ms

For example:

```
xcelium> database -open test_mp -vcd -timescale 10ps
```

- **-vcdmap vcd_mapping:** Specifies a user-defined mapping of VHDL `std_logic` values to the four states for VCD (1, 0, X, Z). By default, the nine values of `STD_LOGIC` (U, X, 0, 1, Z, W, L, H, -) are mapped to (X, X, 0, 1, Z, X, 0, 1, X).

The `vcd_mapping` argument is a string of nine valid VCD values. For example:

```
xcelium> database -open myvcd.vcd -vcd -vcdmap  
XXXXZ1111
```

You can also specify the mapping by setting the Tcl `vhdl_vcdmap` variable. For example:

```
xcelium> set vhdl_vcdmap XXXXZ1111
```

The mapping specified by setting the Tcl variable sets the mapping to be used for all VCD databases that you open. If a VCD mapping is defined by setting the Tcl variable and by specifying the `database -vcd -vcdmap` option, the mapping defined by the `database` command is used.

```
-change  
<shm_database_name>  
[<shm_database_name>...]
```

Starts a new incremental SHM database file, immediately. A new incremental file is created without saving the simulation snapshot.

`database -change` is not supported for VCD or EVCD databases.

A `database -change` command forces a new incremental SHM database file to be started. You can use this command at significant points during the simulation, instead of, or in addition to, automatic incremental file creation using `-incsize`, so that incremental files contain specific time ranges of interest.

The incremental files have the same name as the initial database file but with a number following the name of the database. For example, if the database is called `xmsim`, the initial `.trn` file is `xmsim.trn`. The incremental files are called `xmsim-1.trn`, `xmsim-2.trn`, and so on.

```
-close {<dbase_name>|  
<pattern>}...
```

Closes the database(s) specified by the argument. The argument can be a database, a list of database names, a combination of literal database names and patterns, and wildcard patterns.

| | |
|---|---|
| <code>-disable {<dbase_name> <pattern>}...</code> | Disables the tracing of data into the database(s) specified by the argument. The argument can be a database, a list of database names, a combination of literal database names and patterns, and wildcard patterns. |
| <code>-enable {<dbase_name> <pattern>}...</code> | Resumes the tracing of data into the database(s) specified by the argument. The argument can be a database, a list of database names, a combination of literal database names and patterns, and wildcard patterns. |
| <code>-setdefault <dbase_name> [<dbase_name>...]</code> | <p>Makes the specified database(s) the default database for its kind. For example, the following command makes the database <code>waves.shm</code> the default SHM database:</p> <pre>xcelium> database -setdefault waves.shm</pre> <p>This modifier is useful if you want to specify that a previously opened database is now to be used as the default database for probes and other operations. For example, suppose that you open a database with the following <code>\$shm_open</code> task in your Verilog HDL:</p> <pre>\$shm_open("waves.shm");</pre> <p>This opens a database called <code>_waves.shm</code> (the underscore character indicates that the database was opened from within the HDL). If you then want to add additional probes to this database, you can:</p> <ul style="list-style-type: none">• Use the <code>probe -database _waves.shm</code> command. The <code>-database</code> option specifies that you want the data saved in <code>_waves.shm</code>. In SimVision, use the <i>Simulation - Create Probe</i> command and specify the database on the Set Probe form.• Use the <code>-setdefault</code> modifier to make <code>_waves.shm</code> the default SHM database, and then probe the signals. If you do not make <code>_waves.shm</code> the default database, the simulator opens a default SHM database called <code>xcelium.shm</code> for you, and the signals are probed to that database. |
| <code>-show [{<dbase_name> <pattern>}...]</code> | <p>Displays information about the database(s) specified by the argument. If you do not specify an argument, the simulator displays information about all open databases.</p> <p>The argument can be a database, a list of database names, a combination of literal database names and patterns, and wildcard patterns.</p> |

database Command Examples

The following command opens an SHM database named `waves` and places the data into the file `waves.shm`. The `-open` modifier is optional. The `-shm` option is the default:

```
xcelium> database -open -shm waves
Created SHM database waves
```

The following command opens an SHM database named `waves` and places the data into the file `mywaves.shm` :

```
xcelium> database waves -into mywaves.shm
Created SHM database waves
```

The following command opens an SHM database named `waves` and places the data into the

file `waves.shm`. The `-default` option specifies that `waves` is the default database:

```
xcelium> database waves -default
Created default SHM database waves
```

The following command includes the `-compress` option, which compresses the SHM database:

```
xcelium> database -open waves -compress -default
Created default SHM database waves
```

The following command includes the `-maxsize` option to limit the size of the database to 4 MB:

```
xcelium> database -open -shm shmdb -maxsize 4M
```

The following command includes the `-incsize` option, which specifies a size limit of 1 GB for the SHM database file. When the current SHM database file size reaches 1 GB, a new incremental file is started automatically.

```
xcelium> database -open -shm shmdb -incsize 1G
```

The following command includes the `-incsize` option, which specifies a size limit of 1 GB for the SHM database file, and the `-incfiles` option, which specifies that a maximum of 5 incremental files is to be kept.

```
xcelium> database -open -shm shmdb -incsize 1G -incfiles 5
```

The following command opens a database called `shmdb`. The initial database file is called `shmdb.trn`. A new incremental database file are dumped every 100 nanoseconds. The incremental files are called `shmdb-1.trn`, `shmdb-2.trn`, and so on.

```
xcelium> database -open -shm shmdb -inctime 100 ns
```

The following command includes the `-incfiles` option to specify the number of incremental files to be kept:

```
xcelium> database -open -shm shmdb -inctime 100 ns -incfiles 10
```

The following command opens an SHM database named `waves`. The `-statement` option specifies that statement trace information is to be recorded in the database.

```
xcelium> database -open waves -statement
xmsim: *W,DBSPER: The database -statement option will have an adverse performance impact. The
-LINEDEBUG option must be used during compilation to probe statements.
Created SHM database waves
xcelium>
```

The following command opens a VCD database named `vcddb` and places the data into the file `vcddb.vcd`:

```
xcelium> database -open -vcd vcddb
Created VCD database vcddb
```

The following command opens a VCD database named `vcddb`. The filename of the database is `verilog.dump`. The `-timescale` option sets the `$timescale` value in the VCD file to 1 ns. Value changes

in the VCD file are scaled to 1 ns. The `-compress` option compresses the file. The output file is called `verilog.dump.Z`.

```
xcelium> database -open -vcd vcddb -into verilog.dump -timescale ns -compress
Created VCD database vcddb
```

The following command opens an EVCD database named `evcddb` and places the data into the file `evcddb.evcd`:

```
xcelium> database evcddb -evcd
Created EVCD database evcddb
```

The following command opens an EVCD database named `evcddb`. The database is compressed, and the output file is called `evcddb.evcd.Z`:

```
xcelium> database evcddb -evcd -compress
Created EVCD database evcddb
```

The following command displays information about the status of all databases:

```
xcelium> database -show
```

The following command displays information about the status of all databases that have names that start with `db`:

```
xcelium> database -show db*
```

The following command displays information about the status of all databases that have names that start with `db` and end with `1`:

```
xcelium> database -show db*1
```

The following command displays information about the status of all databases that have names that start with `v` or `w`:

```
xcelium> database -show v* w*
```

The following command is identical to the previous command. It displays information about the status of all databases that have names that start with `v` or `w`. The curly braces suppress command substitution with square brackets.

```
xcelium> database -show {[vw]*}
```

The following command displays information about the status of all databases that have names that have three characters, starting with `db`:

```
xcelium> database -show db?
```

The following command displays information about the status of the database called `waves` and of all databases with names that start with `db`:

```
xcelium> database -show waves db*
```

The following command disables the databases named `db1` and `db2`:

```
xcelium> database -disable db1 db2
```

The following command enables the database named `db1`:

```
xcelium> database -enable db1
```

The following sequence of commands includes two `database -change` commands. These commands are used to create new incremental SHM database files, which contain data for specific time ranges. The incremental files are called `xmsim-1.trn` and `xmsim-2.trn`.

```
xcelium> database -open -shm xmsim
Created SHM database xmsim
xcelium> probe -create -database xmsim -all -depth all
Created probe 1
xcelium> run 1000 ns
Ran until 1 US + 0
xcelium> database -change xmsim
New incremental file started for SHM database: xmsim
xcelium> run 1000 ns
Ran until 2 US + 0
xcelium> database -change xmsim
New incremental file started for SHM database: xmsim
xcelium> run 1000 ns
Ran until 3 US + 0
```

The following command closes the database called `xmsim`:

```
xcelium> database -close xmsim
```

The following command closes all databases that have names that start with `db`:

```
xcelium> database -close db*
```

Related Topics

- [Opening a Database](#)
- [Wildcards Characters in Tcl Commands](#)

Opening a Database

Use the `-open` modifier with the `database` command to open a database. You can open the following kinds of databases for Verilog, VHDL, or mixed-language:

- SHM (the default)
- VCD
- EVCD

After opening an SHM, VCD, or EVCD database, you can then probe the items that you want to dump to the database by using the `probe` command. See [probe](#) for information on probing objects to a database with the `probe` command.

For Verilog, in addition to creating an SHM, VCD, or EVCD database with the `Tcl database` and `probe` commands, you can:

- Create an SHM database by using the `$shm_open` and `$shm_probe` system tasks in your Verilog source code.

For backward compatibility with Verilog code that contains calls to the Signalscan tasks for recording data (`$recordvars`, `$recordfile`, `$recordsetup`, and so on), Cadence has implemented these tasks as system tasks native to the simulator.

- Create a VCD database by using value change dump system tasks (`$dumpfile`, `$dumpvars`, `$dumpall`, and so on) in your Verilog source code.
- Create an EVCD database by using the `$dumpports` system task.

For VHDL, in addition to creating a VCD database with the `database` and `probe` commands, you can also open a VCD database and probe objects to the database by using the `call` command to call predefined CFC routines, which are part of the Xcelium simulator C interface. This feature has been retained for backward compatibility. The recommended method of generating a VCD file is to open a database with the `database -open -vcd` command and to probe objects to the database with the `probe -vcd` command.

Syntax

```
database [-open] <dbase_name> [-shm | -vcd | -evcd]
```

Open an SHM Database

```
database [-open] <dbase_name> [-shm]
    [-incfiles <number_of_incremental_files>]
    [-incsize <incremental_file_size>]
    [-inctime <time_spec>]
    [-use_common_hierarchy <path>]
    [-compress]
    [-default]
    [-event]
    [-into <filename>]
    [-maxsize <max_size>]
    [-statement]
```

Open a VCD Database

```
database [-open] <dbase_name> -vcd
    [-mti]
    [-timescale <time_value>]
    [-vcdmap <mapping_value>]
```

```
[-compress | -gzip]  
[-default]  
[-into <filename>]  
[-maxsize <max_size>]
```

Open an EVCD Database

```
database [-open] [direction] <dbase_name> -evcd  
        [-mti]  
        [-timescale <time_value>]  
        [-compress | -gzip]  
        [-default]  
        [-into <filename>]  
        [-maxsize <max_size>]
```

Options

`-open <dbase_name>`

Creates a new database with the name specified by the `dbase_name` argument. The `-open` modifier is optional.

You can use the following options with `-open`:

- **-compress**: Compresses a database to reduce its size.


For VCD or EVCD databases, this option generates a compressed database file with a `.z` file extension. Use the `uncompress` or `gzip -d` command to uncompress the file.

You can also compress a VCD or EVCD database with the `-gzip` option.

Signal transition data is always compressed. The default level of compression requires no additional memory, and the compression time is minimal. The time spent in compression is, in almost all cases, made up by decreased I/O time since less data is written to disk. For SHM databases, data is stored in the database by signal, and the data for each signal is compressed independently. This allows data for arbitrary signals to be loaded and viewed without uncompressing an entire file. Only the data that is being loaded is uncompressed, and this is done automatically as it is read.

The `-compress` option enables maximum compression. This requires additional memory and takes more compression time but results in a smaller database.

The amount of time spent in compression, and the resulting size of the database, is highly dependent on the characteristics of the data. For example, with the `-compress` option, data that is essentially random may take extra time to write as the program unsuccessfully searches for patterns in the data, and the database may not use significantly less disk space. Without the `-compress` option, random data does not require any additional time other than the time to write that data to disk.

 **Important:** The algorithm for compression allows complex patterns to be detected, with the potential to greatly reduce the database size. However, if the design does not contain complex patterns, the resulting database may actually be larger than an uncompressed database because the memory space is not used as effectively. If this is the case for your design, then you should use the default `database` setting, which only looks for simple patterns, such as clock signals. The default setting is designed to work best for most designs.

The `-compress` option may also not decrease the database size significantly if all of the data consists of very simple patterns that are found using the fast compression enabled by default. Most of the time, however, the data consists of a mixture of simple patterns, more complex patterns, and un-compressible data, and `-compress` produces a smaller database at the cost of some additional time and memory.

There is no penalty when reading the database if `-compress` was used to write the data. In fact, reading is faster if the compression was effective because there is less data that must be read from disk.

Because the size of the trade-off is highly dependent on the data, you might want to try both levels of compression. In general, however, if you want the highest writing speed or lowest memory use, use the default level of compression. If you want minimum database size, use `-compress`.

- **-default**: Specifies that this is the default database for all signal tracing of the same kind (SHM, VCD, or EVCD).

- **-evcd [direction] [-mti] [-timescale <time_value>]**:

Specifies that this is an EVCD database.

The simulator does not dump port direction information for an EVCD database by default. The keyword port is dumped for all ports. Include the `direction` argument if you want to dump port direction in the node information section of the output file.

Use the `-timescale` option to set the `$timescale` value in the EVCD file to the specified timescale. This option lets you output a different timescale in the EVCD file than the timescale being used during simulation. In the output file, the times that are shown for the signal changes reflect the simulation times at the precision that you specify with the `-timescale` option.

The `timescale_value` argument can be:

- fs, 10fs, 100fs
- ps, 10ps, 100ps
- ns, 10ns, 100ns
- us, 10us, 100u
- ms, 10ms, 100ms

The `-mti` option is a compatibility switch that changes the default format for VHDL for-generate or if-generate constructs in the header of the EVCD file. By default, the index is in parentheses.

By default, vector ports are dumped as vectors, not as individual elements. For VHDL, you can dump individual elements of vector ports by including the `-evcd -mode lfcompat` option on the `probe` command.

See [probe](#) for details.

To generate a compressed EVCD output file, include the `-compress` option on the `database` command line.

- **-event**: Dumps all value changes to the database.

By default, when probing to an SHM database, the simulator discards multiple value changes for an object during one simulation time and dumps only the final value at the end of that simulation time. Use `-event` if you want to dump all value changes to the SHM database. You can then use the SimVision waveform viewer to view glitches by expanding a single moment of simulation time to show the sequence of value changes that occurred at that time.

This option is not turned on for a database when a `probe` command causes the automatic creation of a default database.

This option has no effect on VCD or EVCD databases. The simulator always dumps all value changes to a VCD or EVCD database.

If you are doing transaction-based verification, you must use the `-event` option to enable transaction recording.

- **-gzip**: Compresses a VCD or EVCD database to reduce its size.

For VCD or EVCD databases, this option generates a compressed database file with a `.gz` file extension. Use the `gzip -d` command to uncompress the file.

You can also compress a VCD or EVCD (or SHM) database with the `-compress` option.

- **-into <filename>**: Specifies the physical filename for the database. By default, the filename for an SHM database is `dbase_name.shm`, the filename for a VCD database is `dbase_name.vcd`, and the filename for an EVCD database is `dbase_name.evcd`. Use the `-into` option to override these defaults.

- **-maxsize <max_size>**: Sets a limit on the size of the database. By default, there is no size limit.

The *max_size* argument is a positive integer, which can be followed by a qualifier to indicate that the value is in bytes, KB, MB, or GB. The qualifier can be:

- B or b (Size value is in bytes. This is the default.)
- K or k (Size value is in KB.)
- M or m (Size value is in MB.)
- G or g (Size value is in GB.)

The qualifier can be separated from the value with a blank space.

For VCD and EVCD databases, this option specifies a limit on the number of bytes that the simulator can dump to the VCD or EVCD file. This option has the same effect as the Verilog `$dumplimit` system task for VCD. If the size of the VCD or EVCD file reaches the specified limit, the simulator inserts a comment (`dump limit reached`) into the file, and dumping stops.

In most cases, the VCD or EVCD output file size is no more than a few hundred bytes over the specified limit. However, because the header and initial values are always dumped regardless of the limit and because the limit is not checked until after the header and initial values are written, the size of the database could far exceed the limit specified using the `-maxsize` option if you have a large design in which the number of objects that you are probing is very high.

SHM (SST2) databases exist as chunks that vary in size from about 2 MB to about 4 MB. Thus, the minimum possible size of the database is somewhere between 2 MB and 4 MB. If you set the maximum size to less than this approximate range of sizes, the resulting database size can still be up to about 4 MB.

When the maximum size that you specified with the `-maxsize` option is about to be exceeded, the simulator maintains the size limit by discarding an entire chunk (2-4 MB) of the earliest recorded values. This means that if the maximum size that you specify is greater than 4 MB, the database size is below the limit, but it can be up to 4 MB below the limit.

When the size limit is exceeded, the waveform window displays no value for objects from the beginning of the simulation to the time of the first undiscarded value.

If a `probe` command automatically creates a default database, the `-maxsize` option has no effect.

- **-shm**: Specifies that this is an SHM database. This is the default.
- **-create_common_hierarchy**: Specifies the common hierarchy database to use for common signals in a shared database mode.

- **-incfiles** *<number_of_incremental_files>*: Specifies the maximum number of incremental files that are kept for an SHM database.

The `-incfiles` option can be used only when opening an SHM database. This option is not supported for VCD or EVCD databases.

Incremental files are created:

- Through the `-incsize` option
- By a database `-change` command
- By `save`, `restart`, or `reset` commands

By default, all incremental files are preserved. Use the `-incfiles` option to set a limit on the number of incremental files that are kept for the database.

The *number_of_incremental_files* argument is an integer that specifies the number of incremental files to keep. When more than the specified number of incremental files have been written, the oldest files are deleted automatically.

- `-incsize incremental_file_size`: Specifies the incremental file size for the SHM database.

Use the `-incsize` option only when opening an SHM database. This option is not supported for VCD or EVCD databases.

By default, there is no limit on the size of an SHM database. Because a database for a large simulation can be very big, you may want to break up the signal transition information (the `.trn` file) and, if you are tracing statement data, the statement trace information (the `.stc` file) into multiple files. These files correspond to a range of simulation times and are called *incremental files*.

Breaking up a large database file into incremental files can make the simulation results more manageable. You can open just one incremental file, or any subset of the files, in SimVision so that you can view the waveforms for the time range(s) corresponding to that file or set of files. This can improve viewer performance and memory usage. Incremental files can also be used to ensure that database files are kept under some specified size (for example, 2 GB), and files corresponding to uninteresting time ranges can be deleted to save disk space.

Use the `-incsize` option to specify a size limit for the SHM database file. When the current SHM database file size approximates the size specified, a new incremental file is started automatically.

The `incremental_file_size` argument is an integer, which can be followed by a qualifier to indicate that the value is in MB or GB. The qualifier can be:

- M or m (Size value is in MB. This is the default.)
- G or g (Size value is in GB.)

The qualifier can be separated from the value with a blank space.

The actual size of the incremental files can vary from the specified size because the file size is checked on simulation time changes, and the buffered data is subjected to further compression.

You can include the `-incfiles` option to set a limit on the number of incremental files that are kept for the database.

If you specify both the `-incsize` and `-maxsize` options when opening an SHM database, the `-maxsize` option is ignored.

- **-inctime <time_spec>**: Specifies that incremental database files are to be dumped at specified time intervals.

Use the `-inctime` option only when opening an SHM database. This option is not valid for analog simulation.

By default, there is no limit on the simulation time stored in an SHM database. Because the database for some simulations can be extremely large, you may want to break up the signal transaction information (the `.trn` file) into multiple files called incremental files. Use the `-inctime` option to specify that the incremental files are to be dumped at regular time intervals.

The `time_spec` argument specifies the periodic time interval at which to start creating a new incremental database file. The argument can include a time unit. If a time unit is not specified, the default simulation time unit is used.

You can include the `-incfiles` option to set a limit on the number of incremental files that are kept for the database. By default, all incremental files are preserved.

You cannot use `-inctime` with the `-incsize` option.

- **-use_common_hierarchy <path>**: Specifies the common hierarchy database to use for common signals in a shared database mode
- **-statement**: Specifies that statement trace data should be written to the SHM database. After writing statement trace data to the SHM database, you can use the *Explore - Go To - Cause* command in SimVision to help you track down the cause of a signal transition with either the Source Code Browser or the Trace Signals sidebar. See [Finding the Cause of a Signal Transition](#)


For Verilog designs, you can also use the `$recordvars` system task to dump statement trace information.

You must compile design units with the `-linedebug` option to record statement trace data.

You can only write statement trace data into an SHM database. The simulator generates an error if you use the `-vcd` or `-evcd` option with the `-statement` option.

Using the `-statement` option creates an SHM database with the `-event` option. All value changes are dumped to the database.

You can probe specific objects, inputs, outputs, or ports to a database opened with `-statement`. However, no statement trace data are recorded unless you specify one or more scopes as the argument to the `probe` command. All statements within the specified scope(s) are traced. Statements are not traced if you specify an object or use the `-inputs`, `-outputs`, or `-ports` options.

 Recording statement trace information can have a severe performance impact, and the simulator issues a warning message (DBSPER) to this effect when you open a database with the `-statement` option.

- **-vcd**: Specifies that this is a VCD database.

You can specify the following:

- **-mti**: Use the **-mti** option to change the default format for VHDL for-generate or if-generate constructs in the header of the VCD file. By default, the index is in parentheses. For example:

```
$scope begin ADDR_GEN(5) $end
```

If you use the **-mti** option, the format is:

```
$scope begin ADDR_GEN_5 $end
```

- **-timescale <time_value>**: Use the **-timescale** option to set the **\$timescale** value in the VCD file to the specified timescale. This option lets you output a different timescale in the VCD file than the timescale being used during simulation. In the output file, the times that are shown for the signal changes reflect the simulation times at the precision that you specify with the **-timescale** option.

The **<time_value>** argument can be:

- fs, 10fs, 100fs
- ps, 10ps, 100ps
- ns, 10ns, 100ns
- us, 10us, 100us
- ms, 10ms, 100ms

For example:

```
xcelium> database -open test_mp -vcd -timescale 10ps
```

- **-vcdmap <mapping_value>**: Specifies a user-defined mapping of VHDL **std_logic** values to the four states for VCD (1, 0, X, Z). By default, the nine values of **STD_LOGIC** (U, X, 0, 1, Z, W, L, H, -) are mapped to (X, X, 0, 1, Z, X, 0, 1, X).

The **<mapping_value>** argument is a string of nine valid VCD values. For example:

```
xcelium> database -open myvcd.vcd -vcd -vcdmap  
XXXXZ1111
```

You can also specify the mapping by setting the Tcl **vhdl_vcdmap** variable. For example:

```
xcelium> set vhdl_vcdmap XXXXZ1111
```

The mapping specified by setting the Tcl variable sets the mapping to be used for all VCD databases that you open. If a VCD mapping is defined by setting the Tcl variable and by specifying the **database -vcd -vcdmap** option, the mapping defined by the **database** command is used.

Examples

The following command opens an SHM database named `waves` and places the data into the file `mywaves.shm`:

```
xcelium> database waves -into mywaves.shm
Created SHM database waves
```

The following command opens an SHM database named `waves` and places the data into the file `waves.shm`. The `-default` option specifies that `waves` is the default database:

```
xcelium> database waves -default
Created default SHM database waves
```

The following command includes the `-maxsize` option to limit the size of the database to 4 MB:

```
xcelium> database -open -shm shmdb -maxsize 4M
```

The following command includes the `-incsize` option, which specifies a size limit of 1 GB for the SHM database file. When the current SHM database file size reaches 1 GB, a new incremental file is started automatically.

```
xcelium> database -open -shm shmdb -incsize 1G
```

The following command includes the `-incsize` option, which specifies a size limit of 1 GB for the SHM database file, and the `-incfiles` option, which specifies that a maximum of 5 incremental files are to be kept.

```
xcelium> database -open -shm shmdb -incsize 1G -incfiles 5
```

The following command includes the `-incfiles` option to specify the number of incremental files to be kept:

```
xcelium> database -open -shm shmdb -inctime 100 ns -incfiles 10
```

The following command opens an SHM database named `waves`. The `-statement` option specifies that statement trace information is to be recorded in the database.

```
xcelium> database -open waves -statement
xmsim: *W,DBSPER: The database -statement option will have an adverse performance impact. The
-LINEDEBUG option must be used during compilation to probe statements.
Created SHM database waves
xcelium>
```

The following command opens a VCD database named `vcddb` and places the data into the file `vcddb.vcd`:

```
xcelium> database -open -vcd vcddb
Created VCD database vcddb
```

Related Topics

- [Managing the SHM Database](#)
- [The Value Change Dump \(VCD\) File](#)
- [The Extended Value Change Dump \(EVCD\) File](#)
- [Generating a VCD File Using CFC Routines](#)
- [Using \\$recordvars and Related Tasks](#)

Setting a Default Database

Use the `-setdefault` modifier with the `database` command if you have opened a database and want to specify that this previously opened database is now to be used as the default for probes and other operations.

Syntax

```
database -setdefault dbase_name
```

Example

The following command makes the database, `waves.shm`, the default SHM database.

```
xcelium> database -setdefault waves.shm
```

Displaying Information about Databases

Use the `-show` modifier with the `database` command to display information about databases.

Syntax

```
database -show [{dbase_name | pattern} ...]
```

Deactivating a Database

Use the `-disable` modifier with the `database` command to temporarily deactivate a database.

Syntax

```
database -disable [{dbase_name | pattern} ...]
```

Enabling a Database

Use the `-enable` modifier with the `database` command to enable a previously deactivated database.

Syntax

```
database -enable {dbase_name | pattern} ...
```

Creating Incremental SHM Database Files

By default, when using the `database` command to create an SHM database, there is no limit on its size. However, if your environment has disk space or other custom constraints, you may want to break up the signal transition information (the `.trn` file) and, if you are tracing statement data, the statement trace information (the `.stc` file) into multiple files for improved database management instead of managing one very big file. These files correspond to a range of simulation times and are called *incremental files*.

You can create incremental files by using the `-incsize` modifier when you open the SHM database. This specifies the incremental file size for the SHM database. For example, the following command specifies a size limit of 1GB for the SHM database file. When the current SHM database file size reaches 1GB, a new incremental file is started automatically.

```
xcelium> database -open -shm shmdb -incsize 1G
```

The initial database file and all incremental files are stored in the database directory. The incremental files have a number included in the filename. For example, the following command opens a database called `mysim`.

```
xcelium> database -open -shm mysim -incsize 2M
```

The initial database file is called `mysim.trn`. The incremental files will be called `mysim-1.trn`, `mysim-2.trn`, and so on.

You can include the `-incfiles` modifier to set a limit on the number of incremental files that will be kept for the database.

Incremental files can also be created at any time with a `database -change` command. For example:

```
xcelium> database -change shmdb
```

A `database -change` command forces a new incremental SHM database file to be started. You can use this command at significant points during the simulation, instead of, or in addition to, automatic incremental file creation using `-incsize`, so that incremental files contain specific time ranges of interest.

Related Topic

- [Starting a New Incremental SHM Database File](#)

Closing a Database

Use the `-close` modifier with the `database` command to close a database.

Syntax

```
database -close {dbase_name | pattern} ...
```

deposit

The Tcl `deposit` command sets the value of an object. Behaviors sensitive to the object's value changes run when the simulation resumes, as if the value change was caused by the Verilog or VHDL code.

You can specify the `deposit` command to take effect immediately at a time in the future (`-after -absolute`), or after some amount of time has passed (`-after -relative`). Use the `-inertial` or `-transport` option to deposit the value after an inertial delay or after a transport delay, respectively.

The `deposit` command without a delay is similar to a force in which the specified value takes effect and propagates immediately. You can also use this command as another way to ask "what if" questions about your model as you perform interactive debugging in Tcl. However, a deposit differs from a force in that future transactions on the signal are not blocked.

In VHDL, a deposit with a delay is different from Verilog, in which it creates a transaction on a driver, much the same as a VHDL signal assignment statement.

The object to which the value is to be deposited must have write access. An error is generated if the object does not have this access. See [Access to Simulation Objects](#).

The following are common tasks you can perform with the `deposit` command:

- [Depositing to Verilog Objects](#)
- [Depositing to VHDL Objects](#)
- [Depositing to SystemC Objects](#)
- [Depositing to IEEE 1801 Supply Nets](#)
- [Debugging and Random Stability](#)

deposit Command Syntax

```
deposit object_name [=] <value>
    [-after <time_spec> [<value> -after <time_spec>...]]
    [{-relative | -absolute}]
    [-cancel 0]
    [-constraint_mode <constraint_name> [=] {0|1}]
    [-generic]
    [-inertial]
    [-rand_mode <object_name> [=] {0|1}]
    [-release]
    [-repeat <period> [-cancel <period>]]
    [-scope <full_path_from_design_top>]
        [-allone]
        [-allzero]
        [-depth <positive_number | "all">]
        [-exclude_module <module_name>]
        [-exclude_net <net_name | module_name.net_name>]
        [-exclude_scope <path_from_design_top>]
        [-frc_rel_ports <in|out|inout|all>]
        [-if_x]
        [-if_z]
        [-logfile <path_to_file>]
            [-logfile_overwrite]
        [-logfilter <traverse | deposit | unmarked | skipped>]
        [-random]
            [-sense_input]
        [-verbose <default|debug>]
    [-transport]

deposit -supply_on | -supply_partial_on <net_name> <voltage>
    [-supply_off <net_name>]
```

When specifying `-supply_on`, `-supply_partial_on`, or `-supply_off`, Xcelium does not allow additional option specifications with the `deposit` command. If you include any other options, then the simulator generates an error.

deposit Command Options

This section describes the options that you can use with the Tcl `deposit` command.

`-after <time_spec> [<value>
-after <time_spec>...]`

Causes the assignment to occur at a time in the future rather than immediately. The time specified using the *time_spec* argument can be relative or absolute. Relative is the default.

If you do not specify a time, the assignment happens immediately before simulation resumes. If the specified time is the current simulation time, the assignment occurs after simulation resumes but before time advances.

`-relative`

Causes the deposit to occur after the amount of time specified in the *time_spec* argument has passed. This is the default.

`-absolute`

Causes the deposit to occur at the simulation time specified in the *time_spec* argument.

By default, the `deposit` command causes the assignment to occur after the amount of time set by the timing specification has passed. That is, *-relative*.

If you specify multiple *<value> -after <time_spec>* pairs and use *-absolute* to specify an absolute time, the option applies to all pairs. For example, in the following command, the deposits occur at absolute time 10 ns and at time 40 ns:

```
Verilog: xcelium> deposit x 1 -after 10 ns 0 -after 40 ns -  
absolute  
VHDL:    xcelium> deposit :x `1' -after 10 ns `0' -after 40  
ns -absolute
```

However, you cannot use *-absolute* if you set up a repeating deposit cycle using the *-repeat* option. The following command is not valid.

```
Verilog: xcelium> deposit x 1 -after 10 ns 0 -after 40 ns -  
absolute -repeat 100 ns  
VHDL:    xcelium> deposit :x `1' -after 10 ns `0' -after 40  
ns -absolute -repeat 100 ns
```

`-cancel`

The `-cancel` option has the following uses:

- Cancels a repeating deposit after the specified number of time units (`-cancel period`). In this case, the `-cancel` option is used in conjunction with the `-repeat` option.

The *period* argument is a relative time duration. For example, in the following `deposit` command, the value of `sig1` is set to 1 at 10 ns after the current simulation time and then to 0 at 20 ns after the current simulation time:

```
xcelium> deposit sig1 1 -after 10ns 0 -after 20ns -repeat 100ns -cancel 1000 ns
```

These deposits start repeating at 100 ns after the current simulation time, so that the next deposits are at 110 ns and at 120 ns after the current simulation time. The deposit command is canceled 1000 ns after the current simulation time.

- Cancels a repeating deposit immediately (`-cancel 0`).
For example, suppose that you have created a repeating waveform with a command such as the following:

```
xcelium> deposit sig1 1 -after 10ns 0 -after 20ns -repeat 100ns
```

You can cancel the deposit on the object `sig1` at any time with the following command. The deposit is canceled immediately.

```
xcelium> deposit sig1 -cancel 0
```

```
-constraint_mode  
<constraint_name> [=] {0|1}
```

Enables or disables the specified SystemVerilog randomization constraint.

By default, all constraints are enabled. This includes all constraints in the set of constraints that are solved. In some cases, a `randomize()` method call may fail because of the collection of constraints to solve conflict with each other or because they over-constrain a variable. In this case, the simulator generates a warning telling you that the `randomize()` call failed.

The deposit `-constraint_mode` command can help you debug these constraint solver failures. After setting a breakpoint in `randomize()` calls with the `stop -randomize` command, you can then disable specific constraints by setting the constraint mode to 0. No other deposit command options can be included on the command line when using the `-constraint_mode` option. To re-enable the constraint, set the constraint mode to 1. After you have enabled/disabled constraints, you can execute the current `randomize()` call again with the `run -rand_solve` command.

For an example of using these commands, see the `run` command.



No other deposit command options can be included on the command line when using the `-constraint_mode` option.

The deposit `-constraint_mode` command is supported only for class `randomize` calls. The command is not supported for scope `randomize` calls.

A deposit `-constraint_mode` command sets a value that persists for the class instance of the current `randomize()` call. That is, when you continue the simulation using the `run` command, the value remains set and affects other `randomize()` calls of the same class instance. New values also affect static `rand` and `randc` variables for all instances of the class.

```
-generic
```

Deposits the specified value to a VHDL generic. You must use this option if you want to deposit a value to a generic.

Changing the value of a generic may lead to a violation of globally static bounds.

```
-inertial
```

Deposits the value after an inertial delay.

```
-rand_mode <object_name> [=]
{0|1}
```

Enables or disables the specified SystemVerilog randomization (rand or randc) variable.

By default, all variables are enabled, and the `randomize()` method generates random values for all active random variables within an object, subject to the active constraints within the object. In some cases, a `randomize()` call may fail because the collection of constraints to solve conflict with each other or because they over-constrain a variable. In this case, the simulator generates a warning telling you that the `randomize()` call failed.

The deposit `-rand_mode` command can help you debug these failures. After setting a breakpoint in `randomize()` calls with the `stop -randomize` command, you can then disable a specific rand or randc variable by setting the variable mode to 0. When a variable is disabled, it is treated as a state variable and is not randomized by the `randomize()` method. To re-enable the variable, set the mode to 1. After you have enabled/disabled variables, you can execute the current `randomize()` call again with the `run -rand_solve` command.

For an example of using these commands, see the `run` command.

The `object_name` argument must be a simple name that denotes a rand or randc variable in the class of the current `randomize()` call. It cannot be a handle, struct, or array.

No other deposit command options can be included on the command line when using the `-rand_mode` option.

The deposit `-rand_mode` command is supported only for class `randomize` calls. The command is not supported for scope `randomize` calls.

A deposit `-rand_mode` command sets a value that persists for the class instance of the current `randomize()` call. That is, when you continue the simulation using the `run` command, the value remains set and affects other `randomize()` calls of the same class instance. New values also affect static `rand` and `randc` variables for all instances of the class.

```
-release
```

Releases a force if a force is active on the object.

The `-release` option lets you release a force and then apply a deposit with one deposit command.

`-repeat <period>`

Repeats the deposit command.

The *period* argument is the relative time duration at which to start repeating the cycle of deposits. For example, the following `deposit` command sets the value of `sig1` to 1 at 10 time units after the current simulation time and then to 0 at 20 time units after the current simulation time. This cycle repeats at 100 time units after the current simulation time, so that the next deposits are at 110, 120, 210, 220 (and so on) time units after the current simulation time.

```
Verilog: xcelium> deposit sig1 1 -after 10 0 -after 20 -
repeat 100
VHDL:    xcelium> deposit :x '1' -after 10 '0' -after 20 -
repeat 100
```

You can include the `-cancel` option to specify the period at which to stop the deposit cycle.

If there is a repeating deposit on an object, and a second repeating `deposit` command on the same object is issued, the first command is canceled, and the new one comes into effect.

`-scope`
`<path_from_design_top>`

Specifies the scope of the hierarchy from where to start the deposit init flow.

The scope path argument is a Verilog hierarchical path. This path determines the domain of the deposit init flow. Xcelium searches all instances below this hierarchy for matching references defined in the deposit specification file passed to the tool using the `-dfile` command-line option. That is, at simulation time, the deposit command takes `-scope` as input to look for signals defined via the deposit specification. For more information, see [-dfile](#).

The `-scope` option has the following sub-options:

- **-allone**: When depositing on signals/nets specified by the "dfile", all signals/nets are initialized to one.
- **-allzero**: When depositing on signals/nets specified by the "dfile", all signals/nets are initialized to zero.
- **-depth <positive_number | "all">**: Specifies how many scope levels to descend when searching for objects to probe if you have specified a scope. You must specify one of the following arguments:
 - Integer number: Descend the specified number of scopes. For example, `-depth 1` means include only the given scope. `-depth 2` means include the given scope and its subscopes
 - `all`: Include all scopes in the hierarchy below the specified scope. This is the default value.
- **-exclude_module <module_name>**: Specifies the name of the module the user would like to exclude from the deposit flow.

- **-exclude_net** *<net_name | module_name.net_name>*: Specifies the name of the port which needs to be excluded from the deposit flow. The net name specification can be:
 - *net_name|net_name[*|?]*: You can either just specify the name of the net or a regular expression using only "*" and "?" as wildcards.
 - *module_name.net_name*: A user can also specify a port simply by specifying the module name with a port name in case you do not want to exclude only ports.

The "*" and "?" wildcards are not supported

for *<net_name>* and *<module_name.net_name>* in value argument.

- **-exclude_scope** *<scope_path>*: Specifies the hierarchical name of the instance that the user wants to exclude from the deposit processing. In this case, any subscopes of this instance are also ignored.
- **-frc_rel_ports** *<in|out|inout|all>*: Makes the deposits forcefully by doing a force-release operation.

This removes any previous forces on the signal.

- **-if_x**: Checks the value of the port and deposit only if the value of the port is 'X'. In case of a packed or an unpacked port, the deposit is done if any of the bits has an 'X'
- **-if_z**: Checks the value of the port and deposit only if the value of the port is 'Z'. In case of a packed or an unpacked port, the deposit is done if any of the bits has a 'Z'

The sub-options *-if_x* and *-if_z* can be used together.

- **-logfile <file_name>**: Creates separate log files for the `deposit -scope` command. This helps separate these logs from the `xrun/xmsim` logs and makes it easy to read and debug. You can specify the same `<path_to_file>` with the `-logfile` option for multiple `deposit -scope` commands, as, by default, the logs get appended in the log file.
- **-logfile_overwrite**: Overwrites the logs in the log file specified with the `-logfile` option.
- **-logfilter <traverse | deposit | unmarked | skipped>**: Filters out the provided log type from the `deposit -scope` logs. All the log types except the ones provided with the `-logfilter` option are dumped. You can provide multiple values with this option. By default, all the logs are printed if the option is not specified. Below is the mapping of the `-logfilter` values and the log types filtered:
 - **Traverse**: Traversing scope: `<scope_name>`
 - **Deposit**: Depositing on `<signal_name>`
 - **Unmarked**: Element not marked by the dfile `<element_name>`
 - **Skipped**: Skipping deposit as the value does not satisfy `-if_x/-if_z` constrain: `<signal_name>`

- You can specify only one `-logfile` option in the `deposit -scope` command. If you specify multiple `-logfile` options in a single `deposit` command, the tool throws an error (DFMULOPT).
- If these options are specified without the `-scope` option in the `deposit` command, the tool throws an error (DFINVARG).

For more information, see [these examples](#).

- **-random**: Specifies that the simulator generates a random value for the deposit. The following limitations and restriction apply when using this option:
 - It is mandatory to specify `-random` with `-scope`; otherwise, an error results,
 - SV datatypes: only 4-state types are affected by this option.

- **-sense_input**: Enables gate and/or user-defined primitive (UDP) value changes when using deposit.

The Tcl deposit command lets you set the value of an object, like a gate and/or UDP output net. By default, Xcelium overwrites the previous value with the deposited value and preserve this new value until the gate/UDP output is re-evaluated to a value other than the value before the deposit. Use this option to change the default behavior such that the gate/UDP output is updated on the next input value change, regardless of the actual gate/UDP evaluation.

This behavior is synonymous to that of the `xrun/xmelab` command-line option `-deposit_value_change`.

When using this option, you cannot use any other deposit options.

For example, consider the following `top` module, which is instantiated as the dut:

```
module top; reg in1, in2; wire o1, w1, w2;

    assign w1 = in1; assign w2 = in2;

    dut I1(o1, w1, w2);

    initial begin          in1 = 0;  in2 = 1;

        /* At 11ns o1 is deposited with value '1'*/

        #15 in2 = 0;      //15 endendmodule

module dut(o1, in1, in2); output o1; input in1, in2;

    and A1 (o1, in1, in2); specify      (in2 => o1) = (2,
2); endspecifyendmodule
```

The dut has two inputs (`in1`, `in2`) which are passed to an AND gate `A1`, and its output `o1` is the dut output. A path delay `(in2 => o1) = (2, 2)` is specified in the dut. Inputs `in1` and `in2` are initialized with value 0 and 1, respectively. Output `o1` is 0 at 0ns. At 11ns, value 1 is deposited on `o1`. At 15ns, input `in2` changes from 0 to 1. The change in input triggers an evaluation of AND gate `A1`, which outputs value 0. This is also the value of `o1`, before using the deposit command. If you specify the following:

```
xcelium> depoisit top.I1.o1 1
```

The deposited value is 1, which matches the default behavior. However, if you specify `-sense_input` as shown:

```
xcelium> deposit top.I1.o1 -sense_input 1
```

The deposited value changes to 0 at 17ns (with path delay).

- **-verbose**: Displays informational messages that help to debug.

| | |
|---------------------------------|---|
| <code>-transport</code> | Deposits the value after a transport delay. |
| <code>-supply_on</code> | Deposits the voltage and a state of FULL_ON to the supply net specified by <i>net_name</i> . |
| <code>-supply_partial_on</code> | Deposits the voltage and a state of PARTIAL_ON to the supply net specified by <i>net_name</i> . |
| <code>-supply_off</code> | Deposits a state of OFF to the supply net specified by <i>net_name</i> . |

deposit Command Examples

- [Verilog Examples](#)
- [VHDL Examples](#)
- [Vector Size Mismatches](#)
- [Logfile Examples](#)

Verilog Examples

The following command assigns the value `8'h1F` to `test_drink.nickels`. No time for this assignment is specified, so the assignment occurs immediately. The equal sign is optional:

```
xcelium> deposit test_drink.nickels = 8'h1F
```

The following command assigns `25` to `nickels[7:0]` after simulation resumes and 1 time unit has elapsed:

```
xcelium> deposit test_drink.nickels[7:0] = 25 -after 1
```

The following command deposits the value `1` to `nickels[2]`:

```
xcelium> deposit nickels[2] 1
```

The following command uses the `radix#number` syntax to deposit a value of hexadecimal `1a` to `nickels`:

```
xcelium> deposit test_drink.nickels 16#1a
```

The following command uses the `/` character as the hierarchy separator. The binary value `11110000` is deposited to signal `nickels`:

```
xcelium> deposit /test_drink/nickels 2#11110000
```

The following command assigns `25` to `r[8:15]` at simulation time 1 ns:

```
xcelium> deposit r[8:15] = 25 -after 1 ns -absolute
```

The following command sets the value of `x` to the current value of `w`. The assignment occurs at simulation time 10 ns:

```
xcelium> deposit x = #w -after 10 ns -absolute
```

The following command uses both command and value substitution. The object `y` is set to the value returned by the Tcl `expr` command, which evaluates the expression `#r[0] & ~#r[1]` using the current value of `r`:

```
xcelium> deposit y = [expr #r[0] & ~#r[1]]
```

The following command deposits the value `1` to object `my_bit` 10 ns after the current simulation time and then deposits the value `0` 20 ns after the current simulation time:

```
xcelium> deposit my_bit 1 -after 10 ns 0 -after 20 ns
```

The following command deposits the value `4'b0000` to object `bit_vec` 5 ns after the current simulation time and then deposits the value `4'b1111` 15 ns after the current simulation time. This cycle repeats starting at 100 ns after the current simulation time. In other words, `4'b0000` is deposited at time 105, `4'b1111` at time 115 and so on:

```
xcelium> deposit bit_vec 4'b0000 -after 5 ns 4'b1111 -after 15 ns -repeat 100 ns
```

The following command deposits the value `1` to object `my_bit` 10 ns after the current simulation time and then deposits the value `0` to `my_bit` 20 ns after the current simulation time. The cycle repeats starting at 100 ns after the current simulation time. The deposit is canceled at time 1000 ns:

```
xcelium> deposit my_bit = 1 -after 10 ns 0 -after 20 ns -repeat 100 ns -cancel 1000 ns
```

The following command removes an existing force on object `top.myint` and then deposits the value `1`:

```
xcelium> deposit top.myint 1 -release
```

The following command shows the error message that is displayed if you run in regression mode and then try to deposit a value to an object that does not have write access:

```
xcelium> deposit count 4'b0000
xmsim: *E,OBJACC: Object must have write access: board.count.
```

VHDL Examples

The following command deposits the value `1` to object `:t_nickel_out` (`std_logic`). The equal sign is optional:

```
xcelium> deposit :t_nickel_out = '1'
```

The following command deposits the value `1` to object `:top:DISPENSE_tempsig` (`std_logic`):

```
xcelium> deposit :top:DISPENSE_tempsig '1'
```

The following command deposits the value `FF` to object `:t_DIMES` (`std_logic_vector`):

```
xcelium> deposit :t_DIMES {X"FF"}
```

The following command deposits the value `1` to `:t_DIMES(5)`:

```
xcelium> deposit :t_DIMES(5) {'1'}
```

The following command uses the *radix#number* syntax to deposit hexadecimal 1a to t_DIMES:

```
xcelium> deposit :t_DIMES 16#1a
```

The following command uses the / character as the hierarchy separator. test_drink is the top-level entity name. The binary value 11110000 is deposited to signal t_DIMES:

```
xcelium> deposit /test_drink/t_DIMES 2#11110000
```

The following command deposits the value TRUE to object stoppit (boolean):

```
xcelium> deposit stoppit true
```

The following command deposits the value 5 to :G1. This object is a VHDL generic of type integer. You must use the -generic option to deposit to a generic:

```
xcelium> deposit -generic :G1 5
```

The following command deposits the value 10 to object :count (integer):

```
xcelium> deposit :count 10
```

You can deposit values to arrays and records only by depositing to each individual element separately. For example, suppose that you have defined the following record:

```
TYPE record_type is RECORD
    var1: INTEGER;
    var2: STD_LOGIC;
    var3: BOOLEAN;
END RECORD;
Signal record1: record_type;
```

The following commands can be used to deposit values to the elements of this record:

```
xcelium> deposit record1.var1 0
xcelium> deposit record1.var2 '0'
xcelium> deposit record1.var3 true
```

In the following example, an array is defined. This is followed by two deposit commands that deposit values to the elements of the array:

```
TYPE iclk_time_type is ARRAY(1 DOWNT0 0) of time;
signal iclk_period: iclk_time_type;
```

```
xcelium> deposit iclk_period[1] 10 ns
xcelium> deposit iclk_period[0] 5 ns
```

In the following example, a value is deposited to a portion of a multi-dimensional array:

```
subtype DATA_TYPE is bit_vector(0 to 3);
type DATA_ARRAY_TYPE is array(0 to 7) of DATA_TYPE;
```

```
variable DATA_ARRAY : DATA_ARRAY_TYPE;
```

```
xcelium> value :stimulus:DATA_ARRAY
("0000", "0000", "0000", "0000", "0000", "0000", "0010", "0000")
xcelium> deposit {:stimulus:DATA_ARRAY(6) (0 to 2)} B"111"
xcelium> value :stimulus:DATA_ARRAY
("0000", "0000", "0000", "0000", "0000", "0000", "1110", "0000")
```

The following command deposits the value 00 (hex) to object :t_DIMES after 10 ns has elapsed:

```
xcelium> deposit :t_DIMES {X"00"} -after 10 ns -relative
```

The following command deposits the value 1 to object :my_bit 10 ns after the current simulation time and then deposits the value 0 to :my_bit 20 ns after the current simulation time:

```
xcelium> deposit :my_bit = '1' -after 10 ns '0' -after 20 ns
```

The following command deposits the value 0000 to object :bit_vec 5 ns after the current simulation time and then deposits the value 1111 to :bit_vec 15 ns after the current simulation time. This cycle repeats starting at 100 ns after the current simulation time:

```
xcelium> deposit :bit_vec = {"0000"} -after 5 ns {"1111"} -after 15 ns -repeat 100 ns
```

The following command deposits the value 1 to object :my_bit 10 ns after the current simulation time and then deposits the value 0 20 ns after the current simulation time. The cycle repeats starting at 100 ns after the current simulation time. The deposit is canceled 1000 ns after the current simulation time:

```
xcelium> deposit :my_bit = '1' -after 10 ns '0' -after 20 ns -repeat 100 ns -cancel 1000 ns
```

The following command removes an existing force on object :top.myint and then deposits the value 1:

```
xcelium> deposit :top:myint '1' -release
```

The following command shows the error message that is displayed if you run in regression mode and then try to deposit a value to an object that does not have write access:

```
xcelium> deposit :exsum '0'
xmsim: *E,OBJACC: Object must have write access: :exsum.
```

Vector Size Mismatches

If the value being deposited is less than the declared width, a vector size mismatch warning is issued, the right-most bits (LSB) of the value are applied to the right-most (LSB) bits of the vector, and the MSB bits of the object are unchanged. For example:

```
xcelium> value :sig
"UUUUUUUUUUUUUUUUUU"
xcelium> deposit :sig {X"FF"}
xmsim: *W,SEBNDP: Vector size mismatch: "11111111", MSB bits of object unchanged.
xcelium> value :sig
"UUUUUUUUU11111111"
```

If the value being deposited is wider than the declared width, a vector size mismatch warning is issued and the MSB bits of the value are truncated. For example:

```
xcelium> value :sig
"UUUUUUUUUUUUUUUUUU"
xcelium> deposit :sig {X"F0000"}
xmsim: *W,SEBNDT: Vector size mismatch: "11110000000000000000", MSB bits of value truncated.
xcelium> value :sig
"000000000"
```

Logfile Examples

Consider the following example:

dkm_test.v

```
`timescale 1 ns / 1 ns

module dkm_test ;

    wire      a      ;
    wire      b      ;
    reg       c      ;
    reg       d      ;
    reg       rst     ;
    reg       clk     ;

    dkm dkm_i
    (
        a      ,
        b      ,
        c      ,
        d      ,
        rst    ,
        clk
    ) ;

    task reset ;
        begin
            rst <= 1 ;
            @ ( negedge clk ) ;
            rst <= 0 ;
        end
    endtask

    always
        begin
```

```
        clk <= 0 ;
        #10 ;
        clk <= 1 ;
        #10 ;
    end

    initial
    begin
        $monitor (a,, b,, c,, d);
        @ ( negedge clk ) ;
        reset ;
        @ ( negedge clk ) ;
        $finish(0) ;
    end

endmodule

module dkm // drink_machine top
(
    output wire      A      ,
    output wire      B      ,
    input  wire      C      ,
    input  wire      D      ,
    input  wire      rst    ,
    input  wire      clk
) ;
endmodule
```

test.tcl

```
stop -object dkm_test.rst -execute {deposit -scope dkm_test -logfile file.txt -if_z -random
-verbose debug} -continue
run 10 ns
exit
```

dfile (specifies a batch deposit spec file to be used)

```
module *
ports out *
endmodule
```

xrun command:

```
xrun dkm_test.v -access rwc -input test.tcl -dfile dfile -clean
```

In this example, the deposit command executes when `rst` changes. Deposit is done according to the deposit command specified after `-execute`.

Different use-cases:

- Using the `-logfile` option to redirect the logs for the 'deposit -scope' command to the given file

Command:

```
deposit -scope dkm_test -logfile file.txt -if_z -random -verbose debug
```

test.tcl

```
stop -object dkm_test.rst -execute {deposit -scope dkm_test -logfile file.txt -if_z -
random -verbose debug} -continue
run 10 ns
exit
```

Output (file.txt)

```
Traversing scope : dkm_test

Traversing scope : dkm_test.dkm_i
Depositing on dkm_test.dkm_i.A
Depositing on dkm_test.dkm_i.B
Element not marked by dfile C
Element not marked by dfile D
Element not marked by dfile rst
Element not marked by dfile clk
```

- Using the same filename with the `-logfile` option in more than one 'deposit -scope' command

Command:

```
deposit -scope dkm_test -logfile file.txt -if_z -random -verbose debug
```

```
deposit -scope dkm_test -logfile file.txt -if_z -random -verbose debug
```

test.tcl

```
stop -object dkm_test.rst -execute {deposit -scope dkm_test -logfile file.txt -if_z -
random -verbose debug} -continue
run
exit
```

Output (file.txt)

Logs for both commands are appended and redirected to *file.txt*.

```
Traversing scope : dkm_test

Traversing scope : dkm_test.dkm_i
Depositing on dkm_test.dkm_i.A
Depositing on dkm_test.dkm_i.B
Element not marked by dfile C
Element not marked by dfile D
Element not marked by dfile rst
Element not marked by dfile clk

Traversing scope : dkm_test

Traversing scope : dkm_test.dkm_i
Skipping deposit as value does not satisfy -if_x/-if_z constrain : dkm_test.dkm_i.A
Skipping deposit as value does not satisfy -if_x/-if_z constrain : dkm_test.dkm_i.B
Element not marked by dfile C
Element not marked by dfile D
Element not marked by dfile rst
Element not marked by dfile clk
```

- Using the same filename with the `-logfile` option in more than one 'deposit -scope' command but with the `-logfile_overwrite` option

Command:

```
deposit -scope dkm_test -logfile file.txt -logfile_overwrite -if_z -random -verbose
debug
deposit -scope dkm_test -logfile file.txt -logfile_overwrite -if_z -random -verbose
debug
```

test.tcl

```
stop -object dkm_test.rst -execute {deposit -scope dkm_test -logfile file.txt -
logfile_overwrite -if_z -random -verbose debug} -continue
run
exit
```

Output (file.txt)

Logs for the second command are redirected to *file.txt* and overwrite the logs for the first command.

```
Traversing scope : dkm_test
```

```
Traversing scope : dkm_test.dkm_i
```

```
Skipping deposit as value does not satisfy -if_x/-if_z constrain : dkm_test.dkm_i.A
```

```
Skipping deposit as value does not satisfy -if_x/-if_z constrain : dkm_test.dkm_i.B
```

```
Element not marked by dfile C
```

```
Element not marked by dfile D
```

```
Element not marked by dfile rst
```

```
Element not marked by dfile clk
```

- Using the `-logfilter` option to filter out logs specific to 'deposit'

Command:

```
deposit -scope dkm_test -logfilter deposit -logfile file.txt -if_z -random -verbose  
debug
```

test.tcl

```
stop -object dkm_test.rst -execute {deposit -scope dkm_test -logfilter deposit -  
logfile file.txt -if_z -random -verbose debug} -continue  
run 10 ns  
exit
```

Output (file.txt)

All the logs, except the 'deposit' type, are recorded and redirected to *file.txt*.

```
Traversing scope : dkm_test
```

```
Traversing scope : dkm_test.dkm_i
```

```
Element not marked by dfile C
```

```
Element not marked by dfile D
```

```
Element not marked by dfile rst
```

```
Element not marked by dfile clk
```

- Using the `-logfilter` option to filter out 'deposit' and 'unmarked' logs

Command:

```
deposit -scope dkm_test -logfilter deposit -logfilter unmarked -logfile file.txt -if_z -  
random -verbose debug
```

test.tcl

```
stop -object dkm_test.rst -execute {deposit -scope dkm_test -logfilter deposit -  
logfilter unmarked -logfile file.txt -if_z -random -verbose debug} -continue  
run 10 ns  
exit
```

Output (file.txt)

All logs, except for the type 'deposit' and 'unmarked,' are recorded and redirected to *file.txt*.

```
Traversing scope : dkm_test  
  
Traversing scope : dkm_test.dkm_i
```

- Using the `-logfilter` option to filter out 'deposit' and 'unmarked' logs but without using the `-logfile` option

Command:

```
deposit -scope dkm_test -logfilter deposit -logfilter unmarked -if_z -random -verbose  
debug
```

test.tcl

```
stop -object dkm_test.rst -execute {deposit -scope dkm_test -logfilter deposit -  
logfilter unmarked -if_z -random -verbose debug} -continue  
run 10 ns  
exit
```

Output (file.txt)

All logs, except for the type 'deposit' and 'unmarked', are recorded and redirected to *xrun.log/xmsim.log* files.

```
Traversing scope : dkm_test  
  
Traversing scope : dkm_test.dkm_i
```

Related Topic

- [Wildcards Characters in Tcl Commands](#)

Depositing to Verilog Objects

Use the `deposit` command to deposit to ports, signals (wires and registers), and variables in Verilog objects. You can deposit to a subelement of a compressed Verilog vector.

There are three deposit situations depending on the object:

- If the object is a memory or a range of memory elements, the specified value is deposited into each element of the memory or into each element in the specified range.
- If the object is currently forced, you can release the force and then apply a deposit with one `deposit` command by using the `deposit -release` option. If you do not include the `-`

`release` option, the specified value appears on the object after the force is released, unless the release value is overwritten by another assignment in the meantime.

- If the object is a register that is currently forced or assigned, the `deposit` command has no effect.

The value assigned to the object must be a literal. The literal can be generated with Tcl value substitution or command substitution. (See [Value Substitution](#) and [Command Substitution](#) for details on Tcl substitution.)

You can deposit values to vectors in hexadecimal, octal, decimal, or binary formats using the standard Verilog notation. For example:

```
wire [7:0] sig;

xcelium> deposit sig = 8'hff
xcelium> deposit sig = 8'b00000000
xcelium> deposit sig = 8'd10
```

You can also use the following language-independent syntax for specifying the value:

radix # number

The `tcl_relaxed_literal` variable must be set to 1 to enable this functionality.

`deposit sig 2#10111011` For example:

```
deposit sig 8#17
deposit sig 10#13
deposit sig 16#cf
```

Decimal is the default if no radix is specified. For example:

```
xcelium> deposit sig 5
```

Related Topics

- [-deposit_value_change](#)
- [-deposit_sense_input](#)

Depositing to VHDL Objects

Use the `deposit` command to deposit to ports, signals, and variables in VHDL objects if no delay is specified. If a delay is specified, you cannot deposit to variables or to signals with multiple sources. You can deposit to a subelement of a compressed VHDL vector.

For VHDL, the value specified with the `deposit` command must match the type and subtype constraints of

the VHDL object. Integers, reals, physical types, enumeration types, and strings (including `std_logic_vector` and `bit_vector`) are supported. Records and non-character array values are not supported, but objects of these types can be assigned to by issuing commands for each subelement individually.

For VHDL, you can deposit values to vectors by specifying the value for each bit, as follows:

```
signal sig : std_logic_vector(15 downto 0 );
xcelium> deposit :sig = {"111111111111111"}
```

However, an easier, more intuitive way to deposit to a VHDL vector is to use either:

- A bit string literal. The syntax is as follows:

base_specifier "bit_value"

where:

- *base_specifier* can be `b` or `B`, `o` or `O`, or `x` or `X`. If no base is specified, the value is assumed to be binary.
- *bit_value* specifies the bit-string literal in the appropriate base. Underscore characters can be included to improve readability.

Examples:

```
signal sig : std_logic_vector(15 downto 0);
xcelium> deposit sig B"111111111111111"
xcelium> deposit sig b"00000000_11111111"      # Can use underscore characters.
xcelium> deposit sig {"11111111_11111111"}      # No base specifier. Value is binary.
xcelium> deposit sig O"76543"
xcelium> deposit sig X"ffff"
xcelium> deposit sig x"FF_FF"
```

For 9-state logic values (`-`, `U`, `1`, `0`, `Z`, `X`, `L`, `H`, `W`), each value is expanded to four binary bits if the base is hex, and to three binary bits if the base is octal. For example:

```
xcelium> deposit sig {X"FZHA"}
xcelium> value sig
"1111ZZZZHHHH1010"
```

- The following language-independent syntax:

radix# number

The `tcl_relaxed_literal` variable must be set to 1 to enable this functionality.

For example:

```
deposit sig 2#1111000011110000
deposit sig 8#17
```

```
deposit sig 16#abcd
```

For VHDL, base 10 (for example, 10#5) is not supported.

Depositing to SystemC Objects

You can apply the `deposit` command to the following SystemC objects:

- `sc_signal`
- `sc_clock`
- `sc_in`
- `sc_inout`
- `sc_export`
- `sc_register`
- `xmsc_viewable`
- `sc_fifo`
- `sc_fifo_in`
- `sc_fifo_out`
- `sc_mutex`
- `sc_semaphore`
- `tlm_fifo`
- `tlm_req_rsp_channel`

You can deposit values in decimal, hexadecimal, binary, or octal format to the SystemC objects listed above and to the native C/C++ types.

For the templated objects in this list, `deposit` is supported when the object is templated by a C++ data type or by a SystemC built-in data type, or by a user-defined data type that implements the `from_string` virtual function. For `sc_port` and `sc_export` objects, `deposit` is supported if the `sc_object` bound to the port or export supports `deposit`.

Options `-after`, `-cancel`, `-generic`, `-inertial`, `-transport`, `-repeat` are not supported by the Xcelium™ Simulator when specifying SystemC objects.

Related Topic

- [Debugging SystemC Objects and TLM Ports](#)

Depositing to IEEE 1801 Supply Nets

The `deposit` command can also set the voltage and state of an IEEE 1801 supply net in the current scope.

The final state and voltage of the supply net depend on the resolution of all drivers to the supply net. No other options are allowed when using this command with `-supply_on`, `-supply_partial_on`, or `-supply_off`. If any other options are included, then the simulator will generate an error.

Syntax

```
deposit -supply_on|-supply_partial_on <net_name> <voltage>
        [-supply_off <net_name>]
```

Options

| | |
|---------------------------------|---|
| <code>-supply_on</code> | Deposits the voltage and a state of FULL_ON to the supply net specified by <i>net_name</i> . |
| <code>-supply_partial_on</code> | Deposits the voltage and a state of PARTIAL_ON to the supply net specified by <i>net_name</i> . |
| <code>-supply_off</code> | Deposits a state of OFF to the supply net specified by <i>net_name</i> . |

Example

```
xcelium> deposit -supply_on VSS 0.4
```

Related Topic

- [Low-Power Debug Techniques \(IEEE 1801\)](#)

Debugging and Random Stability

Use the `-verbose` debug switch with the `-deposit` command to include information about the scopes that were searched and the signals on which the values were deposited in the log file.

For example:

Traversing the scope:

```
top.dut.I
top.dut.I1
```



```
deposit on top.dut.I.Q
```

All the random values are generated using a checksum which is a factor of the following:

- Full signal hierarchical name
- Any `svseed` argument present to simulation
- The number of deposit call

This way, each run of the design generates the same values unless any of the listed conditions are modified.

The sub-options defined in the table above are supported only when the `-scope` option is specified to the `deposit` command.

describe

The `describe` command displays information about the specified simulation object, including its declaration. You can also use the `describe` command to display low-power simulation information.

The kind of access that has been enabled for simulation objects is shown in the output. For example, the string (`-WC`) is included in the output for objects that have read access but no write or connectivity access. For objects without read access, the output of the `describe` command does not include the object's value.

Additionally, you can use the `scope -describe` command to describe all objects declared within a scope.

You can use wildcard characters in the argument to a `describe` command. However, you cannot use wildcard characters inside escaped names.

You can apply the `describe` command to objects in the SystemC portion of the design that are derived from `sc_object`. For information on using the `describe` command for thread debugging, refer to [Tcl Commands for Debugging Processes](#). For examples of output when the `describe` (and `scope -describe`) command is applied to different data types, SystemC ports, and nested classes, see [describe](#).

describe Command Syntax

```
describe object [object ...] [-verbose]
    -localparam
    -param
    -power
    -psn
```

describe Command Options

This section describes the options that you can use with the Tcl `describe` command.

| | |
|--------------------------|---|
| <code>-verbose</code> | <p>Displays additional properties of the specified object(s) or additional low-power simulation information.</p> |
| <code>-localparam</code> | <p>Displays only the localparams declared in the design.</p> <p>You cannot specify an object with the <code>-localparam</code> option. For example, the following commands generate an error:</p> <pre>xcelium> describe -localparam P xcelium> describe * -localparam</pre> |
| <code>-param</code> | <p>Displays only the parameters declared in the design.</p> <p>You cannot specify an object with the <code>-param</code> option. For example, the following commands generate an error:</p> <pre>xcelium> describe -param P xcelium> describe * -param</pre> |
| <code>-power</code> | <p>Displays low-power simulation information.</p> <p>This option displays information such as:</p> <ul style="list-style-type: none">• The low-power command-line options used in the simulation• The name of the IEEE 1801 or CPF file(s) used in the simulation• The name and scope of each power domain• The name and scope of each isolation and state retention rule• The names of control signals for power shutoff, state retention, and port isolation• Information on power modes and mode transitions (CPF only) <p>By default, this option displays a minimum amount of information, which is the same as that printed to the log file if you specify <code>-lps_verbose 1</code> on the command line for elaboration or simulation. To increase the amount of output information, include the <code>-verbose</code> option. This displays the same amount of information as that printed to the log file if you specify <code>-lps_verbose 5</code> on the command line.</p> |
| <code>-psn</code> | <p>Displays information on supply sets, power switches, and power domains.</p> <p>By default, this option displays the same information as that printed to the log file if you specify <code>-lps_psn_verbose 1</code> on the command line for elaboration or simulation. To increase the amount of output information, include the <code>-verbose</code> option. This displays the same amount of information as that printed to the log file if you specify <code>-lps_psn_verbose 2</code> on the command line.</p> |

describe Command Examples

The section includes the examples of how the `describe` command displays information about simulation objects.

- [Verilog Examples](#)
- [VHDL Examples](#)

Verilog Examples

The following module contains the declarations of objects used in the `describe` command examples. This module contains SystemVerilog constructs and must be compiled with the `-sv` option (`xmvlog -sv`). The `-sv` option is not required if you are running in single-step mode with `xrun`.

The following command displays information about the object `top`:

```
xcelium> describe top
top.....top-level module
```

The following command displays information about the Verilog object `regVector`:

```
xcelium> describe regVector
regVector...variable reg [1:2] = 2'hx
```

The following command displays information about `regVector[1]`:

```
xcelium> describe regVector[1]
regVector[1]...variable reg = 1'hx
```

The following command displays information about two objects: `regMemory` and `wireVector`:

```
xcelium> describe regMemory wireVector
regMemory....variable reg array [3:4] = (1'hx,1'hx)
wireVector...net (wire/tri) logic [1:2] = 2'hz
```

The following command displays information about all objects in the current scope that have names that start with `real`:

```
xcelium> describe real*
realScalar....variable real = 0
realScalarP...parameter real = 0
```

The following command displays information about all objects in the current scope that have names with eight characters and that start with `reg` and end with `dMem`:

```
xcelium> describe reg?dMem
reg2dMem...variable reg [1:2] array [3:4] = (2'hx,2'hx)
reg3dMem...variable reg [1:2] array [3:4] [5:6] = ((2'hx,2'hx), (2'hx,2'hx))
```

The following command shows the output of the `describe` command for an object that does not have read, write, or connectivity access. The output of the command includes the string `(-RWC)` instead of the object's value:

```
xcelium> describe trdrvVector
trdrvVector...net logic [1:2]
```

```
trdrvVector[1] (-RWC)
trdrvVector[2] (-RWC)
```

The following command uses the `-param` option to display information about the parameters in the design:

```
xcelium> describe -param
bitVectorP.....parameter bit [1:2] = 2'h0
logicScalarP.....parameter logic = 1'h0
integerScalarP...parameter integer = 0
int_ScalarP.....parameter int = 0
realScalarP.....parameter real = 0
timeScalarP.....parameter time = 0
enumVectorP.....parameter enum logic [1:2] { aVectorP, bVectorP } = aVectorP
```

The following commands show you the output format of the `describe` command when used to describe different types of objects:

```
xcelium> describe reg*
regScalar...variable reg = 1'hx
regVector...variable reg [1:2] = 2'hx
regMemory...variable reg array [3:4] = (1'hx,1'hx)
reg2dMem....variable reg [1:2] array [3:4] = (2'hx,2'hx)
reg3dMem....variable reg [1:2] array [3:4] [5:6] = ((2'hx,2'hx), (2'hx,2'hx))
xcelium>
xcelium> describe bit*
bitScalar....variable bit = 1'h0
bitVector....variable bit [1:2] = 2'h0
bit2dMem....variable bit [1:2] array [3:4] = (2'h0,2'h0)
bitVectorP...parameter bit [1:2] = 2'h0
xcelium>
xcelium> describe logic*
logicVector....variable logic [1:2] = 2'hx
logicMemory....variable logic array [3:4] = (1'hx,1'hx)
logic3dMem.....variable logic [1:2] array [3:4] [5:6] = ((2'hx,2'hx), (2'hx,2'hx))
logicScalarP...parameter logic = 1'h0
xcelium>
xcelium> describe integer*
integerScalar....variable integer = x
integerVector....variable integer array [3:4] = (x,x)
integer2dMem.....variable integer array [3:4] [5:6] = ((x,x), (x,x))
integerScalarP...parameter integer = 0
xcelium>
xcelium> describe int_*
int_Vector....variable int array [3:4] = (0,0)
int_2dMem.....variable int array [3:4] [5:6] = ((0,0), (0,0))
int_ScalarP...parameter int = 0
xcelium>
```

```
xcelium> describe time*
timeVector....variable time array [3:4] = (x,x)
time2dMem....variable time array [3:4] [5:6] = ((x,x), (x,x))
timeScalarP...parameter time = 0
xcelium>
xcelium> describe enum*
enumScalar....variable enum { aScalar, bScalar } = aScalar
enumVector....variable enum logic [1:2] { aVector, bVector } = 2'hx
enumMemory....variable enum { aMemory, bMemory } array [3:4] = (aMemory,aMemory)
enum3dMem....variable enum logic [1:2] { a3dMem, b3dMem } array [3:4] [5:6] = ((2'hx,2'hx),
(2'hx,2'hx))
enumVectorP...parameter enum logic [1:2] { aVectorP, bVectorP } = aVectorP
enumScalarW...net (wire/tri) enum logic [0:0] { aScalarW, bScalarW } = StX
enumVectorW...net (wire/tri) enum logic [1:2] { aVectorW, bVectorW } = 2'hz
xcelium>
xcelium> describe eType
eType.....typedef enum logic [1:2] { a, b, c, d }
xcelium>
xcelium> describe eType*
eType.....typedef enum logic [1:2] { a, b, c, d }
eTypeScalar...variable eType = 2'hx
eTypeMemory...variable eType array [3:4] = (2'hx,2'hx)
xcelium>
xcelium> describe a b c d
a.....enum constant = 2'h0 (-RWC)
b.....enum constant = 2'h1 (-RWC)
c.....enum constant = 2'h2 (-RWC)
d.....enum constant = 2'h3 (-RWC)
xcelium>
xcelium> describe wire*
wireScalar...net (wire/tri) logic = StX
wireVector...net (wire/tri) logic [1:2] = 2'hz
wireMemory...net (wire/tri) logic array [3:4] = (StX,StX)
wire2dMem....net (wire/tri) logic [1:2] array [3:4] = (2'hz,2'hz)
wire3dMem....net (wire/tri) logic [1:2] array [3:4] [5:6] = ((2'hz,2'hz), (2'hz,2'hz))
xcelium>
xcelium> describe trdrv*
trdrvVector...net logic [1:2]
    trdrvVector[1] (wire/tri) = StX
    trdrvVector[2] (wire/tri) = StX
trdrvMemory...net logic [1:2] array [3:4]
trdrvMemory[3][1] (wire/tri) = StX
    trdrvMemory[3][2] (wire/tri) = StX
    trdrvMemory[4][1] (wire/tri) = StX
    trdrvMemory[4][2] (wire/tri) = StX
xcelium>
```

The following examples show the output of the `describe` command when used on classes and class objects. These examples use the following source code:

```
module test_top;
  class Base;
    integer p1;

    task Base_task(integer i);
      p1 = i;
    endtask

    function Base_function;
      Base_function = p1;
    endfunction

    constraint foo {p1 < 100;}
  endclass

  Base b1 = new;

  class Base2 extends Base;
    integer p2;

    task Base2_task(integer i);
      p1 = 2 * i;
      p2 = 4 * i;
    endtask

    virtual function integer Base2_function();
      Base2_function = this.p1 + 1;
    endfunction
  endclass

  Base2 b2;

  initial begin
    b1.p1 = 2;
  end

endmodule
```

The following command describes the class `Base`:

```
xcelium> describe Base
Base.....class {
    integer p1
}
```

The following command includes the `-verbose` option, which provides more information about the class `Base`:

```
xcelium> describe -verbose Base
Base.....class {
    integer p1
    task Base_task
    function Base_function
    constraint foo
}
```

The following command describes the class `Base2` :

```
xcelium> describe Base2
Base2.....class extends test_top.Base {
    integer p2
    integer p1 (from class Base)
}
```

The following command includes the `-verbose` option, which provides more information about the class `Base2`:

```
xcelium> describe -verbose Base2
Base2.....class extends test_top.Base {
    integer p2
    task Base2_task
    function Base2_function
    integer p1 (from class Base)
    task Base_task (from class Base)
    function Base_function (from class Base)
    constraint foo (from class Base)
}
```

The following command describes a class member in the class `Base2`:

```
xcelium> describe Base2::p2
Base2::p2...variable integer
```

When used on a class object handle, the `describe` command displays the handle's class and value. An object handle's value is the heap data index for the class.

```
xcelium> describe b1
b1.....handle class test_top.Base = null
xcelium> describe b2
b2.....handle class test_top.Base2 = null
```

The following command includes the `-verbose` option to display a detailed description of an object handle:

```
xcelium> describe -verbose b1
b1.....handle class test_top.Base {
```

```
integer p1 = 2
task Base_task
function Base_function
constraint foo
}
```

The following command describes a class handle using the value of the object handle:

```
xcelium> describe [value b1]
test_top.Base@1_1...handle class test_top.Base {
    integer p1 = 2
}
```

VHDL Examples

The following command displays information about the VHDL object `:t_NICKEL_IN`:

```
xcelium> describe :t_NICKEL_IN
t_NICKEL_IN...signal : std_logic = '0'
```

The following command displays information about two VHDL objects: `:t_NICKEL_IN` and `:t_CANS`:

```
xcelium> describe :t_NICKEL_IN :t_CANS
t_NICKEL_IN...signal : std_logic = '0'
t_CANS.....signal : std_logic_vector(7 downto 0) = "11111111"
```

The following command displays information about `:t_CANS(4)`:

```
xcelium> describe :t_CANS(4)
:t_CANS(4)...signal : std_logic = '1'
```

The following command displays information about the object `:top`:

```
xcelium> describe :top
top.....component instantiation
```

The following command displays information about all objects in the current scope that have names that start with `t_DI`:

```
xcelium> describe t_DI*
:t_dime_out...signal : std_logic = '0'
:t_dispense...signal : std_logic = '0'
:t_dimes.....signal : std_logic_vector(7 downto 0) = "11111111"
:t_dime_in....signal : std_logic = '0'
```

In the following example, the `stack -show` command displays the current call stack. The process (process1) is displayed as nest-level 0, the base of the stack. The subprogram `function1` is `:process1[1]`, and the subprogram `function2` is `:process1[2]`.

- The first `describe` command describes the object `tmp5_local`, which is in the current debug scope.

- The second `describe` command describes the object `tmp4` in `:process1[1]`.
- A `scope` command is then executed to set the scope to `:process1[1]`. Because the current debug scope is now `function1`, you can refer to object `tmp4` by simply using its name.
- The last `describe` command describes `var4` in `:process1`.

```
xcelium> stop -subprogram function1
Created stop 1
xcelium> run
0 FS + 0 (stop 1: Subprogram :function1)
./test.vhd:36 tmp4_local := function2 (tmp4);
xcelium> run -step
./test.vhd:29 tmp5_local := tmp5 + 1;
xcelium> stack -show                ;# Display the current call stack
2: Scope: :process1[2] Subprogram:@work.e(a):function2
   File: /usr1/belanger/xm/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 29

1: Scope: :process1[1] Subprogram:@work.e(a):function1
   File: /usr1/belanger/xm/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 36

0: Scope: :process1
   File: /usr1/belanger/xm/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 52
xcelium> scope -show                ;# Display the current debug scope
Directory of scopes at current scope level:

Current scope is (:process1[2])
Highest level modules:
  Top level VHDL design unit:
    entity (e:a)
  VHDL Package:
    STANDARD
    ATTRIBUTES
    std_logic_1164
    TEXTIO
xcelium> describe tmp5_local        ;# Describe tmp5_local in the current debug scope
tmp5_local...variable : INTEGER
xcelium>
xcelium> describe :process1[1]:tmp4 ;# Describe tmp4 in :process1[1]
                                   ;# i.e., function1
:process1[1]:tmp4...constant parameter : INTEGER
xcelium>
xcelium> scope -set :process1[1]    ;# Set scope to function1 (:process1[1])
```

```
xcelium> describe tmp4                                ;# Describe tmp4 in function1
tmp4.....constant parameter : INTEGER
xcelium>
xcelium> describe :process1:var4
:process1:var4....variable : INTEGER = 0
```

Related Topics

- [Wildcards Characters in Tcl Commands](#)
- [Access to Simulation Objects](#)
- [Describing Low-Power Information \(IEEE 1801\)](#)
- [Describing Low-Power Information \(CPF\)](#)

drivers

The Tcl `drivers` command allows you to:

- Display a list of all contributors to the value of the specified object(s) (`-show`).
- List all of the currently active drivers (`-active`).
- Create a run-time driver for a VHDL object and drive a specified value (`-add`).
- Replace an existing run-time driver for a VHDL object and drive a new value (`-replace`).
- Remove a run-time driver for a VHDL object (`-delete`).

You can use the `scope -drivers [scope_name]` command to display the drivers of each object that is declared within a specified scope. See [scope](#) for details on the `scope` command.

The `drivers` command cannot find the drivers of a wire or register unless the object has read and connectivity access. However, even if you have specified access to an object, its drivers may have been collapsed, combined, or optimized away. In this case, the output of the command may indicate that the object has no drivers.

In a low-power simulation, the `drivers` command shows the power drivers, the corresponding IEEE 1801 or CPF file, and the line number of the associated command. To show the active drivers of the specified supply net object, you must specify the `-lps` option.

drivers Command Syntax

```
drivers
    [-show] object_name [object_name ...]
        [-effective]
        [-future]
        [-lps]
        [-novalue]
        [-verbose]
    -active
    -add object_name [=] value
        -delayexecution
    -addcomp object_name [=] value
        -delayexecution
    -delete object_name [object_name ...]
    -replace object_name [=] value
```

drivers Command Options

This section describes the options that you can use with the Tcl `drivers` command.

| | |
|--|---|
| <code>-show object_name [object_name ...]</code> | <p>Displays a list of all contributors to the value of the specified object(s). You must specify at least one object.</p> <p>The <code>-show</code> modifier is optional. The following two commands are equivalent:</p> <pre>xcelium> drivers -show f af xcelium> drivers f af</pre> |
| <code>-effective</code> | <p>Displays contributions to the effective value of the signal. By default, the <code>drivers</code> command displays contributions to the driving value.</p> <p>Only VHDL inout and linkage ports can have different driving and effective values.</p> |
| <code>-future</code> | Displays the transactions that are scheduled on each driver. |
| <code>-lps</code> | Displays the drivers of supply net objects. |
| <code>-novalue</code> | Suppresses the display of the current value of each driver. |

| | |
|---|--|
| <code>-verbose</code> | <p>Displays all of the processes (signal assignment statements), resolution functions, and type conversion functions that contribute to the value of the specified signal.</p> <p>If you do not include the <code>-verbose</code> option, then resolution and type conversion function information is omitted from the output.</p> <p>This option affects VHDL signals only.</p> |
| <code>-active</code> | <p>Lists all of the currently active drivers.</p> |
| <code>-add <i>object_name</i> [=] <i>value</i></code> | <p>Creates a driver for the object and drives the specified value.</p> <p>The following restrictions apply to the object for which a run-time driver is created:</p> <ul style="list-style-type: none">• The object must be a resolved VHDL object (that is, it should have a resolution function associated with it).• The object must be driven by multiple sources in the design.• The object must not have a previously created run-time driver associated with it.• If the object is a VHDL port, it must be of type IN, OUT, or INOUT.• The object must not be a guarded signal.• Objects with composite resolution functions are not supported.• The object must have read, write, and connectivity access.• The value specified with the <code>drivers</code> command must match the type and subtype constraints of the VHDL object. Integers, reals, physical types, enumeration types, and strings (including <code>std_logic_vector</code> and <code>bit_vector</code>) are supported. Records and non-character array values are not supported, but drivers for objects of these types can be created by issuing commands for each subelement individually. <p>The simulator generates error messages if the above conditions are not met, and the driver is not created.</p> <p>Run-time drivers created with <code>drivers -add</code> are not visible in the Trace Signals sidebar.</p> |

`-addcomp object_name [=] value`

Creates a virtual runtime driver for the object and drives the specified value.

The following restrictions apply to the object for which a virtual runtime driver is created:

- The object must be of Record type.
- The object must be a resolved VHDL object (that is, it should have a resolution function associated with it).
- The object must be driven by multiple sources in the design.
- The object must not have a previously created run-time driver associated with it.
- If the object is a VHDL port, it must be of type IN, OUT, or INOUT.
- The object must not be a guarded signal.
- The object must have read, write, and connectivity access.

The simulator generates error messages if the above conditions are not met, and the virtual runtime driver is not created.

Virtual runtime drivers can be created for individual elements of record type object as well for complete record object. Nested records (elements of record are of records type) are also supported.

`-delayexecution`

Drives the value specified with the following commands on the next simulation time:

- `drivers -add`
- `drivers -addcomp`

`-delete object_name [object_name ...]`

Removes the run-time driver (created with a `drivers -add` command) for the specified object(s).

The simulator generates an error if an object specified with the `-delete` option is an object that does not satisfy the conditions for driver creation (see the description of the `-add` option above).

`-replace object_name [=] value`

Replaces the run-time driver (created with a `drivers -add` command) for the specified object and drives the new value.

The simulator generates an error if a run-time driver was not previously created for the specified object.

drivers Command Examples

The following examples illustrate use cases for the `drivers` command:

- [Output for Verilog Signals](#)

- [Output for VHDL Signals](#)
- [Output for Low-Power Simulations](#)

Output for Verilog Signals

- The following command lists the drivers of a signal called `f`:

```
xcelium> drivers f
f.....net (wire/tri) logic = St0
St0 <- (board.counter) assign fifteen = value[0] & value[1] & value[2] & value[3]
```

- The following command lists the drivers of two signals called `f` and `af`:

```
xcelium> drivers f af
f.....net (wire/tri) logic = St0
St0 <- (board.counter) assign fifteen = value[0] & value[1] & value[2] & value[3]
af.....net (wire/tri) logic = St0
St0 <- (board.counter) assign altFifteen = &value
```

- The following command lists the drivers of a signal called `top.under_test.sum`:

```
xcelium> drivers top.under_test.sum
top.under_test.sum...output net (wire/tri) logic [1:0] = 2'h0
2'h0 <- (top.under_test) assign {c_out, sum} = a + b + c_in
```

- The following command lists the drivers of a signal called `board.count`:

```
xcelium> drivers board.count
board.count...net logic [3:0]
count[3] (wire/tri) = St0
St0 <- (board.counter.d) output port 1, bit 0 (./counter.v:19)
count[2] (wire/tri) = St0
St0 <- (board.counter.c) output port 1, bit 0 (./counter.v:18)
count[1] (wire/tri) = St1
St1 <- (board.counter.b) output port 1, bit 0 (./counter.v:17)
count[0] (wire/tri) = St0
St0 <- (board.counter.a) output port 1, bit 0 (./counter.v:16)
```

- The following command lists the drivers of `board.count[2]`:

```
xcelium> drivers board.count[2]
board.count.....net logic [3:0]
count[2] (wire/tri) = St0
St0 <- (board.counter.c) output port 1, bit 0 (./counter.v:18)
```

- The following command shows the error message that the simulator displays if you run in the default "regression" mode (no read, write, or connectivity access to simulation objects) and then use the `drivers` command to find the drivers of an object that does not have read and connectivity

access:

```
xcelium> drivers count
xmsim: *E,OBJACC: Object must have read and connectivity access: board.count.
```

- The following examples illustrate the output of the `drivers` command when the actual driver is from a VHDL model:

```
xcelium> drivers :u1.a
:u1.a.....input net (wire/tri) logic = St1
      St1 <- (:u1) driven by a VHDL model
xcelium> drivers :u1.v.d
:u1.v.d.....input net (wire/tri) logic = St1
      St1 <- (:u1) port 'a' in module 'and2' [File: ./verilog.v],
      driven by a VHDL model
xcelium>
```

This report indicates that the signal `:u1.v.d` is ultimately driven by a port (connected to port `a` of the module `and2`) on a module whose body is an imported VHDL model.

Output for VHDL Signals

The following examples use the VHDL model shown below. A `run` command has been issued after invoking the simulator:

```
library ieee;
use ieee.std_logic_1164.all;
entity e is
end e;
architecture a of e is
    signal s: std_logic;
    function bit_to_std (x: bit) return std_logic is
    begin
        return '0';
    end bit_to_std;
    function std_to_bit (x: std_logic) return bit is
    begin
        return '1';
    end std_to_bit;
begin
    s <= '0' after 1 ns;
    GATE: block
        port (q: inout bit);
        port map (bit_to_std (q) => std_to_bit (s));
    begin
        p: process (q)
        begin
            q <= not q;
        end process;
    end block;
end;
```

- The following command shows the drivers of signal `s`. The `-show` modifier is optional. The string `[verbose report available]` indicates that type conversion functions or resolution functions are part of the hierarchy of drivers. Use the `-verbose` option to display this additional information:

```
% xmsim -tcl -nocopyright e:a
xcelium> run 100 ns
Ran until 100 NS + 0
xcelium> drivers -show s
s.....signal : std_logic = '0'
[verbose report available.....]
'0' <- (:GATE:p) [File: test.vhd]
'0' <- (:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
```

- The following command includes the `-novalue` option, which suppresses the display of the current value of each driver:

```
xcelium> drivers s -novalue
s.....signal : std_logic
```



```
[verbose report available.....]
(:GATE:p) [File: test.vhd]
(:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
```

- The following command includes the `-verbose` option, which causes the output to include resolution function and type conversion function information. This report shows that the port `:GATE:q` is one of the contributing drivers, and that there is a type conversion function `bit_to_std` through which the value of the port is routed before being assigned to the signal `:s`. The report also shows that there is a concurrent signal assignment statement contributing as one of the sources to the resolution function:

```
xcelium> drivers s -verbose
s.....signal : std_logic = '0'
'0' <-[resolution function @ieee.std_logic_1164:resolved()]
<src 1>
    '0' <- (:GATE) [type conversion function
        bit_to_std(<formal>)]
    <formal> connected to port q

    :GATE:q....port : inout BIT = '1'
    '0' <- (:GATE:p) [File: test.vhd, Line: 27]
<src 2>
    '0' <- (:) s <= '0' after 1 ns [File: test.vhd, Line: 21]
```

- The following command shows the drivers `:gate:q`:

```
xcelium> drivers :gate:q
GATE:q.....port : inout BIT = '1'
'0' <- (:GATE:p) [File: test.vhd]
```

- The following command includes the `-effective` option, which displays contributions to the effective value of the signal instead of to the driving value:

```
xcelium> drivers :GATE:q -effective
GATE:q.....port : inout BIT = '1'
[verbose report available.....]
'0' <- (:GATE:p) [File: test.vhd]
'0' <- (:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
```

- The following command includes the `-verbose` option, which helps you to understand where the effective value of 1 in the previous example comes from:

```
xcelium> drivers :GATE:q -effective -verbose
GATE:q.....port : inout BIT = '1'
'1' <- (:GATE) [type conversion function std_to_bit(<actual>)]
<actual> connected to signal s
```

```
:s.....signal : std_logic = '0'
'0' <-[resolution function @ieee.std_logic_1164:resolved()]
    <src 1>
        '0' <- (:GATE) [type conversion function
                        bit_to_std(<formal>)]
        <formal> connected to port q

        :GATE:q....port : inout BIT = '1'
        '0' <- (:GATE:p) [File: test.vhd]
    <src 2>
        '0' <- (:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
```

- The following command includes the `-future` option, which lists the currently scheduled transactions on each driver:

```
% xmsim -tcl -nocopyright e:a
xcelium> run .5 ns
Ran until 500 PS + 0
xcelium> drivers s -future
s.....signal : std_logic = 'U'
[verbose report available.....]
'0' <- (:GATE:p) [File: test.vhd]
    Future Transactions
    None Scheduled

'U' <- (:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
    Future Transactions
    '0' after 1000000.00 fs
```

- The following command creates a driver for the object `:s` and drives the value 1:

```
xcelium> drivers -add :s = '1'
Runtime drivers will not be visible through Signal Flow Browser.
xcelium> drivers :s
:s.....signal : std_logic = 'U'
[verbose report available.....]
'1' <- runtime driver from Tcl or VHPI
'0' <- (:GATE:p) [File: test.vhd]
'U' <- (:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
```

- The following command replaces the existing run-time driver for the object `:s` and drives the new value 0:

```
xcelium> drivers -replace :s = '0'
xcelium> drivers :s
:s.....signal : std_logic = 'U'
[verbose report available.....]
'0' <- runtime driver from Tcl
```

```
'0' <- (:GATE:p) [File: test.vhd]
'U' <- (:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
```

- The following command removes the drivers for the `:s` object:

```
xcelium> drivers -delete :s
```

- Consider a design where you have a vector object `:d`. If you wish to drive the value, then following command creates a driver for an object `:d` and drives the value `00000000000000000000000000000000` with immediate effect:

```
xcelium> drivers -add :d {"00000000000000000000000000000000"}
```

- The following command shows the output of the `drivers` command when the driver is from a Verilog model:

```
xcelium> drivers -effective i1:a
i1:a.....port : in std_logic = '1'
    St1 <- (and2_top.i1) input port 1, bit 0 (./and2.v:9)
xcelium> drivers -effective i1:i1:port1
i1:i1:port1...port : in std_logic = '1'
    St1 <- (and2_top.i1) input port 1, bit 0 (./and2.v:9)
```

The following examples use the VHDL model shown below. A `run` command has been issued after invoking the simulator:

```
library ieee;
use ieee.std_logic_1164.all;
entity e is
end e;
architecture a of e is
signal r, q: std_logic;
begin
    r <= '0' after 1 ns;
    p: process (q)
    begin
        r <= q;
    end process;
end;
```

- The following command adds new driver after next simulation cycle. The new driver gets visible on TCL after next delta.

```
xcelium> drivers -add -delayexecution r '0'
xcelium> drivers -show r
r.....signal : std_logic = 'U'
[verbose report available.....]
```

```
'U' <- (:p) [File: top.vhd, Line: 9]
'0' <- (:) r <= '0' after 1 ns [File: top.vhd, Line: 8]
xcelium> run -delta
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> drivers -show r
r.....signal : std_logic = 'U'
[verbose report available.....]
'0' <- $$FOREIGNPROCESS_0 - runtime driver from TCL or VHPI
'U' <- (:p) [File: top.vhd, Line: 9]
'0' <- (:) r <= '0' after 1 ns [File: top.vhd, Line: 8]
```

The following examples use the VHDL model shown below.

```
library ieee;
use ieee.std_logic_1164.all;
entity e is
end e;

architecture rtl of e is
type rec is record
    e1 : std_logic_vector(1 to 2);
    e2 : integer;
end record;

type t_A0rec is array(natural range <>) of rec;

function f_resolverec (a : t_A0rec) return rec is
variable v1 : rec;
begin
    for i in a'range loop
        v1 := a(i);
        report "Driver " & INTEGER'image(i) severity note;
    end loop;
    return v1;
end ;

subtype res_rec is f_resolverec rec ;

signal s1 : res_rec := ("00", 0) ;
signal d_in : std_logic := '0';
begin
s1 <= ("10", 20) after 5 ns;

pr1 : process(d_in)
begin
    s1 <= (e1=>(others => d_in), e2 => 10);
end process;
end rtl;
```

- The following command displays the existing drivers:

```
xcelium> drivers :s1

:s1.....signal : res_rec = ("10", 20)
[verbose report available.....]
("00", 10) <- (:pr1) [File: test.vhd, Line: 31]
("10", 20) <- (:) s1 <= ("10", 20) after 5 ns [File: test.vhd, Line: 29]
```

- The following command, without the `-delayexecution` option, adds new virtual runtime driver immediately:

```
xcelium> drivers -addcomp s1 {"11", 100}
REPORT/NOTE (time 5 NS) (architecture worklib.e:rtl)
Driver 0
REPORT/NOTE (time 5 NS) (architecture worklib.e:rtl)
Driver 1
REPORT/NOTE (time 5 NS) (architecture worklib.e:rtl)
Driver 2
xcelium> value s1
("11", 100)
```

- The following command uses `-delayexecution` option. It adds new virtual runtime driver after the next simulation cycle:

```
xcelium> drivers -addcomp -delayexecution s1.e2 500
xcelium> run -delta
REPORT/NOTE (time 5 NS) (architecture worklib.e:rtl)
Driver 0
REPORT/NOTE (time 5 NS) (architecture worklib.e:rtl)
Driver 1
REPORT/NOTE (time 5 NS) (architecture worklib.e:rtl)
Driver 2
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> value s1
("UU", 500)
```

Output for a Low-Power Simulation

In a low-power simulation, the `drivers` command shows the power drivers, the corresponding IEEE 1801 or CPF file, and the associated command-line number.

In the following CPF example, there are two defined power domains:

```
create_power_domain -name PDau \
    -instances alu_inst/au1 \
    -shutoff_condition {pcu_inst/pau[2]}

create_power_domain -name PDrf \
    -instances rf_inst \
    -shutoff_condition {pcu_inst/prf[2]}
```

In the following example, the CPF `create_isolation_rule` command is used to illustrate the output of the Tcl `drivers` command:

```
create_isolation_rule -name PDau_iso \  
    -to PDrf \  
    -isolation_condition {pcu_inst/pau[0]} \  
    -isolation_output low  
  
xcelium> # Run until isolation is enabled and power is on  
xcelium> stop -object inst.pcu_inst.pau[0]  
Created stop 1  
xcelium> run  
100 PS + 3 (stop 1: TESTBENCH.inst.pcu_inst.pau[0] = 0)  
xcelium> run  
13400 NS + 4 (stop 1: TESTBENCH.inst.pcu_inst.pau[0] = 1)      # Isolation enabled  
xcelium> run 1 ps  
Ran until 13400001 PS + 0  
xcelium> value inst.alu_inst.aui.a      # Display value of input a  
32'h00000dfe  
xcelium> drivers inst.alu_inst.aui.a    # Display drivers of input a  
inst.alu_inst.aui.a...input net logic [31:0]  
    a[31] (wire/tri) = St0  
        St0 <- low power driver, power domain rule (power domain is on)  
            [File:./nano.cpf, Line:16]  
        St0 <- (TESTBENCH.inst.rf_inst) assign a = rf[ra]  
    ...  
    ...  
    a[0] (wire/tri) = St0  
        St0 <- low power driver, power domain rule (power domain is on)  
            [File:./nano.cpf, Line:16]  
        St0 <- (TESTBENCH.inst.rf_inst) assign a = rf[ra]  
xcelium> value inst.rf_inst.result      # Display value of input result of rf_inst  
32'h00000000      # Value is 0 because of isolation rule.  
xcelium> drivers inst.rf_inst.result    # Display drivers of input result  
inst.rf_inst.result...input net logic [31:0]  
    result[31] (wire/tri) = St0  
        St0 <- low power driver, isolation rule (is enabled)  
            [File:./nano.cpf, Line:60]  
        St0 <- (TESTBENCH.inst) assign result = (opcode == `LOAD) ? dinr : alu  
    ...  
    ...  
    result[0] (wire/tri) = St0  
        St0 <- low power driver, isolation rule (is enabled)  
            [File:./nano.cpf, Line:60]  
        St0 <- (TESTBENCH.inst) assign result = (opcode == `LOAD) ? dinr : alu  
xcelium> # Run until PDau powers down  
xcelium> stop -object inst.pcu_inst.pau[2]  
Created stop 2
```

```
xcelium> run
15800 NS + 4 (stop 2: TESTBENCH.inst.pcu_inst.pau[2] = 1)
xcelium> run 1 ps
Ran until 15800001 PS + 0
xcelium> # Power domain is powered down. Values are unknown.
xcelium> value inst.alu_inst.aui.a
32'hxxxxxxxx
xcelium> drivers inst.alu_inst.aui.a
inst.alu_inst.aui.a...input net logic [31:0]
    a[31] (wire/tri) = StX
        StX <- low power driver, power domain rule (power domain is off)
            [File:./nano.cpf, Line:16]
        St0 <- (TESTBENCH.inst.rf_inst) assign a = rf[ra]
...
...
    a[0] (wire/tri) = StX
        StX <- low power driver, power domain rule (power domain is off)
            [File:./nano.cpf, Line:16]
        St0 <- (TESTBENCH.inst.rf_inst) assign a = rf[ra]
xcelium> value inst.rf_inst.result
32'h00000000 # Value of result is 0 because it is isolated low
xcelium> drivers inst.rf_inst.result
inst.rf_inst.result...input net logic [31:0]
    result[31] (wire/tri) = St0
        St0 <- low power driver, isolation rule (is enabled)
            [File:./nano.cpf, Line:60]
        StX <- (TESTBENCH.inst) assign result = (opcode == `LOAD) ? dinr : alu
...
...
    result[0] (wire/tri) = St0
        St0 <- low power driver, isolation rule (is enabled)
            [File:./nano.cpf, Line:60]
        StX <- (TESTBENCH.inst) assign result = (opcode == `LOAD) ? dinr : alu
xcelium>
```

Related Topics

- [Access to Simulation Objects](#)
- [Displaying the Drivers of an Object \(IEEE 1801\)](#)
- [Displaying the Drivers of an Object \(CPF\)](#)

Report Formats for driver Command

The `drivers` command supports two report formats based on the following:

- [Verilog Signals](#)
- [VHDL Signals](#)

Verilog Signals

The drivers report for Verilog signals is as follows:

```
value <- (scope) verilog_source_line_of_the_driver
```

For example:

```
xcelium> drivers af
af.....net (wire/tri) logic = St0
    St0 <- (board.counter) assign altFifteen = &value
```

Instead of the `verilog_source_line_of_the_driver`, the following is output when the actual driver is from a VHDL model:

```
port 'port_name' in module_name [File:
    path_to_file_containing_module ], driven by a VHDL model.
```

This report indicates that the signal is ultimately driven by a port (connected to `port_name` of the specified module) on a module whose body is an imported VHDL model. The `module_name` corresponds to the module name of the shell being used to import the VHDL model.

VHDL Signals

The drivers report for VHDL signals is as follows:

```
description_of_signal = value

value_contributed_by_driver <- (scope_name) source_description
```

The `source_description` for the various kinds of drivers are as follows:

| | |
|--|---|
| A process | Nothing is generated for the <code>source_description</code> . This implies that a sequential signal assignment statement within a process is the driver. The <code>scope_name</code> is the scope name of the process. |
| Concurrent signal assignment/concurrent procedure call | The <code>source_description</code> is the VHDL source text of the concurrent signal assignment statement or concurrent procedure call that results in a driving value. This concurrent statement is within the scope <code>scope_name</code> . |
| No drivers | If the signal has no drivers, the text <code>No drivers</code> appears verbatim. |

| | |
|---|---|
| A Verilog driver | <p>If the driver is from a Verilog model, the report has the following form:</p> <pre>port 'port_name' in entity(arch) [File: path_to_file_containing_entity], driven by a Verilog model.</pre> <p>This report indicates that the signal is ultimately driven by a port (connected to <i>port_name</i> of the specified entity-architecture pair) on an entity whose body is an imported Verilog model.</p> |
| Driver from a C model | <p>If the driver is from an imported C model, the report has the following form:</p> <pre>port 'port_name' in entity(arch) [File: path_to_file_containing_entity], driven by a C model.</pre> |
| Driver from an OMI model | <p>If the driver is from an imported OMI model, the report has the following form:</p> <pre>port 'port_name' in entity(arch) [File: path_to_shell_file], driven by a OMI model.</pre> |
| Resolution/type conversion function in non-verbose Mode | <p>If you do not use the <code>-verbose</code> option, the text <code>[verbose report available ...]</code> may appear. This indicates that the signal gets its value from a resolution function or a type conversion function. Use <code>-verbose</code> to display more information on the derivation of the signal's value.</p> <p>On the next line of output (indented), a nonverbose driver report is displayed for each signal whose driver contributes to the value of the signal in question.</p> |
| Resolution function | <p>The following text is generated only when the <code>-verbose</code> option is used:</p> <pre>[resolution function function_name()]</pre> <p>This means that the signal is resolved with the named resolution function. A verbose drivers report is displayed (indented) for all inputs to the resolution function.</p> |
| Type conversion on formal of port association | <p>The following text is generated only when the <code>-verbose</code> option is used:</p> <pre>[type conversion function function_name(formal)]</pre> <p>This means that the signal's driving value comes from a type conversion function on a <i>formal</i> in a port association. A verbose drivers report is displayed (indented) for the formal port that is the input to the function.</p> |
| Type conversion on actual of port association | <p>The following text is generated only when the <code>-verbose</code> option is used:</p> <pre>[type conversion function function_name(actual)]</pre> <p>This means that the signal's effective value comes from a type conversion function on an <i>actual</i> in a port association. A verbose drivers report is displayed (indented) for the actual that is the input to the function.</p> |
| Implicit guard signal | <p>The following text is displayed in response to a query on a signal whose value is computed from a <code>GUARD</code> expression:</p> <pre>[implicit guard signal]</pre> |
| Signal attribute | <p>The following is displayed in response to a query on an IN port that ultimately is associated with a signal valued attribute:</p> <pre>[attribute of signal full_path_of_the_signal]</pre> <p>The <i>full_path_of_the_signal</i> corresponds to the complete hierarchical path of the signal whose attribute is the driver.</p> |

| | |
|---|--|
| Constant expression on a port association | The following is displayed when the value of the signal in question is from a constant expression in a port map association: <code>[constant expression associated with port <i>port_name</i>]</code> |
| Composite signals | For a composite signal, a separate report is displayed for each group of subelements that can be uniquely named and that have the same set of drivers. |

Related Topics

- [drivers](#)

Displaying the Drivers of Signals

You can display a list of all of the contributors to the value of a specified signal(s). For example:

```
xcelium> drivers board.count
```

You can use the `scope -drivers [scope_name]` command to display the drivers of each object that is declared within a specified scope.

The `drivers` command cannot find the drivers of a wire or register unless the object has connectivity access. However, even if you have specified access to the object, its drivers may have been collapsed, combined, or optimized away. In this case, the output of the `drivers` command might indicate that the object has no drivers.

Related Topics

- [drivers](#)
- [scope](#)
- [Access to Simulation Objects](#)

dumpsaif

The Tcl `dumpsaif` command generates a Switching Activity Interchange Format (SAIF) file during simulation.

SAIF is an ASCII format developed at Synopsys® designed to help extract toggle rate and state probability data based on switching activity during simulation. The switching activity in this file can then be back annotated into power analysis and optimization tools.

A SAIF file containing switching activity information generated by the simulator is called a *backward* SAIF file. You can use the `dumpsaif` command to generate a backward SAIF file for a Verilog, VHDL, or mixed

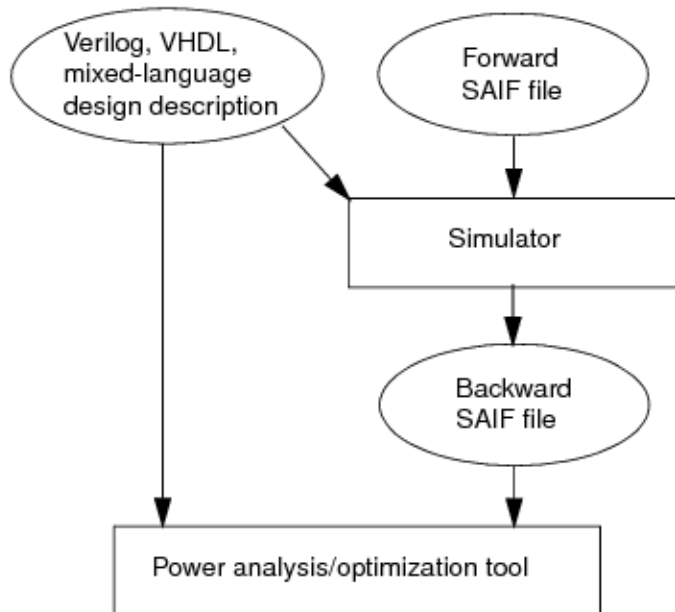
Verilog/VHDL design.

The backward SAIF file generation process also supports the use of SAIF files that provide directives to the simulator about which design elements to trace during the simulation and which switching activity to monitor. These files containing directives to the simulator are usually generated by the power/optimization tool and are called *forward* SAIF files.

Forward and backward SAIF files can be generated for both RTL and gate-level analysis and optimization. To provide the simulator with the necessary directives, there are two types of forward SAIF files:

- **RTL forward SAIF file:** Contains directives that determine which design elements to trace during simulation. These directives list the synthesis-invariant objects in the RTL description (objects that are mapped directly to equivalent design objects in the synthesized gate-level description). An RTL forward SAIF file also provides a mapping from the RTL identifiers of these design objects to their synthesized gate-level identifiers.
- **Library forward SAIF file:** Contains directives for generating state-dependent and path-dependent switching activity during simulation of a gate-level netlist. The directives specify state-dependent and/or path-dependent information that must be gathered for all instances of the cell types.

The following figure illustrates the flow between the simulator and the power analysis and optimization tools using a forward SAIF file.



If you reset the simulation with the `reset` command, the `dumpsaif` command is re-executed, the SAIF probes are re-enabled, and previous SAIF data is lost.

The state of the SAIF database is not saved if you use the `save` command to create a snapshot of the current simulation state. However, the `dumpsaif` command that was used is saved. If you then use the `restart` command to load the saved snapshot, the SAIF probes are re-enabled, and the previous SAIF

data is lost.

dumpsaif Command Syntax

dumpsaif

```
[-compress]  
[-collapse_genblks]  
[-depth integer | to_cells]  
[-divider . | /]  
[-end]  
[-eot]  
[-ewg]  
[-format box | upf]  
[-hierarchy]  
[-input forward_saif_pathname]  
[-inputfile inputfile_pathname]  
  
[-inctoggle]  
[-lib_total_toggle]  
  
[-internal]  
[-interval time_spec]  
[-lps_analyze_output directory_name]  
[-memories]  
[-new_escape_name_format]  
[-output backward_saif_pathname]  
[-overwrite]  
[-scope scope_identifier]  
[-start time_spec]  
[-stop time_spec]  
[-timescale timescale/precision]  
[-verbose]
```

dumpsaif Command Options

This section describes the options that you can use with the Tcl `dumpsaif` command.

`-compress`

Dumps the output in compressed format. This option generates a SAIF file in `.gz` format.

Use the `-compress` option to compress the output file and save on total disk space usage. A compressed SAIF file can also be read by the RTL Compiler.

Example:

```
xcelium> dumsaif -scope : -internal -memories -overwrite -compress
```

By default, the above command generates a file named `xmsim_backward.saif.gz`.

Alternatively, you can specify your own file name for the compressed file using the `-output` option with the `dumsaif` command.

`-collapse_genblks`

Dumps the generate block instances along with the internal module instances, instead of dumping them separately.

By default, when the `dumsaif` command is executed on a design that contains generate blocks, then each generate block is represented as a separate instance. Use the `-collapse_genblks` option to prepend the generate block instances along with the internal module instances.

If there are no generate blocks in the design, then using the `-collapse_genblks` option will not affect anything.

See the `-collapse_genblks` command examples.

`-depth integer |
to_cells`

Dumps the SAIF file for the specified number of levels in the hierarchy or stop at cells.

By default, the `dumsaif` command dumps a SAIF file for all levels of a scope specified with the `-scope` option. If you do not include an argument to the `-scope` option, the top-level design unit is used as the scope.

Use the `-depth` option to dump variables up to a given depth in the hierarchy or use the `to_cells` argument to stop the descent at cells. For example, if a top-level module named `top` instantiates a module `middle`, and `middle` instantiates another module `bottom` (and both modules `middle` and `bottom` are defined in a cell), using:

`-depth 2` prints the variables of `top` and `top.middle`.

`-depth 3` prints the variables of `top`, `top.middle`, and `top.middle.bottom`.

`-depth to_cells` prints the variables of `top`.

The `-depth` option is currently supported only for backward SAIF files.

`-divider . | /`

Specifies the hierarchy divider for saif files.

The supported values are "." and "/".

- If the `-divider` option is not specified, the default saif divider "." is used.
- The `-divider` option overrides the divider specified in the forward saif file.

Usage:

```
dumlsaif -scope -divider /
```

`-end`

Ends all SAIF probing activity.

When a `dumlsaif -end` command is issued, the SAIF database is closed. Any simulation activity after a `dumlsaif -end` command is not reflected in the backward SAIF database.

You cannot restart the SAIF dumping to the same database.

`-eot`

Reports only transitions that occur after a timing interval.

By default, the SAIF file generated by the `dumlsaif` command shows transitions after every delta and timing delay. This does not match the SAIF file that is generated by converting a VCD file in a competitor simulator because that simulator considers transitions only at the end of each timing interval.

The `-eot` option forces the simulator to consider the values of signals after each timing interval, rather than after each delta interval. Only those transitions that occur after a timing interval are reported in the SAIF file. If the value of a signal is being changed multiple times after each delta delay, the SAIF file contains transition values only after each timing delay, rather than after both timing and delta delays.

`-ewg`

Ignores glitches for recording SAIF data.

This option enables SAIF dumping while ignoring glitches on ports.

Until now, the tool used to increment toggle counts even for glitches occurring on the net/ports. Now, glitches are ignored for the SAIF port timing and toggle attributes. The SAIF counters, thus recorded, are expected to provide better power computation compared to the (default) event-based backward SAIF generated by Xcelium, or the backward SAIF generated with the `'-eot'` option.

Usage:

```
dumpsaif -input fwd.saif -out back.saif -ewg
```

The `-ewg` option is not supported along with the existing `-eot` option.

When using the `-inctoggle` option along with `-ewg` option:

All the glitches for a signal within the simulation time are ignored and only the final value is considered while calculating the Inertial Glitch (IG).

IG count for a signal is increased by 1 when the value (ignoring all the glitches within the simulation time) of the signal changes from:

```
0 → X → 0
or
0 → Z → 0
or
1 → X → 1
or
1 → Z → 1
```

`-format box | upf`

Specifies the format for the output SAIF file.

You can specify one of the following arguments with this option:

- `box`

Dumps the SAIF output in an alternative format.

Consider the following example:

```
type bus_data is
  record
    a : std_logic_vector(1 downto 0);
    b : std_logic_vector(2 downto 0);
    c : std_logic;
  end record;
type bus_arr is array (0 to 3) of bus_data; --first define the
type of array.
signal ctrl_sig : bus_arr
```

By default, the simulator uses the above information to output a SAIF file with the following format:

```
(ctrl_sig\[0\]\.a\[1\]
      (T0 0) (T1 0) (TX 100000000)
      (TZ 0) (TB 0) (TC 0)
```



```
)
```

If you want to output a SAIF file using the alternative format, then you can specify this option using `box` as the argument. When specified, the format changes as shown:

```
(ctrl_sig\[0\]\[a\]\[1\]
      (T0 0) (T1 0) (TX 100000000)
      (TZ 0) (TB 0) (TC 0)
)
```

- `upf`

This argument dumps the escape names in the upf style names by applying the following rules:

- There is no leading `'\'` for escape names.
- There is no terminating white space for escape names.
- The following special characters are escaped if they are part of the escape names:

| Type | Character |
|---|-----------|
| Logic hierarchy delimiter | / |
| Escape character | \ |
| Bus delimiter, index operator, or within a <code>regex</code> | [] |
| Range separator (for bus ranges) | : |
| Record field delimiter | . |

The option `-format upf` is not supported with the `-new_escape_name_format` option, since both these are format-modifying options.

The following default output changes when the `-format upf` option is specified:

Default output:

```
(INSTANCE "module_sram" \WIDTH128[0] \.sram
```

upf name style output:

```
(INSTANCE "module_sram" WIDTH128\[0\]\.sram
```

`-hierarchy`

Enables a hierarchical dump of the backward SAIF file.

See [Rules for Library Forward Files](#) for important information on using `-hierarchy` with the `-input` option.

`-input`
`forward_saif_pathname`

Specifies the path to the forward SAIF file.

You can use multiple `-input` options to specify multiple input files. This multiple `-input` option can be used to specify both RTL and Library forward SAIF files in the same session. For example:


```
xcelium> dumsaif -input rtl.saif -input lib.saif - overwrite
```

`-inputfile`
`inputfile_pathname`

Specifies the path to the file containing list of forward saif files.

You can use multiple `-inputfile` options to specify multiple input files. This multiple `-inputfile` option can also be used along with `-input` option in same session. For example:

```
xcelium> dumsaif -inputfile inputfile1.txt -inputfile
inputfile2.txt -input fwd.fsaif - overwrite
```

 You cannot use the `-input/-inputfile` and `-scope` options together on the same command line for an RTL forward SAIF file. If `-input/-inputfile` and `-scope` are both used in this case, then a warning is issued for the specified forward SAIF file, and the `-input/-inputfile` option provides information for generating the backward SAIF file. However, the `-scope` option, when used with `-input/-inputfile` for a library forward SAIF file, is a permitted combination.

The following rules apply to the use of the `-input/-inputfile` option with *library forward SAIF* files:

- If both the `-input/-inputfile` and `-hierarchy` options are specified, the `-hierarchy` option is ignored and dumping reverts to the standard `-input/-inputfile` dumping behavior.
- If the `-input/-inputfile`, `-hierarchy`, and `-internal` options are all specified, both the `-hierarchy` and `-internal` options are ignored and dumping reverts to the standard `-input/-inputfile` dumping behavior.
- If the `-input/-inputfile`, `-scope`, and `-hierarchy` options are all specified, dumping starts from the specified *scope* and each instance is dumped in a hierarchical format, ending when the cell specified in the *input/inputfile* file is reached. No net information is dumped in this case.
- If the `-input/-inputfile`, `-scope`, `-hierarchy`, and `-internal` options are all specified, then the internal nets for each hierarchy in the path to the cell defined in the library file are also dumped to the SAIF file.

`-inctoggle`

Dumps the toggle count and inertial glitch count for X and Z transitions.

By default, the simulator ignores X and Z transitions when the Tcl `dumsaif` command is specified. Use this option to capture these transitions in a SAIF file and use the information in other tools, like RTL Compiler, which reads in data for power analysis.

The following transitions are captured when the `-inctoggle` option is specified:

| Toggle Count | Inertial Glitch |
|--------------|-----------------|
| 1 > X > 0 | 1 > X > 1 |
| 0 > X > 1 | 0 > X > 0 |
| 1 > Z > 0 | 1 > Z > 1 |
| 0 > Z > 1 | 0 > Z > 0 |

Each transition is counted by the simulator as either 1 Toggle Count (TC) or 1 Intertial Glitch (IG).

To generate the toggle count and inertial glitch count for a library forward SAIF file, specify this option together with `-lib_total_toggle`.

For example, a design `dut.v` includes the module `top` below:

```
module top;
  wire s;
  reg a = 1'bx;
  assign s = a;
  initial
  begin
    #1 a = 1'b0;
    #1 a = 1'b1;
    #1 a = 1'bx;
    #1 a = 1'b1;
    #1 a = 1'b0;
    #1 a = 1'bx;
    #1 a = 1'b1;
    #1 a = 1'bz;
    #1 a = 1'b1;
    #1 a = 1'bz;
    #1 a = 1'b0;
  end
endmodule
```

You can compile and elaborate this design by specifying the following commands:

```
% ncvlog dut.v
% ncelab -access +rw top
```

Use the next command to invoke the `ncsim>` prompt:

```
% ncsim -tcl top
```

From the `ncsim>` prompt, specify the following Tcl commands:

```
ncsim> dumphsaif -scope top -internal -memories -overwrite -inctoggle
ncsim> run 12 ns
ncsim> exit
```

A new SAIF file `ncsim_backward.saif` is created with the output shown:

```
(SAIFFILE
(SAIFVERSION "2.0")
(DIRECTION "backward")
(DESIGN )
(DATE "Apr 12 2016 13:36:06 EDT")
(VENDOR "Cadence Design Systems, Inc")
(PROGRAM_NAME "NC Simulation Engine")
(VERSION "TOOL: ncsim 15.20-s003")
(DIVIDER . )
(TIMESCALE 1 fs )
(DURATION 12000000)
(INSTANCE "top" top
  (NET
    (s
      (T0 3000000) (T1 4000000) (TX 3000000)
      (TZ 2000000) (TB 0) (TC 4) (IG 2)
    )
    (a
      (T0 3000000) (T1 4000000) (TX 3000000)
      (TZ 2000000) (TB 0) (TC 4) (IG 2)
    )
  )
)
```

```
)
)
)
```

`-lib_total_toggle` Dumps the total toggle count along with condition toggle count in library forward SAIF file.

This option can only be used with `-input` option and is ignored otherwise. A warning is issued when `-lib_total_toggle` is used without the `-input` option.

Consider the following example where **TC4** indicates the total toggle count:

```
(Q\[0\] (T0 49000000) (T1 40000000) (TC 4) ( COND (CLK)
(RISE) (TC 2) COND (CLK) (FALL) (TC 2) )
```

`-internal` Enables dumping of internal wires and signals in the output SAIF file.

By default, only ports are probed. Use the `-internal` option to include internal nets and signals in the probe. For information on using `-internal` with the `-input` option, see [Rules for Library Forward Files](#).

`-interval time_spec` Specifies a particular period of time for capturing SAIF data during simulation.

The `time_spec` argument can be a positive integer followed by a time unit. If a time unit is not specified, the default simulation time unit is used. When the simulation time matches the indicated time interval, SAIF data is dumped and a new file is created automatically for the next time interval. For example, to create a SAIF file every 100ns use the following command:

```
xcelium> dumsaif -scope top -interval 100ns
```

You can use this option together with `-start` or `-stop`, or use it with both options to create multiple SAIF files within a particular time window. For example, to generate a new SAIF file every 80ns for the scope `top` within the simulation time window specified from 200 ns to 600 ns use the command as shown:

```
xcelium> dumsaif -scope top -start 200 ns -stop 600 ns -interval 80
ns
```

`-memories` Generates switching and toggle information for unpacked registers and unpacked nets.

For example:

```
logic [3:0] unpacked_array [5:0];
```

Use the `-memories` option to dump toggle information for:

- unpacked array
- unpacked struct
- arrays of MDV
- unpacked unions

`-lps_analyze_output`
`directory_name`

Dumps the SAIF files generated with the `-lps_analyze` option in the specified directory. This is used to change the default behavior of the tool where the output files are dumped in the `pwr_sa` directory under the current working directory.

- If the specified directory already exists, it will be overwritten.
- If the switch is specified with designs that are not elaborated with the `-lps_analyze` option, the tool prints a warning message.
- If the design is elaborated with the `-lps_analyze` option but this switch is not specified with the `dumpsaif` command, the generated SAIF files are dumped in the `pwr_sa` directory, as per the default behavior of the tool.

The parent directory(s) specified with `-lps_analyze_output` must exist. Otherwise, the tool throws an error (**BKWSAF**) and the SAIF file(s) are not dumped.

For example:

```
% dumpsaif -lps_analyze_output dir1/dir2/dir3/pwr_sa
```

Here, for the SAIF file to be dumped, the `dir1/dir2/dir3` directory must exist.

You can also run multiple `dumpsaif` commands in the same simulation run and dump the outputs in different directories.

For example:

```
dumpsaif -scope ... -lps_analyze_output my_pwr_sa1
run 100 ns
dumpsaif -end
dumpsaif -scope ... -lps_analyze_output my_pwr_sa2
run 100 ns
dumpsaif -end
```

In this case, SAIF files from `0ns` to `100ns` are dumped in the `my_pwr_sa1` directory and `100ns` to `200ns` are dumped in the `my_pwr_sa2` directory.

`-`
`new_escape_name_form`
`at`

Applies the following changes for all escape names regardless of whether it is a port, net, or instance:

- The `'\'` at the beginning of the escape names is removed.
- A `'\'` is added before every special character in the name.
 Special characters are characters other than the alphabet, numbers, space, and underscore.

For Verilog, `$` is a valid character for simple identifiers, but still, it is escaped.

See the [-new_escape_name_format](#) command examples.

| | |
|---|--|
| <code>-output</code> <code>backward_saif_pathname</code> | <p>Specifies the path to the backward SAIF file.</p> <p>By default, the backward SAIF file is generated in the current working directory and is called <code>xmsim_backward.saif</code>. Use the <code>-output</code> option to override the default. For example:</p> <pre>xcelium> dumsaif -input rtlfwd.saif -output rtlbkwd.saif xcelium> dumsaif -scope : -output ./saifout/xmsim_backward.saif</pre> <p>If a file with the specified name already exists, a warning is generated to inform you that the existing file is not overwritten unless the <code>-overwrite</code> option is used.</p> <p>If you do not include an argument to the <code>-output</code> option, a warning is generated and a backward SAIF file with the default name (<code>xmsim_backward.saif</code>) is generated in the current working directory.</p> |
| <code>-overwrite</code> | <p>Enables overwriting of the generated backward SAIF file. By default, the <code>dumsaif</code> command does not overwrite an existing backward SAIF file.</p> |
| <code>-pause</code> | <p>Stops processing value changes to probed signals. The <code>-pause</code> option skips updating the simple timing (T1/T0) and toggle attributes (TC) of the probed signals till <code>-resume</code> option is used. No changes are made to the probes that were set up when SAIF dumping had been enabled and started.</p> <p>For example:</p> <pre>\$ dumsaif -scope top -internal -memories /** Begin SAIF recording */ \$ run 1ns /** SAIF recording is on */ \$ dumsaif -pause /** SAIF recording is paused*/ \$ run 4 ns /** SAIF recording is not taking place */ \$ dumsaif -resume /** SAIF recording is resumed again */ \$ run /** SAIF recording is on */ \$ dumsaif -end /** Backward SAIF is written out */ \$ exit</pre> |
| <code>-resume</code> | <p>Starts processing the value changes of probed signals and accumulating the timing/toggle attributes, over the earlier saved attributes.</p> |
| <code>-scope</code> <code>scope_identifier</code> | <p>Specifies the scope of the hierarchy for which the backward SAIF needs to be dumped.</p> <p>The <code>scope_identifier</code> argument is a Verilog or VHDL hierarchical path. The specified path determines the domain of the probing activity. All instances and cells below this hierarchy are probed for the SAIF information.</p> <p>If you do not include an argument to the <code>-scope</code> option, a warning is generated and the top-level design unit is used as the scope.</p> <p>For information on using <code>-scope</code> with the <code>-input</code> option, see Rules for Library Forward Files.</p> |

| | |
|--------------------------------------|---|
| <code>-start <i>time_spec</i></code> | <p>Specifies the time to start collecting SAIF data during simulation.</p> <p>The <i>time_spec</i> argument can be a positive integer followed by a time unit. If a time unit is not specified, the default simulation time unit is used. When the simulation time matches the indicated start time, the tool begins collecting data for one or more SAIF files.</p> <p>For example:</p> <pre>xcelium> dumsaif -start 20ns</pre> <p>You can use this option together with <code>-stop</code> and <code>-interval</code>.</p> |
| <code>-stop <i>time_spec</i></code> | <p>Specifies the time to stop collecting SAIF data during simulation.</p> <p>The <i>time_spec</i> argument can be a positive integer followed by a time unit. If a time unit is not specified, the default simulation time unit is used. When the simulation time matches the indicated stop time, the tool stops collecting data for one or more SAIF files.</p> <p>For example:</p> <pre>xcelium> dumsaif -stop 1200ns</pre> <p>You can use this option together with the <code>-start</code> and <code>-interval</code>.</p> <p>This option is part of the RTL power estimation flow. To use this option, you must first elaborate your design with the <code>-lps_analyze</code> option.</p> |

`-timescale`
`timescale/precision`

Controls the timescale in the saif files. The default value is 1fs/1fs. The scaling is done with respect to the default timescale value which is “fs”.

This option, `dumpsaiif -timescale`, is not affected by the timescale at which the simulation is running.

The argument `timescale/precision` is a combination of a scaling factor and a time unit, for example:

`-timescale 1ns/1 ps`

- Valid values of the scale factor are 1, 10, and 100.
- Valid values of the time unit are fs, ps, ns, us, ms, and s.

Examples

- Let's assume that TX = 1000000
 - If timescale is 1ns/1fs
 Here, since the timescale is set to 1ns, so the unit of TX is converted to ns. 1ns = 1000000fs, therefore, TX = 1ns
 - If timescale is 1ps/1fs
 Here, since the timescale is set to 1ps, so the unit of TX is converted to ps. 1ps = 1000fs, therefore, TX = 1000ps
- We may get fractional timings after scaling:
 If TX = 15 ns
`dumpsaiif -timescale 10ns/1ns`
 Then TX after scaling = 1.5
- We can control the number of digits after the decimal through the precision value:
 If TX = 15 ns
`dumpsaiif -timescale 10ns/1ns`
 Then TX after scaling = 1.5

 If TX = 15 ns
`dumpsaiif -timescale 10ns/10ns`
 Then TX after scaling = 1
- All the trailing 0s after the decimal point are removed

`-verbose`

Displays informational messages during the generation of the backward SAIF file. These messages include information on:

- The scope being probed, or the forward SAIF file being used.
- The name of the output file.
- The instances being probed (if `-scope` or an RTL forward SAIF file is used), or library cells (if a Library forward SAIF file is used).
- When the information collected by the data structures is written to the output backward file.

dumpsaif Command Limitations

The following limitations exist on the dumping of SAIF databases:

- Multiple SAIF files cannot be dumped at the same time. Only one `dumpsaif` command is active during a given simulation timeframe. Another `dumpsaif` command can be issued only if the first has been terminated with `dumpsaif -end`.
- Real and other complex data types, such as access types, file types, and physical data types are not supported. Individual bits of vectors can be specified in the forward file.
- Arrays of unpacked struct and unpacked union are not supported.
- Internal signals can be probed for SAIF activity. However, variables within a process are dropped from the list of elements being probed.
- Any state-dependent expression with a component value `x` or `z` is treated as `COND_DEFAULT`.
- The `-pause` and `-resume` options do not co-work along with the following low-power options:
 - `-start`
 - `-stop`
 - `-interval`

dumpsaif Command Examples

On the `dumpsaif` command, you must use either the `-scope` option to specify the scope of the hierarchy for which the backward SAIF needs to be dumped, or the `-input` option to specify a forward SAIF file.

The following command uses the `-scope` option. No argument is provided, and the top-level design unit is used as the scope. A backward SAIF file called `xmsim_backward.saif` is generated in the current working directory:

```
xcelium> dumpsaif -scope
```

In the following command, the `-scope` option specifies the Verilog scope `top.dut`. All instances and cells below this hierarchy are probed for the SAIF information:

```
xcelium> dumpsaif -scope top.dut
```

In the following command, the `-scope` option specifies the VHDL scope `:dut`:

```
xcelium> dumpsaif -scope :dut
```

The following command includes the `-input` option, which specifies a forward SAIF file called `fwd.saif`:

```
xcelium> dumpsaif -input fwd.saif
```

The following command includes the `-output` option. A backward SAIF file called `bkwd.saif` is generated

in the current working directory:

```
xcelium> dumpsaif -input fwd.saif -output bkwd.saif
```

In the following sequence of commands, the simulation is run for 3000 ns. A `dumpsaif -end` command is then issued to close the SAIF database and end the SAIF dumping:

```
xcelium> dumpsaif -input fwd.saif -output bkwd.saif  
xcelium> run 3000 ns  
xcelium> dumpsaif -end  
xcelium> run
```

The following example uses the `-memories` option to dump toggle information for a multi-dimensional unpacked array of `reg` type, unpacked structs and unpacked unions:

```
module top();  
  
// Packed Arrays 1-D  
    reg [1:0] pa;  
  
// Unpacked Arrays 2-D  
    reg c [1:0][1:0];  
  
// Unpacked Struct  
    struct {reg r; reg [1:0] arr;} st;  
  
    reg clk;  
  
    always @(clk)  
        begin  
            c[0][0] = clk;  
            c[0][1] = ~clk;  
            c[1][0] = clk;  
            c[1][1] = ~clk;  
            pa[0] = clk;  
            pa[1] = ~clk;  
            st.arr[0] = clk;  
            st.arr[1] = ~clk;  
            st.r = clk;  
        end  
  
    initial  
        begin  
            clk = 1'b0;  
            forever #20 clk = ~clk;  
        end  
endmodule
```

```
xcelium> dumpsaif -scope top -internal -memories
```

Make sure to elaborate with read permission (-access +r)

The SAIF file generated for this Verilog code is as follows:

```
(SAIFFILE
(SAIFVERSION "2.0")
(DIRECTION "backward")
(DESIGN )
(DATE "Sep 25 2017 12:36:23 IST")
(VENDOR "Cadence Design Systems, Inc")
(PROGRAM_NAME "NC Simulation Engine")
(VERSION "TOOL:          xmsim    17.10-a001-20170922")
(DIVIDER . )
(TIMESCALE 1 fs )
(DURATION 1000000000)
(INSTANCE "top" top
  (NET
    (pa\[1\]
      (T0 400000000) (T1 600000000) (TX 0)
      (TZ 0) (TB 0) (TC 4)
    )
    (pa\[0\]
      (T0 600000000) (T1 400000000) (TX 0)
      (TZ 0) (TB 0) (TC 4)
    )
    (c\[1\]\[1\]
      (T0 400000000) (T1 600000000) (TX 0)
      (TZ 0) (TB 0) (TC 4)
    )
    (c\[1\]\[0\]
      (T0 600000000) (T1 400000000) (TX 0)
      (TZ 0) (TB 0) (TC 4)
    )
    (c\[0\]\[1\]
      (T0 400000000) (T1 600000000) (TX 0)
      (TZ 0) (TB 0) (TC 4)
    )
    (c\[0\]\[0\]
      (T0 600000000) (T1 400000000) (TX 0)
      (TZ 0) (TB 0) (TC 4)
    )
    (st\.r
      (T0 600000000) (T1 400000000) (TX 0)
      (TZ 0) (TB 0) (TC 4)
    )
    (st\.arr\[1\]
      (T0 400000000) (T1 600000000) (TX 0)
      (TZ 0) (TB 0) (TC 4)
    )
  )
)
```

```
        (T0 40000000) (T1 60000000) (TX 0)
        (TZ 0) (TB 0) (TC 4)
    )
    (st\..arr\[0\]
        (T0 60000000) (T1 40000000) (TX 0)
        (TZ 0) (TB 0) (TC 4)
    )
    (clk
        (T0 60000000) (T1 40000000) (TX 0)
        (TZ 0) (TB 0) (TC 4)
    )
)
)
)
```

Examples Using the Library Forward File Rules

The following examples illustrate using the `-input`, `-hierarchy`, `-internal`, and `-scope` options with *library forward* files:

- Using the `-input` and the `-hierarchy` options.

In this case, `-hierarchy` is ignored and the design elements traced during simulation and the switching activity monitored are defined only by the `fwd.saif` forward input file.

```
xcelium> dumsaif -input fwd.saif -hierarchy
```

- Using the `-input`, `-hierarchy`, and `-internal` options.

In this case, both the `-hierarchy` and the `-internal` options are ignored, and the design elements traced during simulation and the switching activity monitored are defined only by the `fwd.saif` forward input file.

```
xcelium> dumsaif -input fwd.saif -hierarchy -internal
```

- Using the `-input`, `-scope`, and `-hierarchy` options.

In this case, dumping starts from the hierarchical path specified by the `-scope` argument. Each instance is dumped in a hierarchical format and all instances and cells below this hierarchy are probed for the SAIF information.

```
xcelium> dumsaif -input fwd.saif -scope top.dut -hierarchy
```

- Using the `-input`, `-scope`, `-hierarchy`, and `-internal` options.

In this case, dumping starts from the hierarchical path specified by the `-scope` argument, and the `-internal` and `-hierarchy` options perform a hierarchical dump of the internal nets and signals to the backward SAIF file.

```
xcelium> dumsaif -input fwd.saif -scope -hierarchy top.dut -internal
```

Examples Using the `-collapse_genblks` Command Option

The following examples show the output generated by `-dumpsaiif` command when the `-collapse_genblks` option is used.

File: test.sv

```
module top();
mid #(1,1) mid_inst();
endmodule

module mid();
parameter p = 0 ;
parameter p1 = 0;

//===== GEN IF =====//
generate if (p ==1) begin : GEN_IF
            reg r_gen_if;
            bot bot_inst_if();
        end

        else begin
            bot bot_inst_else();
        end
endgenerate
endmodule

module bot();
endmodule
```

Command:

```
xcelium> dumpsaiif -scope top -internal -memories -overwrite -output saif.log -hier -collapse_genblks
```

Use the following command if running the command from *xrun*:

```
% xrun -REDUCE_MESSAGES -NOCOPYRIGHT -clean -access +rwc -input run.tcl -sv -v93 -status test.sv -log_xmsim xmsim.log
```

Output:

- **When using `-hier` switch:**

```
(INSTANCE "top" top
(INSTANCE "mid" mid_inst
(NET
```

```
(GEN_IF\.r_gen_if
(T0 0) (T1 0) (TX 0)
(TZ 0) (TB 0) (TC 0)
)
)
(INSTANCE "bot" GEN_IF\.bot_inst_if
)
)
)
```

- **Without using `-hier` switch:**

```
(INSTANCE "top" top
)
(INSTANCE "mid" top.mid_inst
)
(INSTANCE top.mid_inst.GEN_IF
(NET
(r_gen_if
(T0 0) (T1 0) (TX 0)
(TZ 0) (TB 0) (TC 0)
)
)
)
(INSTANCE "bot" top.mid_inst.GEN_IF.bot_inst_if
)
```

File: test.sv

```
module top();
mid #(1,1) mid_inst();
endmodule

module mid();
parameter p = 0 ;
parameter p1 = 0;

//===== GEN CASE =====//
generate case (p1)
    //reg r_gen_case;
    0: bot bot_inst_case_0();
    1: bot bot_inst_case_1();
endcase
endgenerate
endmodule

module bot();
endmodule
```

Command:

`dumpsaiif -scope top -internal -memories -overwrite -output saif.log -hier -collapse_genblks`

Output:

```
(INSTANCE "top" top
(INSTANCE "mid" mid_inst
(INSTANCE "bot" genblk1\.bot_inst_case_1
)
)
)
```

File: test.sv

```
module top();
mid #(1,1) mid_inst();
endmodule

module mid();
parameter p = 0 ;
parameter p1 = 0;

//===== GEN FOR =====//
generate for ( genvar i=0 ; i<1 ; i++) begin : GEN_FOR
    reg r_gen_for;
    if(p1 == 1) begin
        bot bot_inst_for();
    end
end

endgenerate
endmodule

module bot();
endmodule
```

Command:

`dumpsaif -scope top -internal -memories -overwrite -output saif.log -hier -collapse_genblks`

Output:

```
(INSTANCE "top" top
  (INSTANCE "mid" mid_inst
    (NET
      (GEN_FOR\[0\]\.r_gen_for
        (T0 0) (T1 0) (TX 0)
        (TZ 0) (TB 0) (TC 0)
      )
    )
  (INSTANCE "bot" GEN_FOR\[0\]\.genblk1\.bot_inst_for
  )
)
```

File: test.sv


```
module top();
mid #(1,1) mid_inst();
endmodule

module mid();
parameter p = 0 ;
parameter p1 = 0;

//===== Nested GEN =====//
generate for (genvar ii = 0 ; ii < 1 ; ii++) begin : GEN_NESTED_II
    reg r_gen_nested_1;

    bot bot_inst_nested_ii();

    for (genvar jj = 0 ; jj < 2 ; jj++) begin : GEN_NESTED_JJ
        reg r_gen_nested_3;

        bot bot_inst_nested_ii_jj();
    end
    reg r_gen_nested_5;

    bot bot_inst_nested_ii_jj_1();
end

endgenerate
endmodule

module bot();
endmodule
```

Command:

```
dumpsaif -scope top -internal -memories -overwrite -output saif.log -hier -collapse_genblks
```

Output:

```
(INSTANCE "top" top
  (INSTANCE "mid" mid_inst
    (NET
      (GEN_NESTED_II\[0\]\.r_gen_nested_1
        (T0 0) (T1 0) (TX 0)
        (TZ 0) (TB 0) (TC 0)
      )
      (GEN_NESTED_II\[0\]\.r_gen_nested_5
        (T0 0) (T1 0) (TX 0)
        (TZ 0) (TB 0) (TC 0)
      )
    )
  )
  (INSTANCE "bot" GEN_NESTED_II\[0\]\.bot_inst_nested_ii
```

```

)
  (NET
    (GEN_NESTED_II\[0\]\.GEN_NESTED_JJ\[0\]\.r_gen_nested_3
      (T0 0) (T1 0) (TX 0)
      (TZ 0) (TB 0) (TC 0)
    )
  )
)
(INSTANCE "bot" GEN_NESTED_II\[0\]\.GEN_NESTED_JJ\[0\]\.bot_inst_nested_ii_jj
)
  (NET
    (GEN_NESTED_II\[0\]\.GEN_NESTED_JJ\[1\]\.r_gen_nested_3
      (T0 0) (T1 0) (TX 0)
      (TZ 0) (TB 0) (TC 0)
    )
  )
)
(INSTANCE "bot" GEN_NESTED_II\[0\]\.GEN_NESTED_JJ\[1\]\.bot_inst_nested_ii_jj
)
(INSTANCE "bot" GEN_NESTED_II\[0\]\.bot_inst_nested_ii_jj_1
)
)
)

```

File: cs_if.vhd

```

-----top module-----
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity GEN_BLOCK is
  port( x,y: in std_logic;
    sum_xy,carry_xy,borrow_xy: out std_logic);
end GEN_BLOCK;

architecture behave of GEN_BLOCK is

  component HA
    port (a,b: in std_logic;
      sum,carry: out std_logic);
  end component;

  component HS
    port (a,b: in std_logic;
      sum,borrow: out std_logic);
  end component;

begin

```

```
G1: for I in 0 to 1 generate

  G2: if I = 0 generate
    DUT1: HA port map( x,y,sum_xy,carry_xy);
  end generate G2;

  G3: if I = 1 generate
    DUT2: HS port map( x,y,sum_xy,borrow_xy);
  end generate G3;

end generate G1;

end behave;

-----HA-----
library IEEE;
use IEEE.std_logic_1164.all;

entity HA is
  port (a,b: in std_logic;
        sum,carry: out std_logic);
end HA;

architecture behave of HA is
begin
  sum <= a xor b;
  carry <= a and b;
end behave;

-----HS-----
library IEEE;
use IEEE.std_logic_1164.all;

entity HS is
  port (a,b: in std_logic;
        sum,borrow: out std_logic);
end HS;

architecture behave of HS is
begin
  sum <= a xor b;
  borrow<= (not a) and b;
end behave;
```

Command:

`dumpsaif -scope : -internal -memories -overwrite -output saif.log -hier -collapse_genblks`

Output:

```
(INSTANCE "GEN_BLOCK" behave
(INSTANCE "HA" G1\ (0\)\.G2\.DUT1
)
(INSTANCE "HS" G1\ (1\)\.G3\.DUT2
)
)
```

Examples Using the `dumpsaif -new_escape_name_format` Command Option

File: test.v

```
module mod(\port[0]_test1 , net);
    input \port[0]_test1 ;
    output net;
endmodule

module top;

    wire clk;
    wire \net(ab)_q ;
    wire \[]net ;
    wire \()net$ ;

    mod m1(clk, \net(ab)_q );
    mod \m1_[2]_inst (\[]net , \()net );
endmodule // top
```

Tcl file: input.tcl

```
dumpsaif -scope top -internal -memories -overwrite -output xmsim.saif -
new_escape_name_format
dumpsaif -end

dumpsaif -scope top -internal -memories -overwrite -output xmsim_wo.saif
dumpsaif -end

exit
```

Command: Using the `-new_escape_name_format` command option

```
dumpsaif -scope top -internal -memories -overwrite -output xmsim.saif -new_escape_name_format
```

Output:

```
(DIVIDER . )
(TIMESCALE 1 fs )
(DURATION 0)
```

```
(INSTANCE "top" top
(NET
(clk
(T0 0) (T1 0) (TX 0)
(TZ 0) (TB 0) (TC 0)
)
(net\ab\)_q
(T0 0) (T1 0) (TX 0)
(TZ 0) (TB 0) (TC 0)
)
(\[\]net
(T0 0) (T1 0) (TX 0)
(TZ 0) (TB 0) (TC 0)
)
(\(\)net$
(T0 0) (T1 0) (TX 0)
(TZ 0) (TB 0) (TC 0)
)
)
)
(INSTANCE "mod" top.m1
(PORT
(port\[0\]_test1
(T0 0) (T1 0) (TX 0)
(TZ 0) (TB 0) (TC 0)
)
(net
(T0 0) (T1 0) (TX 0)
(TZ 0) (TB 0) (TC 0)
)
)
)
(INSTANCE "mod" top.m1_\[2\]_inst
(PORT
(port\[0\]_test1
(T0 0) (T1 0) (TX 0)
(TZ 0) (TB 0) (TC 0)
)
(net
(T0 0) (T1 0) (TX 0)
(TZ 0) (TB 0) (TC 0)
)
)
)
)
```

Command: Not using the `-new_escape_name_format` command option

```
dumpsaif -scope top -internal -memories -overwrite -output xmsim_wo.saif
```

Output:

```
(DIVIDER . )
(TIMESCALE 1 fs )
(DURATION 0)
(INSTANCE "top" top
  (NET
    (clk
      (T0 0) (T1 0) (TX 0)
      (TZ 0) (TB 0) (TC 0)
    )
    (\net(ab)_q
      (T0 0) (T1 0) (TX 0)
      (TZ 0) (TB 0) (TC 0)
    )
    ([ ]net
      (T0 0) (T1 0) (TX 0)
      (TZ 0) (TB 0) (TC 0)
    )
    (\()net\$(
      (T0 0) (T1 0) (TX 0)
      (TZ 0) (TB 0) (TC 0)
    )
    (\()net
      (T0 0) (T1 0) (TX 0)
      (TZ 0) (TB 0) (TC 0)
    )
  )
)
(INSTANCE "mod" top.m1
  (PORT
    (\port[0]_test1
      (T0 0) (T1 0) (TX 0)
      (TZ 0) (TB 0) (TC 0)
    )
    (net
      (T0 0) (T1 0) (TX 0)
      (TZ 0) (TB 0) (TC 0)
    )
  )
)
(INSTANCE "mod" top.\m1_[2]_inst
  (PORT
    (\port[0]_test1
```

```

        (T0 0) (T1 0) (TX 0)
        (TZ 0) (TB 0) (TC 0)
    )
    (net
        (T0 0) (T1 0) (TX 0)
        (TZ 0) (TB 0) (TC 0)
    )
)
)
)

```

dump tcf

The Tcl `dump tcf` command generates a Toggle Count Format (TCF) file during simulation. The TCF is the Cadence standard to describe the switching activity information in the design. The switching activity information contained in the file is required for accurate power analysis or power optimization of a design.

The switching activity includes:

- The toggle count, which indicates how often the pin or net switches between the logic 1 and logic 0 states for the duration of the TCF dump.
- The probability of the pin or net to be in the logic 1 state (total time in logic state 1 / total time of TCF dump).

For VHDL, state `L` is considered as 0, while `H` is considered as 1.

This information is collected for pins (ports in VHDL or interface elements in Verilog) and for nets (signals in VHDL or wires in Verilog).

✓ By default, only transitions from 0 to 1 and 1 to 0 are included in the toggle count. Use the `-inctoggle` option to dump TCF data for `x` and `z` transitions.

The format of the TCF output generated by the `dump tcf` command is identical to the format of TCF files generated by the RTL Compiler `write_tcf` command.

See the *Toggle Count Format Reference*, which is part of the RTL Compiler documentation set, for details on the TCF file.

By default, a TCF file called `xmsim.tcf` is written to the directory from which the simulator was invoked. Use the `-output` option to specify a different name or location for the output file.

dump tcf Command Syntax

`dump tcf`

```
[-compress]
[-collapse_genblks]
[-depth integer | to_cells]
[-dumpportsonly]
[-dumpwireasnet]
[-end]
[-eot]
[-flatformat]
[-inctoggle]
[-internal]
[-memories]
[-output filename]
[-overwrite]
[-scope scope_identifier]
[-timescale]
[-verbose]
```

dumpstcf Command Options

This section describes the options that you can use with the Tcl `dumpstcf` command.

| | |
|--------------------------------|--|
| <code>-compress</code> | <p>Dumps the output in compressed format. This option generates the output tcf file in <code>.gz</code> format.</p> <p>Use the <code>-compress</code> option to compress the output file to save the total disk space usage. This compressed output file is also be read by the RTL Compiler.</p> <pre>xcelium> dumpstcf -scope : -internal -memories -overwrite -compress</pre> <p>A file named <code>xmsim_backward.saif.gz</code> is created. To use a custom filename for the compressed file, add the <code>-output filename</code> option.</p> |
| <code>-collapse_genblks</code> | <p>Dumps the generate block instances along with the internal module instances instead of dumping them separately.</p> <p>By default, when the <code>dumpsaiif</code> command is executed on a design that contains generate blocks, then each generate block is represented as a separate instance. Use the <code>-collapse_genblks</code> option to prepend the generate block instances along with the internal module instances.</p> <div style="border: 1px solid #f0e68c; padding: 10px; margin: 10px 0;"><p>If there are no generate blocks in the design, then using the <code>-collapse_genblks</code> option will not affect anything.</p></div> |

See the `-collapse_genblks` command examples.

| | |
|--|---|
| <code>-depth integer to_cells</code> | <p>Dumps the TCF file for the specified number of levels in the hierarchy or stops at cells.</p> <p>By default, the <code>dumptcf</code> command dumps a TCF file for all levels of a scope specified with the <code>-scope</code> option. If you do not use the <code>-scope</code> option, the top-level design unit is used as the scope.</p> <p>Use the <code>-depth</code> option to dump variables up to a given depth in the hierarchy or use the <code>to_cells</code> argument to stop the descent at cells. For example, if a top-level module named <code>top</code> instantiates a module <code>middle</code>, and <code>middle</code> instantiates another module <code>bottom</code> (and both modules <code>middle</code> and <code>bottom</code> are defined in a cell), using:</p> <ul style="list-style-type: none"> • <code>-depth 2</code> prints the variables of <code>top</code> and <code>top.middle</code>. • <code>-depth 3</code> prints the variables of <code>top</code>, <code>top.middle</code>, and <code>top.middle.bottom</code>. • <code>-depth to_cells</code> prints the variables of <code>top</code>. |
| <code>-dumpportsonly</code> | Dumps only ports in the pins section. |
| <code>-dumpwireasnet</code> | <p>Dumps only ports in the pins section of the TCF file.</p> <p>By default, the simulator reports the toggle count for nets/wires in the pins category of the TCF file. This format could cause annotation issues in downstream power analysis tools.</p> <p>Use the <code>-dumpwireasnet</code> option to dump only ports in the pins section. All wires are dumped in the nets section of the file. For example:</p> <pre> module mod(in1, in2, out1); input in1; input in2; output out1; wire w1; reg r2; // Some logic endmodule </pre> <p>The following command dumps the ports (<code>in1</code>, <code>in2</code>, and <code>out1</code>) in the pins section of the output file. The wire <code>w1</code> is dumped in the nets section:</p> <pre> xcelium> dumptcf -scope mod -dumpwireasnet </pre> <p>The following command dumps the ports (<code>in1</code>, <code>in2</code>, and <code>out1</code>) in the pins section of the output file. The wire <code>w1</code> and the reg <code>r2</code> are dumped in the nets section:</p> <pre> xcelium> dumptcf -scope mod -internal -dumpwireasnet </pre> <p>The following command dumps the ports (<code>in1</code>, <code>in2</code>, and <code>out1</code>) in the pins section of the output file. The wire <code>w1</code> is dumped in the nets section:</p> <pre> xcelium> dumptcf -scope mod -dumpwireasnet </pre> <p>The following command dumps the ports (<code>in1</code>, <code>in2</code>, and <code>out1</code>) in the pins section of the output file. The wire <code>w1</code> and the reg <code>r2</code> are dumped in the nets section:</p> <pre> xcelium> dumptcf -scope mod -internal -dumpwireasnet </pre> |

| | |
|--------------------------|--|
| <code>-end</code> | <p>Ends all TCF probing activity.</p> <p>By default, the TCF file is written at the end of the simulation. Use the <code>dumptcf -end</code> command at any point during the simulation to close the TCF database and write the TCF file. Any simulation activity after a <code>dumptcf -end</code> command is not reflected in the output file.</p> <p>You cannot restart the TCF dumping to the same TCF file.</p> |
| <code>-eot</code> | <p>Reports only transitions that occur after a timing interval.</p> <p>By default, the TCF file generated by the <code>dumptcf</code> command shows transitions after every delta and timing delay. This does not match the TCF file that is generated by converting a VCD file in a competitor simulator because that simulator considers transitions only at the end of each timing interval.</p> <p>The <code>-eot</code> option forces the simulator to consider the values of signals after each timing interval, rather than after each delta interval. Only those transitions that occur after a timing interval are reported in the TCF file. If the value of a signal is being changed multiple times after each delta delay, the TCF file contains transition values only after each timing delay, rather than after both timing and delta delays.</p> |
| <code>-flatformat</code> | <p>Specifies that the TCF output should be in flat format.</p> <p>A TCF file can contain the switching activity information in a flat or hierarchical representation.</p> <ul style="list-style-type: none">• In the flat representation, all net and pin names are specified by a full path with respect to the top-level design.• In a hierarchical representation, the net and pin names are specified with respect to the instance scope. <p>By default, the <code>dumptcf</code> command dumps a TCF file in a hierarchical format. Use the <code>-flatformat</code> option if you want a flat representation.</p> |
| <code>-inctoggle</code> | <p>Dumps TCF data for X and Z transitions.</p> <p>By default, only transitions from 0 to 1 and 1 to 0 are included in the toggle count. Use the <code>-inctoggle</code> option to include data for X and Z transitions. The additional information in the TCF file includes:</p> <ul style="list-style-type: none">• The toggle counts for X and Z states. The probability of the pin or net to be in the X state, and the probability of the pin or net to be in the Z state.<ul style="list-style-type: none">◦ How often the pin or net switches from 1/0/X to X, and vice-versa◦ How often the pin or net switches from 1/0/X/Z to Z, and vice-versa |

`-internal`

Enables probing of internal Verilog regs and VHDL signals to the TCF file.

By default, ports and Verilog internal wires are probed. Use the `-internal` option to include internal Verilog regs and VHDL signals in the probe. For example:

```
module mod(in1, in2, out1);
    input in1;
    input in2;
    output out1;
    wire w1;
    reg r2;
    // Some logic
endmodule
```

The following command dumps the ports (`in1`, `in2`, and `out1`) and the wire `w1` in the pins section of the output file:

```
xcelium> dumptcf -scope mod
```

The following command dumps the ports (`in1`, `in2`, and `out1`), the wire `w1`, and the reg `r2` in the pins section of the output file:

```
xcelium> dumptcf -scope mod -internal
```

The following command dumps the ports (`in1`, `in2`, and `out1`) in the pins section of the output file. The wire `w1` and the reg `r2` are dumped in the nets section:

```
xcelium> dumptcf -scope mod -internal -dumpwiresnet
```

`-memories`

Generates switching and toggle information for unpacked registers and unpacked nets. For example:

```
logic [3:0] unpacked_array [5:0];
```

Use the `-memories` option to dump toggle information for:

- unpacked array
- unpacked struct
- arrays of MDV
- unpacked unions

`-output filename`

Specifies the path to the output TCF file.

By default, the TCF file is generated in the directory from which the simulator was invoked, and the file is called `xmsim.tcf`. Use the `-output` option to override the default. For example:

```
xcelium> dumptcf -output tcf.dump
xcelium> dumptcf -scope :dut -output ./tcfout/tcfdump.file
```

If a file with the specified name already exists, an error is generated telling you that the existing file is not overwritten unless the `-overwrite` option is used.

`-overwrite`

Enables overwriting of an existing TCF file.

By default, the `dumptcf` command does not overwrite an existing TCF file with the same name (`xmsim.tcf` or the name specified with the `-output` option).

| | |
|--------------------------------------|---|
| <code>-scope scope_identifier</code> | <p>Specifies the scope of the hierarchy for which the TCF needs to be dumped.</p> <p>By default, the top-level scope or the current debug scope is the scope for dumping TCF. Use the <code>-scope</code> option to specify a scope in the design. The <code>scope_identifier</code> argument is a Verilog or VHDL hierarchical path that determines the domain of the probing activity. All instances and cells below this hierarchy are probed for the TCF information.</p> |
| <code>-timescale</code> | <p>Converts the duration in different time units. The supported time units are <code>fs</code>, <code>ps</code>, <code>ns</code>, <code>us</code>, <code>ms</code>, and <code>s</code>. If you do not specify anything, the duration is shown in <code>fs</code> which is the default unit.</p> <pre>xcelium> dumptcf -scope tb.u_sample -overwrite -internal -timescale us duration : "2"; unit : "us";</pre> |
| <code>-verbose</code> | Displays informational messages during the generation of the TCF file. |

dumptcf Command Limitations

The following limitations exist on the dumping of TCF databases:

- Multiple TCF files cannot be dumped at the same time. Only one `dumptcf` command is active during a given simulation timeframe. Another `dumptcf` command can be issued only if the first has been terminated with `dumptcf -end`.
- All generated TCF files have a default timescale of 1 fs, but this can be changed using `-timescale` option.
- Real and other complex data types, such as access types, file types, and physical data types, are not supported.
- Arrays of unpacked struct and unpacked union is not supported.
- Internal signals can be probed for TCF activity. However, variables within a process are dropped from the list of elements being probed.

dumptcf Command Examples

The following command enables TCF probing. No scope is specified, so the simulator uses the top-level scope (or the current debug scope) for dumping TCF. A TCF file in a hierarchical format called `xmsim.tcf` is written to the directory from which the simulator was invoked:

```
xcelium> dumptcf
```

In the following command, the `-scope` option specifies the Verilog scope `top.dut`. All instances and cells below this hierarchy are probed for the TCF information. Only ports are probed:

```
xcelium> dumptcf -scope top.dut
```

The following command includes the `-depth` option to limit the number of levels of hierarchy to be dumped:

```
xcelium> dump tcf -scope :dut -depth 3
```

In the following command, the `-scope` option specifies the VHDL scope `:dut`. The `-internal` option is included to enable probing of internal signals/wires:

```
xcelium> dump tcf -scope :dut -internal
```

By default, the name of the TCF output file is `xmsim.tcf`. The following command includes the `-output` option to specify that the output file is to be called `tcf.dump`:

```
xcelium> dump tcf -scope top.dut -output tcf.dump
```

In the following sequence of commands, the simulation is run for 3000 ns. A `dump tcf -end` command is then issued to close the TCF database and end the TCF dumping. The `-flatformat` option is used to generate an output file in flat format:

```
xcelium> dump tcf -scope testbench.top -output tcf.dump -flatformat -overwrite  
xcelium> run 3000 ns  
xcelium> dump tcf -end  
xcelium> run  
xcelium> exit
```

The following command includes the `-memories` option to dump toggle information for a multi-dimensional unpacked array of `reg` type, unpacked structs, and unpacked unions:

```
module top();

// Packed Arrays 1-D
reg [1:0] pa;

// Unpacked Arrays 2-D
reg c [1:0][1:0];

//Unpacked Struct
struct {reg r; reg [1:0] arr;} st;

reg clk;

always @(clk)
begin
    c[0][0] =clk;
    c[0][1] =~clk;
    c[1][0] =clk;
    c[1][1] =~clk;
    pa[0] =clk;
    pa[1]= ~clk;
    st.arr[0]= clk;
    st.arr[1] =~clk;
    st.r = clk;
end

initial
begin
    clk = 1'b0;
    forever #20
    clk = ~clk;
end
endmodule
```

xcelium> **dumptcf -scope top -internal -memories**

The TCF file generated for this Verilog code is as follows:

```
tcffile () {
    tcfversion      :      "1.0";
    generator       :      "NC Simulation Engine";
    genversion      :      "TOOL:  xmsim   17.10-a001-20170922";
    duration        :      "1e+08";
    unit            :      "fs";
    instance("top") {
        pin() {
            "pa[1]" :  "0.600000  4";
            "pa[0]" :  "0.400000  4";
            "c[1][1]" :  "0.600000  4";
            "c[1][0]" :  "0.400000  4";
            "c[0][1]" :  "0.600000  4";
            "c[0][0]" :  "0.400000  4";
            "st.r"   :  "0.400000  4";
            "st.arr[1]" :  "0.600000  4";
            "st.arr[0]" :  "0.400000  4";
            "clk"    :  "0.400000  4";
        }
    }
}
```

Examples Using the `-collapse_genblks` Command Option

The following examples show the output generated by `-dumptcf` command when the `-collapse_genblks` option is used.

File: test.sv

```
module miniperipheral2;
endmodule

module miniperipheral1;
generate genvar i;
for(i=0; i<2; i++) begin : pqr
miniperipheral2 peri2();
end
endgenerate
endmodule

module sample(clk, q);

input clk;
output q;
reg r_data;

assign q = r_data;

always @(clk)
r_data = clk;

generate genvar i,j,k,l;
for(i=0; i<2; i++) begin : abc
miniperipheral1 peril();
end
endgenerate

endmodule
```

Command:

```
xcelium> dumptcf -scope sample -overwrite -eot -internal -collapse_genblks -output tcf.log
```

Use the following command if running the command from *xrun*:

```
% xrun -REDUCE_MESSAGES -NOCOPYRIGHT -clean -access +rwc -input run.tcl -sv -v93 -status
test.sv -log_xmsim xmsim.log
```

Output:

```
tcffile () {
    tcfversion : "1.0";
    generator : "NC Simulation Engine";
    genversion : "TOOL: xmsim 24.02-a071-20240215";
    duration : "0";
    unit : "fs";
    instance("sample") {
        pin() {
```



```
        "clk" : "0.0 0";
        "q" : "0.0 0";
        "r_data" : "0.0 0";
    }
    instance("abc[0].peri1") {
    instance("pqr[0].peri2") {
    }
    instance("pqr[1].peri2") {
    }
    }
    instance("abc[1].peri1") {
    instance("pqr[0].peri2") {
    }
    instance("pqr[1].peri2") {
    }
    }
    }
```

File: test.sv

```
module top();
mid #(1,1) mid_inst();
endmodule

module mid();
parameter p = 0 ;
parameter p1 = 0;

//===== GEN CASE =====//
generate case (p1)
    //reg r_gen_case;
    0: bot bot_inst_case_0();
    1: bot bot_inst_case_1();
endcase
endgenerate
endmodule

module bot();
endmodule
```

Command:

```
dump tcf -scope top -internal -memories -overwrite -output tcf.log -collapse_genblks
```

Output:

```
tcffile () {
```

```
tcfversion :      "1.0";
generator  :      "NC Simulation Engine";
genversion :      "TOOL:  xmsim   24.02-a071-20240215";
duration   :      "0";
unit       :      "fs";
instance("top") {
instance("mid_inst") {
instance("genblk1\bot_inst_case_1") {
}
}
}
```

File: test.sv

```
module top();
mid #(1,1) mid_inst();
endmodule

module mid();
parameter p  = 0 ;
parameter p1 = 0;

//===== GEN FOR =====//
generate for ( genvar i=0 ; i<1 ; i++) begin : GEN_FOR
    reg r_gen_for;
    if(p1 == 1) begin
        bot bot_inst_for();
    end
end

endgenerate
endmodule

module bot();
endmodule
```

Command:

```
dumptcf -scope top -internal -memories -overwrite -output tcf.log -collapse_genblks
```

Output:

```
tcffile () {
    tcfversion :      "1.0";
    generator  :      "NC Simulation Engine";
    genversion :      "TOOL:  xmsim   24.02-a071-20240215";
    duration   :      "0";
    unit       :      "fs";
```

```
instance("top") {
instance("mid_inst") {
    pin() {
        "GEN_FOR[0]\.r_gen_for" : "0.0 0";
    }
instance("GEN_FOR[0].genblk1\.bot_inst_for") {
}
}
}
```

File: test.sv

```
module top();
mid #(1,1) mid_inst();
endmodule

module mid();
parameter p = 0 ;
parameter p1 = 0;

//===== Nested GEN =====//
generate for (genvar ii = 0 ; ii < 1 ; ii++) begin : GEN_NESTED_II
    reg r_gen_nested_1;

    bot bot_inst_nested_ii();

    for (genvar jj = 0 ; jj < 2 ; jj++) begin : GEN_NESTED_JJ
        reg r_gen_nested_3;

        bot bot_inst_nested_ii_jj();

        end
        reg r_gen_nested_5;

        bot bot_inst_nested_ii_jj_1();

        end
    endgenerate
endmodule

module bot();
endmodule
```

Command:

```
dumptcf -scope top -internal -memories -overwrite -output tcf.log -collapse_genblks
```

Output:

```
tcffile () {
```

```

tcfversion      :      "1.0";
generator       :      "NC Simulation Engine";
genversion      :      "TOOL:  xmsim   24.02-a071-20240215";
duration        :      "0";
unit            :      "fs";
instance("top") {
instance("mid_inst") {
    pin() {
        "GEN_NESTED_II[0]\.r_gen_nested_1"  :  "0.0  0";
        "GEN_NESTED_II[0]\.r_gen_nested_5"  :  "0.0  0";
    }
instance("GEN_NESTED_II[0].bot_inst_nested_ii") {
}
    pin() {
        "GEN_NESTED_II[0].GEN_NESTED_JJ[0]\.r_gen_nested_3"  :  "0.0  0";
    }
instance("GEN_NESTED_II[0].GEN_NESTED_JJ[0].bot_inst_nested_ii_jj") {
}
    pin() {
        "GEN_NESTED_II[0].GEN_NESTED_JJ[1]\.r_gen_nested_3"  :  "0.0  0";
    }
instance("GEN_NESTED_II[0].GEN_NESTED_JJ[1].bot_inst_nested_ii_jj") {
}
instance("GEN_NESTED_II[0].bot_inst_nested_ii_jj_1") {
}
}
}

```

File: cs_if.vhd

```

-----top module-----
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity GEN_BLOCK is
    port(  x,y: in std_logic;
           sum_xy,carry_xy,borrow_xy: out std_logic);
end GEN_BLOCK;

architecture behave of GEN_BLOCK is

component HA
    port (a,b: in std_logic;
          sum,carry: out std_logic);

```

```
    sum,carry: out std_logic,,
end component;

component HS
    port (a,b: in std_logic;
          sum,borrow: out std_logic);
end component;

begin

G1: for I in 0 to 1 generate

    G2: if I = 0 generate
        DUT1: HA port map( x,y,sum_xy,carry_xy);
    end generate G2;

    G3: if I = 1 generate
        DUT2: HS port map( x,y,sum_xy,borrow_xy);
    end generate G3;

end generate G1;

end behave;
```

```
-----HA-----
library IEEE;
use IEEE.std_logic_1164.all;

entity HA is
    port (a,b: in std_logic;
          sum,carry: out std_logic);
end HA;

architecture behave of HA is
begin
    sum <= a xor b;
    carry <= a and b;
end behave;
```

```
-----HS-----
library IEEE;
use IEEE.std_logic_1164.all;

entity HS is
    port (a,b: in std_logic;
          sum,borrow: out std_logic);
end HS;

architecture behave of HS is
```

```
begin
sum <= a xor b;
borrow<= (not a) and b;
end behave;
```

Command:

`dumptcf -scope : -internal -memories -overwrite -output tcf.log -collapse_genblks`

Output:

```
tcf file () {
  tcfversion : "1.0";
  generator : "NC Simulation Engine";
  genversion : "TOOL: xmsim 24.02-a071-20240215";
  duration : "0";
  unit : "fs";
  instance("behave") {
    pin() {
      "x" : "0.0 0";
      "y" : "0.0 0";
      "sum_xy" : "0.0 0";
      "carry_xy" : "0.0 0";
      "borrow_xy" : "0.0 0";
    }
    instance("G1[0]") {
      instance("G2") {
        instance("DUT1") {
          pin() {
            "a" : "0.0 0";
            "b" : "0.0 0";
            "sum" : "0.0 0";
            "carry" : "0.0 0";
          }
        }
      }
    }
    instance("G1[1]") {
      instance("G3") {
        instance("DUT2") {
          pin() {
            "a" : "0.0 0";
            "b" : "0.0 0";
            "sum" : "0.0 0";
            "borrow" : "0.0 0";
          }
        }
      }
    }
  }
}
```

```
}  
}  
}  
}
```

exit

The Tcl `exit` command terminates the simulation and returns control to the operating system.

 The `finish` command can also exit a simulation.

exit Command Syntax

```
exit  
-analogsolver
```

exit Command Options

This section describes the options that you can use with the Tcl `exit` command.

| | |
|----------------------------|--|
| <code>-analogsolver</code> | Exits the analog solver and saves a snapshot of the setup so that you can rework from that checkpoint. |
|----------------------------|--|

exit Command Examples

The following command ends the simulation session:

```
xcelium> exit
```

fault

The Tcl `fault` command injects a fault of type SA0, SA1, SEU, or SET during a targeted fault campaign by specifying one or more fault node names at simulation. The specified fault nodes must fall within the fault injection region defined by the `fault_target` command, which is instrumented during elaboration using a fault specification file. If you do not specify a node name or if the specified node name is not found, then the simulator generates an error.

fault Command Syntax

```
fault -inject <list_of_nodes> -type <fault_type>
      [-time <start_time[:end_time]>]
      [-absolute | -relative]
      [-timeout <time>]
      [-wooid <string>]
```

Or:

```
fault -stop_severity <number> [-inject <list_of_nodes> -type <fault_type>]
```

fault Command Options

This section describes the options that you can use with the Tcl `fault` command.

`-inject <list_of_nodes>`

Uses this option together with `-type` to apply a fault to one or more specified fault nodes. Both options are required. A fault node can be a hierarchical name or a simple name in the current scope. The node must resolve to a scalar object. Otherwise, the software generates an error.



If multiple faults are injected in a single fault simulation run, then they are each treated as independent faults. Multiple faults interact with each other and affect simulation behavior.

`-type <fault_type>`

Uses this option with `-inject` to indicate the type of fault model to apply to the specified fault nodes. Both options are required.

Xcelium supports the following fault model types:

- Stuck-at-0 (`sa0`)
- Stuck-at-1 (`sa1`)
- Single event upset (`seu`)
- Single event transient with a specified time delay (`set+time_spec`)
The specified time is the period of time the fault type is in effect.

`-time <start_time>`
`[:<end_time>]`
`[-absolute | -relative]`

Specifies the injection time in which to apply a fault. The time can be either `-relative` to the current simulation time (the default setting) or `-absolute`.

If the `start_time` and `end_time` are both specified, the simulator picks a random time within the set time window as the injection time for all fault nodes. If only the `start_time` is specified, then this time becomes the injection time for all fault nodes.



When a `fault -inject` command is run at time 0, the relative and absolute times are the same.

`-timeout <time>`

Specifies the time to terminate a serial fault simulation when injecting a transient SEU fault type. The *time* value is relative to the fault injection time. Using this option when injecting SA0, SA1, or SET fault types results in an FLTSEU error.

This option is not supported by the concurrent engine.

If both the `-fault_timeout` command-line option and the Tcl command `fault -inject -timeout` are specified for a serial fault simulation, then the Tcl `fault` command takes precedence.

`-wooid <string>`

Specifies a Window of Opportunity ID when injecting a transient SEU fault type. Using this option when injecting SA0, SA1, or SET fault types results in an FLTSEU error.

A Window of Opportunity ID is a string which can help to identify unique faults in a transient fault list and is used by other tools in the Cadence Safety flow. The ID can include the following legal characters but note that only letters are allowed as the first character of the string.

- Lowercase letters
- Uppercase letters
- Digits (0123456789)

`-stop_severity <num>`

Indicates the stop-level severity for the simulator after generating observation points with the `fs_strobe` command.

Choose one of the following arguments when specifying this option (the default value is 1):

- 0: Continues the simulation to the end.
- 1: Stops the simulation after detecting a fault.
- 2: Stops the simulation when the injected faults are potentially detected.
- 3: Stops the simulation when the injected faults are detected or potentially detected.

fault Command Examples

The following command injects a fault of type `SA1` to multiple fault nodes `top.dut_inst.in[0]` and `top.dut_inst.sel` using a specified time window between 1000 and 3000ns:

```
xcelium> fault -inject -time 1000:3000 -type sa1 top.dut_inst.in[0] top.dut_inst.sel
```

These two commands use different fault parameters to inject different fault types (`SEU` and `SA0`) to different fault nodes (`top.dut_inst.out[1]` and `top.dut_inst.sel`), and at different injection times:

```
xcelium> fault -inject -time 300:500 -type seu top.dut_inst.out[1]
xcelium> fault -inject -time 1025ns -type sa0 top.dut_inst.sel
```

This command sets the level of stop severity to 3, which halts the simulation when all injected faults are DETECTED or POTENTIALLY_DETECTED:

```
xcelium> fault -stop_severity 3
```

Related Topics

- [fs_strobe](#)
- [Fault Injection](#)
- [The Fault Specification File](#)

find

The Tcl `find` command searches for objects in the design hierarchy.

By default, this command displays a list of HDL objects in the current debug scope whose name matches the specified *object_name* argument. It prints each object to a single line so that you can then reuse this output as input to other Tcl commands which accept lists of objects, such as the `force`, `deposit`, `value`, `stop`, and `probe` commands.

This command enables you to restrict the search to certain objects such as ports, VHDL signals and variables, Verilog wires and registers, instances, and so on. Additionally, you can modify the default output format. For instance, you can choose to have the simulator list objects on separate lines instead or on one line or use absolute paths instead of relative paths.

The following are common tasks that you can perform with this command:

- [Searching for objects in a specified scope](#)
- [Searching for objects in all scopes or a set number of subscopes](#)

The `find` command applies to Verilog and VHDL design objects only.

SystemC, analog, SystemVerilog constructs, and assertion objects are not supported.

find Command Syntax


```
find [options] object_name [object_name ...]
    [-absolute]
    [-blocks]
    [-instances]
    [-internals]
        [-registers]
        [-signals]
        [-variables]
        [-wires]
    [-newline]
    [-nocase]
    [-packages] [-exclude <package_name>]
    [-ports]
        [-inputs]
        [-outputs]
        [-inouts]
        [-intfports]
        [-refports]
    [-recursive] [{<levels>|all}]
    [-scope scope]
    [-subprograms]
    [-verbose]
```

- The syntax of this command is the same for Verilog and VHDL; however, in the output, the Verilog hierarchy separator is a dot (.) and the VHDL separator is a colon (:)
- The wildcard characters * and ? are allowed in the *object_name* specification

find Command Options


This section describes the options that you can use with the Tcl `find` command.

| | |
|------------------------|--|
| <code>-absolute</code> | Displays the absolute path for each object. By default, the <code>find</code> command displays paths relative to the current debug scope. |
|------------------------|--|

| | |
|--|---|
| <code>-blocks</code> | <p>Searches for named or unnamed scopes that match the object name, rather than matching instances in the design.</p> <p>The matching blocks can be Verilog named blocks, specify blocks, VHDL processes, VHDL blocks, for-generate, or if-generate blocks.</p> <div>  Use both the <code>-blocks</code> and <code>-instances</code> options to display all scoped objects in the design. </div> |
| <code>-instances</code> | <p>Searches only for modules, UDPs, and component instances that match the object name.</p> |
| <code>-internals</code> | <p>Searches for internal objects only and not ports. To search for both ports and internal objects, include the <code>-ports</code> option.</p> <p>Additionally, you can use one of the following modifiers with this option to further filter the internal objects:</p> <ul style="list-style-type: none"> • -registers: Searches for Verilog internal registers only. • -signals: Searches for VHDL internal signals only. • -variables: Searches for VHDL internal variables only. • -wires: Searches for Verilog internal wires only. |
| <code>-newline</code> | <p>Displays objects on separate lines. By default, objects are displayed on a single line.</p> <p>This option also generates a warning message if unsupported constructs or protected code is encountered.</p> |
| <code>-nocase</code> | <p>Performs a case-insensitive search. By default, the search is case-sensitive.</p> <p>This option is useful when searching for Verilog objects because Verilog, unlike VHDL, is case-sensitive.</p> |
| <code>-packages</code> | <p>Includes packages when searching for objects.</p> <p>This option extends the search to include objects in packages.</p> |
| <code>-exclude</code> <code><package_name></code> | <p>Excludes a specified package from the search.</p> <p>You can specify multiple <code>-exclude</code> options in a single command. Additionally, the <code>package_name</code> argument can include the wildcard characters <code>*</code> and <code>?</code>.</p> |
| <code>-ports</code> | <p>Searches for ports only.</p> <p>Use one of the following modifiers to search for different types of ports. You can use these modifiers with or without the <code>-ports</code> option.</p> <ul style="list-style-type: none"> • -inputs: Searches for input ports only. • -outputs: Searches for output ports only. |

| | |
|-------------------------------------|--|
| | <ul style="list-style-type: none">• -inouts: Searches for inout ports only.• -intfports: Searches for interface ports only.• -refports: Searches for reference ports only. |
| -recursive [<levels> all] | <p>Searches for objects by recursively descending the design hierarchy.</p> <p>You can choose to control the recursive search with the following arguments:</p> <ul style="list-style-type: none">• <levels>: Specifies an explicit number of scope levels to descend when searching for objects. For instance, 0 includes only the given scope, 1 includes the given scope and its subscopes.• all: Specifies that the simulator should include all scopes in the hierarchy below the given scope when searching for objects. This is the default. |
| -scope <scope> | <p>Searches for objects in the specified scope.</p> <p>The specified scope can be relative to the current debug scope or absolute. Additionally, you can use multiple -scope options to specify more than one scope to search.</p> |
| -subprograms | <p>Searches for any subprograms used in the design. This includes VHDL functions and procedures, and Verilog tasks and functions.</p> |
| -verbose | <p>Displays additional information about objects, such as the compiled design unit name, the source file name, the line number of the declaration, and the current value, if applicable.</p> <p>This option also generates a warning message if unsupported constructs or protected code is encountered.</p> |

find Command Examples

 A simple Verilog example is used to illustrate the use model of the `find` command in this section.

The following command displays all objects in the current debug scope:

```
xcelium> find *  
clockGen counter af f clock count
```

The following command includes the **-instances** option to limit the search to instances in the current debug scope:

```
xcelium> find -instances *  
clockGen counter
```

The following command includes the **-scope** option to specify that the search is limited to the scope

called `counter`:

```
xcelium> find -scope counter *
counter.d counter.c counter.b counter.a counter.clock counter.altFifteen counter.fifteen
counter.value
```

The following command includes the `-recursive` option. No argument to the option is specified. The `find` command searches for all objects called `q` in the entire design:

```
xcelium> find -recursive q
counter.a.q counter.b.q counter.c.q counter.d.q
```

The following command searches for objects called `value` in the current debug scope and its subscopes:

```
xcelium> find -recursive 1 value
counter.value
```

The following command searches for output ports in the scope `counter`:

```
xcelium> find -scope counter -ports -outputs *
counter.clock counter.altFifteen counter.fifteen counter.value
```

The following command includes the `-absolute` option. The output displays the output ports using absolute paths:

```
xcelium> find -scope counter -ports -outputs -absolute *
board.counter.altFifteen board.counter.fifteen board.counter.value
```

The following command includes the `-internals` and `-wires` options to limit the search to internal Verilog wires:

```
xcelium> find -internals -wires *
af f clock count
```

The following command includes the `-newline` option to display objects on separate lines:

```
xcelium> find -internals -wires -newline *
af
f
clock
count
```

The following command illustrates the additional information about objects displayed with the `-verbose` option:

```
xcelium> find -recursive -verbose value
board.counter.value...output net logic [3:0]
    value[3] (wire/tri) = St0
    value[2] (wire/tri) = St0
    value[1] (wire/tri) = St1
    value[0] (wire/tri) = St0
Design Unit: @worklib.m16
```

File: ./counter.v
Line number: 13

The following example uses the output of the `find` command as input to a `probe` command:

```
xcelium> probe -create -shm [find -ports -scope counter *]
Created default SHM database xcelium.shm
Created probe 1
xcelium> probe -show
1      Enabled      board.counter.clock (database: xcelium.shm) -shm
                        board.counter.altFifteen
                        board.counter.fifteen
                        board.counter.value
                        Number of objects probed : 4

xcelium>
```

Commands, like the one in the previous example, work even if objects have Verilog escaped names. For example, the following command uses the output of the `find` command as input to a `describe` command:

```
xcelium> describe [find -recursive all -internals -absolute *]
```

or:

```
xcelium> set foo [find -recursive all -internals -absolute *]
xcelium> describe $foo
top.\myBuf[0] .\int[0] ...net (wire/tri) logic = StX
top.\myBuf[0] .\int[1] ...net (wire/tri) logic = StX
top.\myBuf[0] .\int[2] ...net (wire/tri) logic = StX
top.\myBuf[0] .\int[3] ...net (wire/tri) logic = StX
```

However, if you have a Tcl script that repeats an operation based on the output of the `find` command, the path name parser interprets spaces in the output as word separators. For example:

```
xcelium> set foo [find -recursive all -internals -absolute *]
top.\myBuf[0] .\int[0] top.\myBuf[0] .\int[1] top.\myBuf[0] .\int[2] top.\myBuf[0] .\int[3]
xcelium>
xcelium> foreach f $foo {
> puts $f
> }
top.myBuf[0]
.int[0]
top.myBuf[0]
.int[1]
top.myBuf[0]
.int[2]
top.myBuf[0]
.int[3]
```

A workaround for this issue is to include the `-newline` option on the `find` command, and then split the list

using `\n`. For example:

```
xcelium> set foo [find -recursive all -internals -absolute -newline *]
top.\myBuf[0] .\int[0]
top.\myBuf[0] .\int[1]
top.\myBuf[0] .\int[2]
top.\myBuf[0] .\int[3]
xcelium> set foo [split $foo \n]
{top.\myBuf[0] .\int[0] } {top.\myBuf[0] .\int[1] } {top.\myBuf[0] .\int[2] } {top.\myBuf[0]
.\int[3] } {}
xcelium> foreach f $foo {
> puts $f
> }
top.\myBuf[0] .\int[0]
top.\myBuf[0] .\int[1]
top.\myBuf[0] .\int[2]
top.\myBuf[0] .\int[3]
```

Searching for Objects in a Specified Scope

Use the `-scope` option with the `find` command to search for objects only within a given scope of the design.

Syntax

```
find -scope object_name [object_name ...]
```

Options

| | |
|-----------------------------------|--|
| <code>-scope <scope></code> | Searches for objects in the specified scope. The specified scope can be relative to the current debug scope or absolute. Additionally, you can use multiple <code>-scope</code> options to specify more than one scope to search. |
|-----------------------------------|--|

Examples

The following command includes the `-scope` option to specify that the search is limited to the scope called `counter`.

```
xcelium> find -scope counter *
counter.d counter.c counter.b counter.a counter.clock counter.altFifteen counter.fifteen
counter.value
```


Searching for Objects in All Scopes or a Set Number of Subscopes

Use the `-recursive` option with the `find` command to search for objects by recursively descending the design hierarchy.

This option enables you to control the recursive search by:

- Specifying a set number of subscopes, or
- Specifying objects in all scopes.

Syntax

```
find -recursive [{<levels>|all}] object_name [object_name ...]
```

Options

`-recursive`
`[{<levels>|all}]`

Searches for objects by recursively descending the design hierarchy.

You can choose to control the recursive search with the following arguments:

- **<levels>**: Specifies an explicit number of scope levels to descend when searching for objects. For instance, 0 includes only the given scope, 1 includes the given scope and its subscopes.
- **all**: Specifies that the simulator should include all scopes in the hierarchy below the given scope when searching for objects. This is the default.

Examples

The following command includes the `-recursive` option, and no argument to this option is specified. The `find` command searches for all objects called `q` in the entire design:

```
xcelium> find -recursive q  
counter.a.q counter.b.q counter.c.q counter.d.q
```

The following command searches for objects called `value` in the current debug scope and its subscopes:

```
xcelium> find -recursive 1 value  
counter.value
```

finish

The Tcl `finish` command causes the simulator to exit and returns control to the operating system.

finish Command Syntax

```
finish [0|1|2]
```

finish Command Options

This command takes an optional argument that determines what type of information to display.

Choose one of the three possible values shown:

- 0: Prints nothing. This is the default.
- 1: Prints the simulation time.
- 2: Prints the simulation time and statistics on memory and CPU usage.

finish Command Examples

The following command ends the simulation session:

```
xcelium> finish
```

The following command ends the simulation session and prints the simulation time:

```
xcelium> finish 1  
Simulation complete via $finish(1) at time 0 FS +
```

The following command ends the simulation session, prints the simulation time, and displays memory and CPU usage statistics:

```
xcelium> finish 2  
Memory Usage - 7.6M program + 2.1M data = 9.8M total  
CPU Usage - 0.9s system + 2.5s user = 3.4s total (28.5% cpu)  
Simulation complete via $finish(2) at time 500 NS +
```

fmibkpt

The Tcl `fmibkpt` command performs operations on breakpoints that are coded into C models using the `fmiBreakpoint` call.

The simulator initially disables all FMI breakpoints. By using this command, you can choose to enable or disable certain FMI breakpoints or display information about all FMI breakpoints. Be aware that this Tcl command is available only when using the C interface to integrate C models into a VHDL design.

fmibkpt Command Syntax

```
fmibkpt {-enable <bkpt_number>|-disable <bkpt_number>|-show}
```

fmibkpt Command Options

This section describes the options that you can use with the Tcl `fmibkpt` command.

| | |
|---|--|
| <code>-enable <bkpt_number></code> | Enables the FMI breakpoint with the specified number. |
| <code>-disable <bkpt_number></code> | Disables the FMI breakpoint with the specified number. |
| <code>-show</code> | Displays information about all FMI breakpoints. |

force

The Tcl `force` command sets a specified object to a given value and forces it to retain that value until another force is placed on it or until it is released with:

- A `release` command
- A `force -release` command
- A `deposit -release` command

In this way, you can ask "what if" questions about your model by interactively forcing objects to desired values and seeing if a patch fixes the problem. If it does, you can then edit your source file to incorporate the change.

Forces can apply to Verilog, VHDL, or SystemC objects. When specifying a supported object type, it must have write access. If a specified object does not have write access, the simulator will generate an error. By default, new forces will take effect on the specified object immediately. Alternatively, you can choose to use the `-after` option to delay the force assignment for a specified period of time.

For Verilog, be aware that a force created using this command will have identical behavior to a force created using the Verilog `force` procedural statement, meaning that the force can be released by a Verilog `release` statement or replaced by a Verilog `force` statement during a subsequent simulation. Additionally, forces created by the `force` command and those created by Verilog `force` procedural statements are saved if the simulation is saved.

For Verilog wires and VHDL signals and ports, new forces propagate throughout the design hierarchy before Tcl returns to the command prompt. Vector Verilog wires and VHDL signals are compressed if the model does not require operations on individual bits of the vector. For VHDL, you can force a subelement of a compressed vector signal. For Verilog, however, you must elaborate the design with the `-expand` option (`xmelab -expand` or `xrun -expand`) in order to force a subelement of a compressed vector

wire.

Releasing a force will cause the object to immediately return to whatever value it would have had in the design if the force were not present blocking transactions. However, if using the `-keepvalue` option, the forced value will remain on the object until a driver modifies it.

The following are common tasks that you can perform with this command:

- [Forcing Verilog Objects](#)
- [Forcing VHDL Objects](#)
- [Forcing SystemC Objects](#)
- [Listing the Forces to Reapply at Power-Up](#)
- [Forcing the State and Voltage of a Supply Net](#)

force Command Syntax

```
force object_name [=] value
    [-after <time_spec> [value -after <time_spec>...]]
    [-cancel <time_spec>]
    [-delete object_name]
    [-release [-keepvalue] <time_spec>]
    [-repeat <time_spec> [-cancel <time_spec>]]
    [-show [-quiet]
        [-lps]
        [-domain <domain_name>]
        [-pup_on] [-pup_off]]

force -supply_on | -supply_partial_on net_name voltage
    -supply_off net_name
```

force Command Options

This section describes the options that you can use with the Tcl `force` command.

`-after <time_spec>`
`[value -after`
`<time_spec>...]`

Delays the assignment of the new value by the time specified.

You can choose to specify multiple values using the following syntax:

```
force <object> <value> -after <time_spec> <value> -after
<time_spec> ...
```

The time specification (the `time_spec` argument) is relative to the current simulation time. For example, if the current simulation time is 100 ns, and you specify `-after 50 ns`, the assignment takes place at time 150 ns.

`-cancel <time_spec>`

Cancels the assignment of the value after the specified time.

`-delete <object_name>`

Deletes scheduled and current active forces on the specified object.

This command deletes all future forces on the object, as well as the currently active force.

`-release [-keepvalue]
<time_spec>`

Releases a force on the object after the delay specified by the *time_spec* argument.

By default (without the `-keepvalue` option), this option releases the force on the object causing the internal value to take effect immediately. If you choose to include the `-keepvalue` option, the force will remain on the object until a driver modifies it.

The forced value must match the output that will be driven on the signal. Otherwise, the only way to change the output value is to first drive the inputs to the required force.

For instance, consider a two-input AND gate with A1 and A2 as inputs, and Z as the output. If a value of 1 is forced on Z and either A1 or A2 is 0, then you must first force A1 and A2 to 1 to release the forced value on Z. However, when either A1 or A2 is 0, changing both values to 0 will not affect the value of Z in this scenario. It will stay at 1.

`-repeat <time_spec>
[-cancel <time_spec>]`

Repeats the *force* command after the specified time.

The time specification (the *time_spec* argument) is the relative time duration at which to start repeating the cycle of forces.

Example 1: cycle repeats at 100-time units

```
xcelium> force sig 1 -after 10 0 -after 20 -repeat 100
```

Assuming that the current simulation time is 0ns, the following *force* command sets the value of *sig* to 1 at 10 time units after the current simulation time, and then to 0 at 20-time units after the current simulation time. This cycle repeats at 100-time units after the current simulation time so that the next forces are at 110, 120, 210, 220 (and so on) time units after the current simulation time. Here, you can include the `-cancel` option to specify the period at which to stop the cycle:

```
xcelium> force sig 1 -after 10 0 -after 20 -repeat 100 -cancel  
1000
```

Example 2: If there are 2 *force -repeat* commands on the same signal, say, 'r1':

```
xcelium> force r1 1'bz -after 5ns 1'bx -after 10ns -repeat 10  
ns
```

```
xcelium> force r1 1 -after 4 ns 0 -after 8 ns -repeat 10 ns
```

Here, the second *force* command will overwrite the first *force* command. So, 'r1' will change to '1' at 4ns, '0' at 8ns, '1' at 14ns, '0' and 18ns, and so on.

Example 3: `-after` and `-release` at the same time, with `-repeat` on the same object, say, 'r1':

```
xcelium> force r1 1 -after 10ns -release 10 ns
```

Here, the order of (`-after`) *force* and *release* is ambiguous. The behaviour will be undefined and will be accompanied with the *W, AFTREL warning.

| | |
|--|---|
| <code>-show [-quiet]</code> | Lists objects in the design hierarchy that have been explicitly forced to a value. |
| <code>-lps</code> | Lists the forces waiting to be reapplied to a corrupted power domain when the domain is powered up. By default, Xcelium lists all forces at the current simulation time in a low-power design. You can use the following sub-options to modify this behavior: <code>-domain</code> : Specifies a <i>domain_name</i> to narrow the scope to an explicit domain. <code>-pup_on</code> : Enables listing force-related information at power-up. <code>-pup_off</code> : Disables listing force-related information at power-up. |
| <code>-supply_on net_name voltage</code> | Forces the specified <i>voltage</i> and a state of FULL_ON to the supply net specified by <i>net_name</i> . |
| <code>-supply_partial_on net_name voltage</code> | Forces the specified <i>voltage</i> and a state of PARTIAL_ON to the supply net specified by <i>net_name</i> . |
| <code>-supply_off net_name</code> | Forces a state of OFF to the supply net specified by <i>net_name</i> . |

force Command Limitations

The `force -show` specification does not display objects that have been forced to a value with a Verilog procedural `force` continuous assignment by default. You must include the `-show_forces` option on the `xrun` or `xmelab` command line.

The `force -show` specification might not display all the forces applied on multi-dimensional arrays (MDAs) correctly.

force Command Examples

The following examples illustrate use cases for the `force` command:

- [Verilog](#)
- [VHDL](#)
- [Vector Size](#)

Verilog Examples

The following command forces object `r` to the value ``bx`. The equal sign is optional:

```
xcelium> force r = 'bx
```

The following command sets the value of `sig` to 0. The new value takes effect immediately:

```
xcelium> force sig 0
```

The following command delays the assignment of the value 0 to `sig` by 10 ns :

```
xcelium> force sig 0 -after 10 ns
```

The following command forces the value of `sig` to 0 at the current simulation time and then to 1 at 50 ns after the current simulation time. For example, if the current simulation time is 20 ns, `sig` is forced to 0 at time 20 ns, and then to 1 at time 70 ns :

```
xcelium> force sig 0 -after 0 ns 1 -after 50 ns
```

The following force command sets the value of `sig` to 0 at the current simulation time, and then to 1 at 50 ns after the current simulation time. This cycle repeats at 100 ns after the current simulation time:

```
xcelium> force sig 0 -after 0 ns 1 -after 50 ns -repeat 100 ns
```

Assuming that the current simulation time is 0 ns, the value of `sig` changes as follows:

```
0 NS:   sig = 0
50 NS:   sig = 1
100 NS:  sig = 0
150 NS:  sig = 1
200 NS:  sig = 0
250 NS:  sig = 1
...
```

The following command includes the `-cancel` option to cancel the repeating force 300 ns after the current simulation time:

```
xcelium> force sig 0 -after 0 ns 1 -after 50 ns -repeat 100 ns -cancel 300 ns
```

Assuming that the current simulation time is 0 ns, the value of `sig` changes as follows:

```
0 NS:   sig = 0
50 NS:   sig = 1
100 NS:  sig = 0
150 NS:  sig = 1
200 NS:  sig = 0
250 NS:  sig = 1
300 NS:  sig = 0    <- Repeat scheduled for 400 ns is cancelled.
350 NS:  sig = 1
```

The following command releases the force on `sig` 300 ns after the current simulation time:

```
xcelium> force sig 0 -after 0 ns 1 -after 50 ns -release 300 ns
```

```
0 NS:   sig = 0
50 NS:   sig = 1
300 NS:  sig = 0    <- The value on sig at 300 ns, had the force not been applied.
```

The following command specifies that the force on `sig` is to be released 300 ns after the current simulation time. The `-keepvalue` option preserves the forced value after 300 ns until the driver value changes `sig` to 0

at 320 ns:

```
xcelium> force sig 0 -after 0 ns 1 -after 50 ns -release -keepvalue 300 ns
0 NS:    sig = 0
50 NS:    sig = 1
300 NS:  sig = 1    <- Force on sig is released, but forced value still in effect.
320 NS:  sig = 0    <- Driver changes value of sig.
```

The following command specifies that the force on `sig` is to be released 100 ns after the current simulation time. The cycle of forces and release is repeated 500 ns after the current simulation time:

```
xcelium> force sig 0 -after 0 ns 1 -after 50 ns -release 100 ns -repeat 500 ns
0 NS:    sig = 0
50 NS:    sig = 1
100 NS:   sig = 0    <- Force released.
...
...      <- Value of sig changes in response to its drivers.
...
500 NS:   sig = 0    <- sig forced to 0.
550 NS:   sig = 1
600 NS:   sig = 0    <- Force released.
...
...      <- Value of sig changes in response to its drivers.
...
1000 NS:  sig = 0    <- sig forced to 0.
1050 NS:  sig = 1
1100 NS:  sig = 0    <- Force released.
...
```

The following command forces the signal `nickels` (declared as an 8-bit Verilog wire) to the value 2 :

```
xcelium> force test_drink.nickels 2
```

In the above example, `test_drink.nickels` is a compressed Verilog wire. You cannot force a subelement of a compressed Verilog wire unless you have elaborated the design with the `-expand` command-line option. The following example shows the error message that is generated if the design has not been elaborated with `-expand` :

```
xcelium> force test_drink.nickels[2] 0
xcelium: *E,FOCMSB: cannot force a bit-select of a compressed wire: test_drink.nickels[2].
```

The following command uses the `radix#number` syntax to force `nickels` to hexadecimal 1a :

```
xcelium> force test_drink.nickels 16#1a
```

The following command uses the `/` character as the hierarchy separator. Signal `nickels` is forced to binary 11110000 :

```
xcelium> force /test_drink/nickels 2#11110000
```


The following command uses value substitution. Object *x* is forced to the current value of *w* :

```
xcelium> force x = #w
```

The following command uses command substitution and value substitution. Object *y* is forced to the result of the Tcl `expr` command, which evaluates the expression `#r[0] & ~#r[1]` using the current value of *r* :

```
xcelium> force y [expr #r[0] & ~#r[1]]
```

The following command shows the error message that is displayed if you run in regression mode and then use the `force` command on an object that does not have write access:

```
xcelium> force clrb 1
xmsim: *E,OBJACC: Object must have write access: clrb.
```

The following command illustrates the output displayed by a `force -show` command. The command displays a list of objects that are forced, the value to which the signals are forced, and the source, or origination, of the force:

By default, objects that have been forced to a value by a Verilog procedural `force` continuous assignment are not displayed. If you want to include these forced objects, you must compile the source code with the `-linedebug` option (`xmvlog -linedebug`) or elaborate the design with the `-show_forces` option (`xmelab -show_forces` or `xrun -show_forces`). You must also set the value of the `show_force` variable to 1.

```
xcelium> force -show
:s_my_record.rec_int <- 111 ... from xm_force() [ File: ./testbench.vhd, Line: 78 ]
:I2.wout <- 4'hf ... from xm_force() [ File: ./testbench.vhd, Line: 78 ]
:I1:s_stdlogic1 <- '1' ... from TCL
:I1:s_character <- 'b' ... from xm_force() [ File: ./testbench.vhd, Line: 78 ]
:I1:s_stdlogica2 <- "1111" ... from TCL
xcelium>
```

The following `force -show` command includes the `-quiet` option. The forced object names only are listed on a single line:

```
xcelium> force -show -quiet
:s_my_record.rec_int :I2.wout :I1:s_stdlogic1 :I2.in1[3:2] :I1:s_character :I2.reg1
:I1:s_stdlogica2 :I2.bus1[7:2]
```

The following command shows how you can use the output of a `force -show -quiet` command as input for another Tcl command. In this example, all forced objects will be released by the `release` command:

```
xcelium> release [force -show -quiet]
```

VHDL Examples

The following command forces object `:t_nickel_out` (`std_logic`) to 1. The equal sign is optional:

```
xcelium> force :t_nickel_out = '1'
```

The following command forces object `:top:DISPENSE_tempsig` (`std_logic`) to 1:

```
xcelium> force :top:DISPENSE_tempsig '1'
```

The following command forces object `:t_DIMES` (`std_logic_vector`) to FF (hex):

```
xcelium> force :t_DIMES X"FF"
```

The following command forces a subelement of the vector `:t_DIMES` to 1:

```
xcelium> force :t_DIMES[2] {'1'}
```

The following command uses the `radix#number` syntax to force `t_DIMES` to hexadecimal 1a:

```
xcelium> force :t_DIMES 16#1a
```

The following command uses the `/` character as the hierarchy separator. `test_drink` is the top-level entity name. Signal `t_DIMES` is forced to binary 11110000:

```
xcelium> force /test_drink/t_DIMES 2#11110000
```

The following command forces object `is_ok` (boolean) to TRUE:

```
xcelium> force :is_ok true
```

The following command forces object `:count` (integer) to 10:

```
xcelium> force :count 10
```

The following command shows the error message that is displayed if you run in regression mode and then use the `force` command on an object that does not have write access:

```
xcelium> force :exsum {"11111111"}
```

```
xmsim: *E,OBJACC: Object must have write access: :exsum.
```

The following command delays the assignment of the value 1 to `:sig` by 10 ns:

```
xcelium> force :sig ' 1 ' -after 10 ns
```

The following command forces the value of `:sig` to 1 at the current simulation time and then to 0 at 50 ns after the current simulation time. For example, if the current simulation time is 20 ns, `:sig` is forced to 1 at time 20 ns, and then to 0 at time 70 ns:

```
xcelium> force :sig ' 1 ' -after 0 ns ' 0 ' -after 50 ns
```

The following force command sets the value of `:sig` to 1 at the current simulation time, and then to 0 at 50 ns after the current simulation time. This cycle repeats at 100 ns after the current simulation time:

```
xcelium> force :sig ' 1 ' -after 0 ns ' 0 ' -after 50 ns -repeat 100 ns
```

Assuming that the current simulation time is 0 ns, the value of `:sig` changes as follows:

```
0 NS:    :sig = '1'
50 NS:    :sig = '0'
100 NS:   :sig = '1'
150 NS:   :sig = '0'
200 NS:   :sig = '1'
250 NS:   :sig = '0'
...
```

The following command includes the `-cancel` option to cancel the repeating force 300 ns after the current simulation time:

```
xcelium> force :sig ' 1 ' -after 0 ns ' 0 ' -after 50 ns -repeat 100 ns -cancel 300 ns
```

Assuming that the current simulation time is 0 ns, the value of `:sig` changes as follows:

```
0 NS:    :sig = '1'
50 NS:    :sig = '0'
100 NS:   :sig = '1'
150 NS:   :sig = '0'
200 NS:   :sig = '1'
250 NS:   :sig = '0'
300 NS:   :sig = '1'    <- Repeat scheduled for 400 ns is cancelled.
350 NS:   :sig = '0'
```

The following command releases the force on `:sig` 300 ns after the current simulation time:

```
xcelium> force :sig ' 1 ' -after 0 ns ' 0 ' -after 50 ns -release 300 ns
0 NS:    :sig = '1'
50 NS:    :sig = '0'
300 NS:   :sig = '1'    <- The value on :sig at 300 ns, had the force not been applied.
```

The following command specifies that the force on `:sig` is to be released 300 ns after the current simulation time. The `-keepvalue` option preserves the forced value after 300 ns until the driver value changes `:sig` to 1 at 320 ns:

```
xcelium> 1 ' -after 0 ns ' 0 ' -after 50 ns -release -keepvalue 300 ns
0 NS:    :sig = '1'
50 NS:    :sig = '0'
300 NS:   :sig = '0'    <- Force on :sig is released, but forced value still in effect.
320 NS:   :sig = '1'    <- Driver changes value of :sig.
```

The following command specifies that the force on `:sig` is to be released 100 ns after the current simulation time. The cycle of forces and release is repeated 500 ns after the current simulation time:

```
0 NS:    :sig = '1'
50 NS:    :sig = '0'
100 NS:   :sig = '1'    <- Force released.
...
...                <- Value of :sig changes in response to its drivers.
```

```
...
500 NS:  :sig = '1'    <- :sig forced to '1'.
550 NS:  :sig = '0'
600 NS:  :sig = '1'    <- Force released.
...
...      <- Value of :sig changes in response to its drivers.
...
1000 NS: :sig = '1'    <- :sig forced to '1'.
1050 NS: :sig = '0'
1100 NS: :sig = '1'    <- Force released.
...
```

The following command illustrates the output displayed by a `force -show` command. The command displays a list of objects that are forced, the value to which the signals are forced, and the source, or origination, of the force:

By default, objects that have been forced to a value by a Verilog procedural `force` continuous assignment are not displayed. If you want to include these forced objects, you must compile the source code with the `-linedebug` option (`xmvlog -linedebug`) or elaborate the design with the `-show_forces` option (`xmelab -show_forces` or `xrun -show_forces`). You must also set the value of the `show_force` variable to 1.

```
xcelium> force -show
:s_my_record.rec_int <- 111 ... from xm_force() [ File: ./testbench.vhd, Line: 78 ]
:I2.wout             <- 4'hf ... from xm_force() [ File: ./testbench.vhd, Line: 78 ]
:I1:s_stdlogic1      <- '1' ... from TCL
:I1:s_character       <- 'b' ... from xm_force() [ File: ./testbench.vhd, Line: 78 ]
:I1:s_stdlogica2     <- "1111" ... from TCL
xcelium>
```

The following `force -show` command includes the `-quiet` option. The forced object names only are listed on a single line:

```
xcelium> force -show -quiet
:s_my_record.rec_int :I2.wout :I1:s_stdlogic1 :I2.in1[3:2] :I1:s_character :I2.reg1
:I1:s_stdlogica2 :I2.bus1[7:2]
```

The following command shows how you can use the output of a `force -show -quiet` command as input for another Tcl command. In this example, all forced objects will be released by the `release` command:

```
xcelium> release [force -show -quiet]
```

Vector Size Mismatches

If the specified value is less than the declared width, a vector size mismatch warning is issued, and the right-most bits (LSB) of the value are applied to the right-most (LSB) bits of the vector, and the MSB bits of the object are unchanged. For example:

```
signal sig : std_logic_vector(15 downto 0);
xcelium> value sig
"UUUUUUUUUUUUUUUUUU"
xcelium> force sig X"FF"
xmsim: *W,SEBNDP: Vector size mismatch: "11111111", MSB bits of object unchanged.
xcelium> value sig
"UUUUUUUUU11111111"
```

If the value being deposited is wider than the declared width, a vector size mismatch warning is issued, and the MSB bits of the value are truncated. For example:

```
xcelium> value sig
"UUUUUUUUUUUUUUUUUU"
xcelium> force sig 16#F0000
xmsim: *W,SEBNDT: Vector size mismatch: "11110000000000000000", MSB bits of value truncated.
xcelium> value sig
"00000000"
```

Forcing Verilog Objects

When using the `force` command on a Verilog object, the `object_name` specification must be a literal.

The object being forced cannot be:

- A Verilog memory
- A Verilog memory element
- A bit-select or part-select of a Verilog register

Xcelium will treat the literal as a constant, even if the literal is generated using [value substitution](#) or Tcl's `expr` command. Additionally, if the object used to generate the literal changes value during a subsequent simulation, the forced value will not change.

Syntax

This section lists the command syntax to `force` Verilog Objects.

```
force object_name [=] value [other_options]
```


Examples

The following commands illustrate how you can `force` values to vectors in binary, octal, decimal, or hexadecimal formats using standard Verilog notation.

```
wire [7:0] sig;  
xcelium> force sig = 8'hff  
xcelium> force sig = 8'b00000000  
xcelium> force sig = 8'd10
```

The following commands use language-independent `radix#number` syntax for specifying the value.

```
force sig 2#10111011  
force sig 8#17  
force sig 10#13  
force sig 16#cf
```

 The `tcl_relaxed_literal` variable must be set to 1 to enable this functionality.

If no radix is specified, decimal is the default format. For example:

```
xcelium> force sig 5
```

Forcing VHDL Objects

When using the `force` command on a VHDL object, the `object_name` specification must match the type and subtype constraints of that VHDL object. Integers, reals, physical types, enumeration types, and strings (including `std_logic_vector` and `bit_vector`) are supported.

The object being forced cannot be a VHDL variable. Additionally, records and non-character array values are not supported, but objects of these types can be assigned to by issuing commands for each subelement individually.

Syntax

This section lists the command syntax to `force` VHDL objects.

```
force object_name [=] value [other_options]
```

Examples

The following commands illustrate how you can `force` values to VHDL vectors by specifying the value for each bit:

```
signal sig : std_logic_vector(15 downto 0);  
xcelium> force :sig = {"1111111111111111"}
```

Alternatively, it may be easier and more intuitive to force a VHDL vector using either bit string literal or language-independent syntax.

- Bit string literal syntax is `base_specifier"bit_value"` where:
 - `base_specifier` can be `b` or `B`, `o` or `O`, or `x` or `X`. If no base is specified, the value is assumed to be binary.
 - `bit_value` specifies the bit-string literal in the appropriate base. If a base specifier is not specified, the value should be enclosed in curly braces or escape characters. Underscore characters can be included to improve readability.
- Language-independent syntax is `radix#number`; however, base 10 is not supported when using this syntax with VHDL.

 The `tcl_relaxed_literal` variable must be set to 1 to enable the `radix#number` functionality.

The following commands illustrate how you can use bit string literal syntax:

```
signal sig : std_logic_vector(15 downto 0);
xcelium> force sig B"1111111111111111"
xcelium> force sig b"00000000_11111111"      Can use underscore characters.
xcelium> force sig {"11111111_11111111"}      No base_specifier. Value is binary.
                                              Curly braces or escape characters are required.

xcelium> force sig O"76543"
xcelium> force sig X"ffff"
xcelium> force sig x"FF_FF"
xcelium> force sig x"FF_FF"
```

When using 9-state logic values (-, U, 1, 0, Z, X, L, H, W), each value is expanded to four binary bits if the base is hex and to three binary bits if the base is octal:

```
xcelium> force sig X"FZHA"
xcelium> value sig
"1111ZZZZHHHH1010"
```

The following commands use language-independent syntax for specifying the value:

```
force sig 2#1111000011110000
force sig 8#17
force sig 16#abcd
```

Forcing SystemC Objects

When using the `force` command on a SystemC object, the `object_name` specification can match to one of the following types:

- `sc_signal`

- `sc_clock`
- `sc_in`
- `sc_out`
- `sc_inout`
- `xmsc_register`

For the objects in this list, the `force` command is supported when the object is templated by a C++ data type, by a SystemC built-in data type, or by a user-defined data type that implements the `from_string` virtual function.

Alternatively, for `sc_port` and `sc_export` objects, this command is supported if the `sc_object` is bound to the port or the export supports the force.

If an arbitrary SystemC object declares support for forcing a value (the member method `xmsc_supports_force()`), you can apply the `force` and `deposit -force` commands to that object.

Related Topic

- [Specifying Supported Commands](#)

Listing the Forces to Reapply at Power-Up

Use the `-lps` option with the `force` command during low-power simulation to display the list of forces waiting to be reapplied to a corrupted power domain when the domain is powered up. This list includes those forces that exist at the time of power domain corruption and those that change during power domain corruption.

The origination can be:

- A Tcl `force` command
- A VPI force (`vpi_put_value()` with a `vpiForceFLAG` flag)
- A VHPI force (`vhpi_put_value` with an update mode of `vhpiForce` or `vhpiForcePropagate`)
- An `xm_force` call
- A Verilog procedural `force` continuous assignment

If a Verilog procedural `force` continuous assignment is an expression, the expression will not be displayed with the Tcl command `force -lps`, but the software will output the file and line number of the force.

To reapply constant forces at power-up, use the command-line option `-lps_force_reapply`. To reapply Verilog forces that are expressions, you must also specify the option `-lps_expr_force_reapply`. For more information, see the *Low-Power Simulation Guide (IEEE 1801)*.

Syntax

This section lists the `force` command syntax to list the forces waiting to be reapplied at power-up.

```
force -lps [-show [-quiet]]  
          [-domain <domain_name>] [-pup_on] [-pup_off]
```

Options

| | |
|-----------------------------|---|
| <code>-show [-quiet]</code> | Lists objects in the design hierarchy that have been explicitly forced to a value. |
| <code>-lps</code> | Lists the forces waiting to be reapplied to a corrupted power domain when the domain is powered up. By default, Xcelium lists all forces at the current simulation time in a low-power design. You can use the following sub-options to modify this behavior: <code>-domain</code> : Specifies a <code>domain_name</code> to narrow the scope to an explicit domain. <code>-pup_on</code> : Enables listing force-related information at power-up. <code>-pup_off</code> : Disables listing force-related information at power-up. |

Forcing the State and Voltage of a Supply Net

Use the Tcl `force` command to set the state and voltage values of a given supply net. Depending on which options you choose, you can set a FULL_ON, PARTIAL_ON, or OFF state for the specified net. Once the supply net is forced, the simulator does not change the state and voltage values until either another force command updates the values or the force is released.

Syntax

```
force -supply_on | -supply_partial_on net_name voltage  
      -supply_off net_name
```

Options

| | |
|--|--|
| <code>-supply_on net_name voltage</code> | Forces the specified <code>voltage</code> and a state of FULL_ON to the supply net specified by <code>net_name</code> . |
| <code>-supply_partial_on net_name voltage</code> | Forces the specified <code>voltage</code> and a state of PARTIAL_ON to the supply net specified by <code>net_name</code> . |
| <code>-supply_off net_name</code> | Forces a state of OFF to the supply net specified by <code>net_name</code> . |

Limitations

The following limitations apply when forcing the state and voltage of a supply net:

- One force command can set the state and voltage for one specified *net_name* only. Having multiple *net_names* is not supported.
- The `-supply_on` and `-supply_partial_on` options require both the *net_name* and *voltage* arguments. A missing voltage value results in an LPTSNNOV warning.
- Using other force options with `-supply_on`, `-supply_partial_on`, or `-supply_off` is not supported. Including other force options results in a DELPSNNOV error.

Examples

A supply net `tb.dut.t1.VDD` is forced to a `FULL_ON` state with a voltage of 1.2 until the Tcl `release` command removes it.

```
force -supply_on tb.dut.t1.VDD 1.2
```

A supply net `tb.dut.t1.VDD` is forced to a `PARTIAL_ON` state with a voltage of 0.8 until the Tcl `release` command removes it.

```
force -supply_partial_on tb.dut.t1.VDD 0.8
```

A supply net `tb.dut.t1.VDD` is forced to an `OFF` state with a voltage of 0 until the Tcl `release` command removes it.

```
force -supply_off tb.dut.t1.VDD
```

In the example below, the first `force` command sets a `FULL_ON` state with a voltage of 1.2 for the supply net `tb.dut.t1.VDD`; however, the second command updates this force and changes the state to `OFF` with a voltage of 0. The updated values apply until the force is released.

```
force -supply_on tb.dut.t1.VDD 1.2  
force -supply_off tb.dut.t1.VDD
```

fs_strobe

The Tcl `fs_strobe` command specifies one or more signals to monitor as strobe points during a fault campaign. Normally, this command is run using a Tcl script executed with the `-input` option on the `xrun` or `xmsim` command line. The Tcl script can include multiple strobe specifications, defining a complete hardware strobe list for your campaign run. Alternatively, you can invoke `fs_strobe` directly from the `xcelium>` prompt. With defined hardware strobes, you can:

- Monitor each and every signal value change for an entire simulation. The default.
- Monitor only those signal value changes recorded using a stop trigger condition.

The data generated by this command helps to determine if an injected fault is DETECTED, POTENTIALLY DETECTED, or UNDETECTED at one or more of the specified nets. Optionally, the `fs_strobe` command can be used to mark strobe points as functional and checker outputs, which allow for the following additional fault classifications: DANGEROUS_DETECTED, DANGEROUS_UNDETECTED, SAFE_DETECTED, and SAFE_UNDETECTED.

The following are common tasks that you can perform using this command:

- [Defining a single strobe list](#)
- [Defining a dual strobe list](#)
- [Setting a stop on condition](#)

fs_strobe Command Syntax

```
fs_strobe list_of_signals
    -checker {strobe_list}
    -functional {strobe_list}
    -checker_delay_window <time_spec> [-absolute | -relative]
    -mapping <pessimistic | optimistic>
    -signal_change
    -single_value
    -stop_on <detected | potentially_detected | none>
```

fs_strobe Command Options

This section describes the options that you can use with the Tcl `fs_strobe` command.

| | |
|--|--|
| <code>-checker {strobe_list}</code> | Specifies a list of strobe points to monitor as checker outputs. |
| <code>-functional {strobe_list}</code> | Specifies a list of strobe points to monitor as functional outputs. |
| <code>-checker_delay_window <time_spec></code> | <p>Specifies the amount of time in which to monitor the checker outputs after an injected fault is detected at one of the functional outputs.</p> <p>The simulator stops monitoring the checker outputs once it reaches the time specified, even if no fault is detected. Specify a time of 0 to enable the simulator to monitor checker outputs at the same time a fault is detected for a functional output. Use one of the following sub-options to explicitly categorize the time specified:</p> <ul style="list-style-type: none">• -relative: Specifies a time relative to the current simulation time. The default.• -absolute: Specifies an absolute time for simulation. |

`-mapping <pessimistic | optimistic>`

Specifies the method for mapping potentially detected faults (`optimistic` or `pessimistic`). The default method is `pessimistic`.

This option is a global setting and applies to the strobe points in multiple `fs_strobe` commands. If you specify this option more than once, the software generates an `FSTMVW` warning and uses the first setting specified.

`-signal_change`

Records the value of the specified signal whenever that value changes. This is the default behavior.

The Xcelium Fault Simulator supports multiple conditional strobes: methods for monitoring one or more nets (or strobe points) during a fault simulating run. Use this option to explicitly specify continuous fault sampling, and monitor each and every signal value change for an entire simulation.

`-single_value`

Records the value of the specified signal at execution time. This fault strobing behavior can be helpful when creating Tcl `stop` conditions.

Use this option to explicitly specify a single sample value, monitoring the value change recorded using a Tcl `stop` trigger condition. Note that this is preferred whenever the strobe points specified to the `fs_strobe` command are not sequential elements or signals, since it can help avoid race conditions.

`-stop_on`

Sets the stop severity for the fault run, during the good simulation run, by specifying one of the following arguments:

- **detected:** Stops the simulation whenever a fault is detected at a strobe point. This is the default.

If the `-functional` and `-checker` options are also specified, the simulation stops when a fault is detected at a checker output. If only the `-functional` option is specified, then the simulation stops when a fault is detected at a functional output.

- **potentially_detected:** Stops the simulation whenever a fault is potentially detected at one of the specified strobe points.
- **none:** Completes simulation regardless of whether a fault is detected or potentially detected. If specified with `-functional`, this applies only to functional strobes. If specified with `-checker`, or both `-functional` and `-checker`, this applies only to checker strobes.

fs_strobe Command Examples

Example 1

The following command strobos the signal `top.dut_inst.out` for the entire simulation. All signal value changes are saved to an SHM database in the `fault_db` directory, or in the directory specified by the `-fault_work` option on the `xrun` or `xmsim` command line:

```
xcelium> fs_strobe top.dut_inst.out
```

Example 2

The following command strobos `top.dut_inst.out2` and `top.checker_inst.out` respectively as functional and checker outputs. The simulator monitors the checker output for 100ns after the injected fault is detected at the functional output. If the fault is also detected at the checker output in this time window, the simulation stops (this is the default behavior).

```
xcelium> fs_strobe -functional {top.dut_inst.out2} -checker {top.checker_inst.out} -  
checker_delay_window 100ns
```

Example 3

The following command sets up functional and checker strobos in multiple commands. A `-stop_on` condition is specified for each strobe list. If an injected fault is detected in the first `fs_strobe` command, the simulation does not stop and continues until the end of the simulation or until the fault is detected at `top.checker_inst.out2` in the second `fs_strobe` command:

```
xcelium> fs_strobe -functional {top.dut_inst.out} -checker {top.checker_inst.out} -stop_on  
none  
xcelium> fs_strobe -functional {top.dut_inst.out2} -checker {top.checker_inst.out2} -stop_on  
detected
```

Example 4

The following command shows one method of strobing a signal by using the Tcl `stop` command and setting a control on the posedge of `tb.clk`. By default, the control condition is used once to enable fault strobing and is ignored thereafter. Each time the value of `top.dut_inst.out` changes, that value is recorded for the circuit. Cadence recommends using the `-silent` option to avoid undue verbosity in the log file.

```
xcelium> stop -object tb.clk -if {#tb.clk == 1'b1} -execute {fs_strobe top.dut_inst.out -  
signal_change} -silent -delbreak 1
```

Example 5

The following command enables the simulator to record the value of `top.dut_inst.out`, but only when the specified control condition is TRUE:

```
xcelium> stop -object tb.clk -if {#tb.clk == 1'b1} -execute {fs_strobe top.dut_inst.out -  
single_value} -silent -continue
```

Setting a Stop Trigger Condition to Monitor Signal Value Changes

The following command shows one method of strobing a signal by using the Tcl `stop` command and setting a control on the posedge of `tb.clk`. By default, the control condition is used once to enable fault strobing and is ignored thereafter. Each time the value of `top.dut_inst.out` changes, that value is recorded for the circuit. Cadence recommends using the `-silent` option to avoid undue verbosity in the log file.

```
xcelium> stop -object tb.clk -if {#tb.clk == 1'b1} -execute {fs_strobe top.dut_inst.out -  
signal_change} -silent -delbreak 1
```

The following command enables the simulator to record the value of `top.dut_inst.out`, but only when the specified control condition is TRUE:

```
xcelium> stop -object tb.clk -if {#tb.clk == 1'b1} -execute {fs_strobe top.dut_inst.out -  
single_value} -silent -continue
```

Related Topic

- [Fault Strobing](#)

Defining a Single Strobe List

Define a single strobe list with the `fs_strobe` command using one or more signals to monitor as strobe points.

This enables the fault simulator to generate the following fault classifications during a campaign run:

| | |
|----------------------|--|
| DETECTED | Indicates that a signal value change for the specified strobe point was recorded during the good run and a <i>mismatched</i> value was compared during the fault run (or vice versa). |
| POTENTIALLY_DETECTED | Indicates that a signal value change for the specified strobe point was recorded during the good run and an <i>unconfirmed</i> value (X, Z, U, H, or L) was compared during the fault run (or vice versa). |
| UNDETECTED | Indicates that a signal value for the specified strobe point was recorded during the good run and the <i>same</i> value was compared during a fault run. |

Syntax

This section lists the `fs_strobe` command syntax to define a single strobe list.

```
fs_strobe list_of_signals  
-mapping <pessimistic | optimistic>
```

-signal_change
-single_value

Options

-mapping <pessimistic |
optimistic>

Specifies the method for mapping potentially detected faults (`optimistic` or `pessimistic`). The default method is `pessimistic`.

This option is a global setting and applies to the strobe points in multiple `fs_strobe` commands. If you specify this option more than once, the software generates an `FSTMVW` warning and uses the first setting specified.

-signal_change

Records the value of the specified signal whenever that value changes. This is the default behavior.

The Xcelium Fault Simulator supports multiple conditional strobes: methods for monitoring one or more nets (or strobe points) during a fault simulating run. Use this option to explicitly specify continuous fault sampling, and monitor each and every signal value change for an entire simulation.

-single_value

Records the value of the specified signal at execution time. This fault strobing behavior can be helpful when creating Tcl `stop` conditions.

Use this option to explicitly specify a single sample value, monitoring the value change recorded using a Tcl stop trigger condition. Note that this is preferred whenever the strobe points specified to the `fs_strobe` command are not sequential elements or signals, since it can help avoid race conditions.

Example

The following command strobes the signal `top.dut_inst.out` for the entire simulation. All signal value changes are saved to an SHM database in the `fault_db` directory, or in the directory specified by the `-fault_work` option on the `xrun` or `xmsim` command line:

```
xcelium> fs_strobe top.dut_inst.out
```

Defining a Dual Strobe List

Define a dual strobe list with the `fs_strobe` command by specifying one or more strobe points as functional and checker outputs.

This enables the fault simulator to generate additional fault classifications during a campaign run:

DANGEROUS_DETECTE Faults that are detected on both functional and checker outputs.
D

| | |
|-----------------------|---|
| DANGEROUS_UNDETECTED | Faults that are detected on functional outputs but are not detected on checker outputs. |
| UNOBSERVED_DETECT | Faults that are not detected on functional outputs but are detected on checker outputs. |
| UNOBSERVED_UNDETECTED | Faults that are not detected for either functional or checker outputs. |

Syntax

This section lists the `fs_strobe` command syntax to define a dual strobe list.

```
fs_strobe -checker {strobe_list}
          -functional {strobe_list}
          -checker_delay_window <time_spec> [-absolute | -relative]
          -mapping <pessimistic | optimistic>
          -signal_change
          -single_value
```

Options

| | |
|--|---|
| <code>-checker {strobe_list}</code> | Specifies a list of strobe points to monitor as checker outputs. |
| <code>-functional {strobe_list}</code> | Specifies a list of strobe points to monitor as functional outputs. |
| <code>-checker_delay_window <time_spec></code> | <p>Specifies the amount of time in which to monitor the checker outputs after an injected fault is detected at one of the functional outputs.</p> <p>The simulator stops monitoring the checker outputs once it reaches the time specified, even if no fault is detected. Specify a time of 0 to enable the simulator to monitor checker outputs at the same time a fault is detected for a functional output. Use one of the following sub-options to explicitly categorize the time specified:</p> <ul style="list-style-type: none"> • -relative: Specifies a time relative to the current simulation time. The default. • -absolute: Specifies an absolute time for simulation. |

`-mapping <pessimistic | optimistic>`

Specifies the method for mapping potentially detected faults (`optimistic` or `pessimistic`). The default method is `pessimistic`.

This option is a global setting and applies to the strobe points in multiple `fs_strobe` commands. If you specify this option more than once, the software generates an `FSTMVW` warning and uses the first setting specified.

`-signal_change`

Records the value of the specified signal whenever that value changes. This is the default behavior.

The Xcelium Fault Simulator supports multiple conditional strobes: methods for monitoring one or more nets (or strobe points) during a fault simulating run. Use this option to explicitly specify continuous fault sampling, and monitor each and every signal value change for an entire simulation.

`-single_value`

Records the value of the specified signal at execution time. This fault strobing behavior can be helpful when creating Tcl `stop` conditions.

Use this option to explicitly specify a single sample value, monitoring the value change recorded using a Tcl stop trigger condition. Note that this is preferred whenever the strobe points specified to the `fs_strobe` command are not sequential elements or signals, since it can help avoid race conditions.

Example

The following command strobes `top.dut_inst.out2` and `top.checker_inst.out` respectively as functional and checker outputs. The simulator monitors the checker output for **100ns** after the injected fault is detected at the functional output. If the fault is also detected at the checker output in this time window, the simulation stops (this is the default behavior).

```
xcelium> fs_strobe -functional {top.dut_inst.out2} -checker {top.checker_inst.out} -  
checker_delay_window 100ns
```

Setting a Stop On Condition

Use the `-stop_on` option with the `fs_strobe` command to stop a simulation when a fault is `DETECTED` or `POTENTIALLY_DETECTED` at a particular strobe point.

Syntax

This section lists the `fs_strobe` command syntax to set a stop on condition.

```
fs_strobe [other_options]  
-stop_on <detected | potentially_detected | none>
```

Options

`-stop_on`

Sets the stop severity for the fault run, during the good simulation run, by specifying one of the following arguments:

- **detected:** Stops the simulation whenever a fault is detected at a strobe point. This is the default.

If the `-functional` and `-checker` options are also specified, the simulation stops when a fault is detected at a checker output. If only the `-functional` option is specified, then the simulation stops when a fault is detected at a functional output.
- **potentially_detected:** Stops the simulation whenever a fault is potentially detected at one of the specified strobe points.
- **none:** Completes simulation regardless of whether a fault is detected or potentially detected. If specified with `-functional`, this applies only to functional strobes. If specified with `-checker`, or both `-functional` and `-checker`, this applies only to checker strobes.

Examples

The following is an example of setting a stop on condition for fault simulation. It sets up functional and checker strobes in multiple commands. A `-stop_on` condition is specified for each strobe list. If an injected fault is detected in the first `fs_strobe` command, the simulation will not stop and will continue until the end of the simulation or until the fault is detected at `top.checker_inst.out2` in the second `fs_strobe` command:

```
xcelium> fs_strobe -functional {top.dut_inst.out} -checker {top.checker_inst.out} -stop_on none
xcelium> fs_strobe -functional {top.dut_inst.out2} -checker {top.checker_inst.out2} -stop_on detected
```

glitch

The Tcl `glitch` command provides the capability to enable or disable the glitch detector functionality during simulation. The default state of 'glitch' is `on` (given that Glitch Detector is enabled with `xrun/xmelab` option `ENGATEGLITCH`).

glitch Command Syntax

`glitch`

```
[-on]  
[-off]  
-dump [-overwrite] <filename>
```

glitch Command Options

This section describes the options that you can use with the Tcl `glitch` command.

| | |
|---|---|
| <code>-on</code> | Enables the glitch detector. |
| <code>-off</code> | Disables the glitch detector. |
| <code>-dump [-overwrite]</code> <code><i><filename></i></code> | <p>Dumps the glitch data in a file with the provided file name. The glitch must not be in the off state (<code>glitch -off</code>) for this option to work.</p> <p>The tool throws an error <i>GLTCHFER</i>, if:</p> <ul style="list-style-type: none">the provided <i><filename></i> already exists; the logs are dumped in the previous <i><filename></i>. You can use the <code>glitch -dump -overwrite <i><filename></i></code> command to clear the existing file and start dumping in the file.the <i><filename></i> is not specified with the <code>glitch -dump</code> commandthe <i><filename></i> could not be opened |

Related Topics

- [-engateglitch](#)
- [-enallateglitch](#)

heap

The Tcl `heap` command provides information on objects allocated on the heap. Objects are allocated onto the heap when the SystemVerilog `new` function is used to allocate storage for a dynamic object, such as a class object.

heap Command Syntax

```
heap
  -gc
  -report
    -append <file>
    -distribution
    -limit <decimal>
    -redirect <file>
    -reference <object>
    -size
      -handle
      -inclusive
      -top <decimal>
      -type <code>
    -track_id <@input_heap_id> [-verbose]
    -type <code>
  -show
  -verbose
```

heap Command Options

This section describes the options that you can use with the Tcl `heap` command.

| | |
|---|--|
| <code>-gc</code> | <p>Starts heap garbage collection.</p> <p>The <code>heap -gc</code> command lets you run garbage collection on the heap at any time.</p> <p>You can also control when garbage collection is initiated by setting the following predefined Tcl variables:</p> <ul style="list-style-type: none">• <code>heap_garbage_size</code>• <code>heap_garbage_time</code>• <code>heap_garbage_check</code> |
| <code>-report [-redirect file -append file]</code> | <p>Reports information about heap usage during simulation.</p> <p>You can use the following options with the <code>heap -report</code> command to generate different reports:</p> <ul style="list-style-type: none">• -append: Similar to the <code>-redirect</code> option, but appends the data to the report file, if it exists. If a report file does not exist, it creates one. The appended report data is timestamped with the current simulation time.• -distribution: Generates an object distribution report that shows how many classes, queues, and strings are currently allocated on the heap. This is the default. |

- **-limit *n***: Limits the number of items to be reported on to *n*, where *n* is an unsigned decimal integer. Use this option to override the default limit (10000) for flexibility on the amount of information generated by the `-reference` and `-type` subcommand.
- **-redirect**: Specifies the path to the file where the report is written. If the file does not exist, it is created; if the file exists, it is overwritten. If you do not specify this option, the report is written to the standard output device. The report data is timestamped with the current simulation time.
- **-reference *class_object***: Reports references to and from the specified class object. You can specify multiple class objects. The default limit on the number of objects reported by this subcommand is set to 10000.

- **-size [-handle|-inclusive|-top|-type]**: Displays the list of all heap objects sorted by size. The object with the maximum size is reported first.

You can use the following options with the `-size` option:

- **-handle**: Gives the size ordered list of heap objects. It prints the heap id, followed by the handle description and then the size.
 - **-inclusive**: Gives the size ordered list of heap objects and includes the size of the nested QDAs within the class object.
 - **-top**: Gives the size ordered list of the top <decimal> heap objects.
 - **-type**: Gives the sizes of all objects of a given type.
- **-track_id <@input_heap_id> [-verbose]**: Traverses the design hierarchy and finds the first reference of the specified *input_heap_id* held in another heap object and prints the heap id of the referring heap object. If the *input_heap_id* is a root level heap id, a suitable message is printed.

You can use the optional `-verbose` option with `-track_id`. If the referring heap id is a class object, this option prints the details of that class object in a similar way as that of the `describe` Tcl command. No details are printed if the referring heap id is not a class object.

- **-type *object_type***: Reports the names, sizes, and reference counts of all objects of a given type. The *object_type* argument specifies the type of heap object about which you want to produce a report. This option reports the definitions used, the number of instances of each definition, and the memory footprint of each instance.

The *object_type* argument can be one or more of the following characters:

- **s** String
- **e** Event
- **g** Covergroup
- **a** Associative array
- **q** Queue
- **d** Dynamic array
- **c** Class
- **v** Virtual interface
- **m** Mailbox
- **4** Semaphore

The default limit on the number of objects reported by this subcommand is set to 10000.

-show

Displays the current heap content.

The `heap -show` command displays information about objects in the heap, including the number of objects in the heap, their allocated handles, and heap system parameters (garbage collection size policy and time policy).

-verbose

The `-verbose` option displays additional information, such as where the object was allocated, the size and type of the object, and its current value.

heap Command Examples

The examples in this section use the following HDL code:

```
module top;
  int myq[$];
  int queueSize;

  class c1;
    real r;
    byte u;
    byte u1;
    real r1;
    integer p1;
  endclass

  class c2;
    real r;
    integer foo;
    integer p1;
    integer p2;
  endclass

  reg [63:0] addr[];
  c1 pc1, pc12, pc13;
  c2 pc2;
  integer i1;

  initial
    begin
      #100;
      pc1 = new;
      pc2 = new;
      pc12 = pc1;
      pc13 = pc12;
      queueSize = 10;
    end

  task queueInit(input int qsiz);
    for (int j = 0; j <= qsiz; j++)
      begin
        myq.push_back( j );
      end
  endtask

endmodule
```

The following are examples of using the `-heap` command options.

```
xcelium> heap -size
```

```
No object allocated on heap  # Does not return any information because objects
```

```

                                # not allocated on the heap at start of simulation.
xcelium>
xcelium> run 120 ns
xcelium> heap -size
2 objects allocated on heap      # Number of objects on the heap
128 total storage bytes         # Size in bytes of allocated data
xcelium> heap -show
2 objects allocated on heap
User Allocated Handles: 5 , 6   # Returns handles 3 and 4, because the first two
                                # handles were internally allocated.

Heap System Parameters:
  Garbage collection size policy (%)= -200
  Garbage collection time policy (sec) = (default)
xcelium> heap -show -verbose
2 objects allocated on heap
User Allocated Handles: 5 , 6
5.....handle class top.c1{
    real r
    byte u
    byte u1
    real r1
    integer p1
}
Object size = 64 bytes
Handle has 3 references

6.....handle class top.c2 {
    real r
    integer foo
    integer p1
    integer p2
}
Object size = 64 Bytes
Handle has 1 reference

Heap System Parameters:
  Garbage collection size policy (%)= -200
  Garbage collection time policy (sec) = (default)
xcelium>
xcelium> # Generate an object distribution report that shows how many
xcelium> # classes, queues, and strings are currently allocated on the heap.
xcelium> heap -report -distribution
  Dynamic Array: 1 [ 32 bytes ]
    Queue: 1 [ 80 bytes ] including 1 prototypes and 0 elements
    Class: 2 [ 128 bytes ]
      + ---
```



```
total objects: 6 [ 240 bytes ]
xcelium> # Report names, sizes, and reference counts of all objects of type class.
xcelium> heap -report -type c
2 objects of type : Class [ 128 bytes ]
Index - Datatype - Hierarchical Pathname
5:handle class top.c1 top.pc1 [ 64 bytes ]
   handle class top.c1 top.pc12 [ 64 bytes ]
   handle class top.c1 top.pc13 [ 64 bytes ]
6:handle class top.c2 top.pc2 [ 64 bytes ]
xcelium> heap -report -type cq      # Specify more than one report type.
2 objects of type : Class [ 128 bytes ]
Index - Datatype - Hierarchical Pathname
5:handle class top.c1 top.pc1 [ 64 bytes ]
   handle class top.c1 top.pc12 [ 64 bytes ]
   handle class top.c1 top.pc13 [ 64 bytes ]
6:handle class top.c2 top.pc2 [ 64 bytes ]
1 object of type : Queue [ 80 bytes ]
Index - Datatype - Hierarchical Pathname
3: int queue top.myq [ 80 bytes ] 0 elements
xcelium> # Report references for a particular class object.
xcelium> heap -report -reference pc1
References to top.c1@5_1:
top.pc1
top.pc12
top.pc13
xcelium> heap -report -reference pc1 pc2    # Can specify multiple class objects.
References to top.c1@5_1:
top.pc1
top.pc12
top.pc13
References to top.c2@6_1:
top.pc2
xcelium> # To generate a report about references for all class objects
xcelium> # on the heap, use the value -classlist command as an argument to -reference
xcelium> heap -report -reference [value -classlist]
References to top.c1@5_1:
top.pc1
top.pc12
top.pc13
References to top.c2@6_1:
top.pc2
xcelium> heap -gc      # Garbage collection
Performing heap garbage collection
xcelium>
```

help

The Tcl `help` command displays information about simulator (*xmsim*) commands and options, and predefined variable names and values.

You can:

- Get help on all commands.
- Get detailed help on a specific command.
- Get help on a command option.
- Display help on all commands that take a specific option.
- Display help for predefined simulation variable names and values.
- Display help for Tcl functions that can be used in expressions.

You also can use the `help` command to display help on standard Tcl commands. Only basic information is provided for these commands. Man pages for Tcl commands, as well as a summary of the Tcl language syntax, can be found on the Web at: <http://elf.org/etc/tcltk-man-html.html>.

help Command Syntax

```
help [help_options] [command | all [command_options]]  
    -brief  
    -functions [function_name ...]  
    -variables [variable_name ...]
```

help Command Options

This section describes the options that you can use with the Tcl `help` command.

| | |
|---|---|
| <code>all</code> | Shows full help for all commands. The special keyword <code>all</code> can be used instead of a specific command name. |
| <code>-brief</code> | Displays a list of commands with no description of the commands or options. |
| <code>-functions</code> <code>[function_name ...]</code> | Displays help for Tcl functions that can be used in expressions. This includes help on mathematical functions that are predefined in Tcl, as well as special functions that have been added to deal with Verilog values. If no function name is specified, help is displayed for all functions. |
| <code>-variables</code> <code>[variable_name ...]</code> | Displays a description of the specified predefined simulation variable name(s) and its current value. If no variable name is specified, help is displayed for all predefined variables. |

help Command Examples

The following command displays a list of all *xmsim* commands and Tcl standard commands:

```
xcelium> help
```

The following command displays help for the `probe` command and all its options:

```
xcelium> help probe
```

The following command displays a list of all options to the `probe` command. No description of the command or options is displayed:

```
xcelium> help -brief probe
```

The following command displays a list and description of options that can be used with the `probe` command used with the `-create` modifier:

```
xcelium> help probe -create
```

The following command displays the options that can be used with the `probe` command used with the `-create` modifier. No description of the options is displayed:

```
xcelium> help -brief probe -create
```

The following command displays full help for all *xmsim* commands and basic help for Tcl standard commands:

```
xcelium> help all
```

The following command displays a simple list of all *xmsim* and Tcl standard commands. No description of the commands or options is displayed:

```
xcelium> help -brief all
```

The following command displays brief help for all commands that have the `-enable` option:

```
xcelium> help -brief all -enable
```

The following command displays help for the `describe` command and for the `-nounit` option of the `time` command:

```
xcelium> help describe time -nounit
```

The following command displays help for the `describe` command, the `-nounit` option of the `time` command, and all commands with the `-enable` option:

```
xcelium> help describe time -nounit all -enable
```

The following command displays help for all predefined simulation variables:

```
xcelium> help -variables
```

The following command displays a description of the predefined simulation variable `time_scale`:

```
xcelium> help -variables time_scale
```

```
time_scale = NS.....Timescale of the current debug scope (read only)
```

The following command displays help for all Tcl functions that can be used in expressions:


```
xcelium> help -functions
```

history

The Tcl `history` command is a built-in Tcl command that allows you to re-execute commands without having to retype them. You also can use the `history` command to modify old commands; for example, to fix typographical errors.

The `history` command is similar to the UNIX `history` command, but with different syntax in some cases. You can:

- Modify the number of commands retained by the history mechanism (`keep`). By default, the history mechanism keeps track of the last twenty commands.
- Re-execute commands by giving their event number (`redo`).
- Modify parts of a previous command before re-executing it (`substitute`).

 You can also use the `!` command to re-execute commands.

history Command Syntax

```
history
    keep n
    redo event_number
    substitute old new event_number
```

history Command Options

This section describes the options that you can use with the Tcl `history` command.

`keep n`

Changes the number of commands retained by the history mechanism. The default is the 20 most recent commands.

`redo`

`[event_number]`

Re-execute the command specified by the `event_number` argument. The argument can be:

- **A positive number:** Re-executes the command with that number.

- **A negative number:** Re-executes a command relative to the current command. For example, `-1` re-executes the last command, `-2` re-executes the one before that, and so on.
- **A string:** Re-executes the command that matches the string. The string matches the command if it is the same as the first characters of the command or it meets Tcl's rules for string matching.

If no argument is specified, `redo` re-executes the most recent command.

| | |
|--|---|
| <code>substitute old new event_number</code> | Modifies the old command before executing it. |
|--|---|

history Command Examples

The `history` command is a built-in Tcl command with many features and options. The examples here illustrate only the most commonly used options.

Assume you have entered the following seven commands:

```
xcelium> database -open waves -into waves1.shm
xcelium> database -show
xcelium> stop -create -line 10
xcelium> stop -show
xcelium> run
xcelium> run 100 ns
xcelium> value data
```

The following sequence of commands illustrates some of the most useful features of the `history` command:

;`# Get a list of interactive commands.`

```
xcelium> history
1 database -open waves -into waves1.shm
2 database -show
3 stop -create -line 10
4 stop -show
5 run
6 run 100 ns
7 value data
8 history
```

;`# Reexecute the second from the last command (value data). Same as !-2.`

```
xcelium> history redo -2
4'b0000
```

;`# Reexecute command number 6 (run 100 ns). Same as !6.`

```
xcelium> history redo 6
```

```
Ran until 200 NS + 0
```

```
;# Reexecute the last command matching the string dat. Same as !dat.
```

```
xcelium> history redo dat
```

```
waves Enabled (file: waves1.shm) (SHM)
```

```
;# Reexecute command number 3, changing 10 to 11.
```

```
xcelium> history substitute 10 11 3
```

```
Created stop 2
```

Related Topic

- <https://www.tcl.tk/>

ida_probe

The Tcl `ida_probe` command is part of the Verisium Debug workflow. You can use this command to specify which objects you want to record as simulation data.

ida_probe Command Syntax

```
ida_probe options
```

ida_probe Command Options

This section describes the options that you can use with the Tcl `ida_probe` command.

| | |
|-----------------------------------|--|
| <code>-help</code> | Prints the help message. |
| <code>-log[= ...]</code> | Dumps log information to the Debug Analyzer database. This records all messages that apply to the current verbosity level. |
| <code>-log_objects</code> | Enables recording of links to dynamic objects in messages. |
| <code>-log_verbosity = ...</code> | Sets the verbosity level of messages, as NONE, LOW, MEDIUM, HIGH, or FULL (default). See Defining Message Verbosity in <i>Creating e Testbenches</i> for information on these settings. <code>-log_verbosity</code> applies to e only. Example: <code>ida_probe -log -log_verbosity=LOW</code> |

| | |
|--|--|
| <code>-e_files</code> | <p>List of e files to be recorded to Debug Analyzer database (all other files are not recorded). This option applies only to flow recording.</p> <p>Example: <code>ida_probe -flow -e_files="*monitor* ahb_frame_s.e"</code></p> <p>Use a space to separate file names. Use wildcards to include more than one file (wildcards apply to the full path of the file).</p> |
| <code>-ignore_e_files = "files"</code> | <p>List of e files to be excluded from Debug Analyzer record. This option applies only to flow recording. When specifying multiple files, enclose the list in quotes and separate the file names with spaces. You can also use wildcard characters anywhere in the full pathname.</p> <p>For example, <code>ida_probe -flow -ignore_e_files="*coverage* clock_handle.e"</code></p> <p>Use a space to separate file names. Use wildcards to include more than one file (wildcards apply to the full path of the file).</p> |
| <code>-e_list_max_size = ...</code> | <p>Determines the max size of an e list to be recorded in the IDA database (default is recording the first 1000 items of a list due to scalability reasons).</p> |
| <code>-e_events = ...</code> | <p>Filters in all information which is related to the given e events from Debug Analyzer database. This option applies only if the flag <code>-record_e_events</code> is used.</p> <p>For example, <code>ida_probe -record_e_events -e_events="my_protocol_monitor.*"</code></p> |
| <code>-ignore_e_events = ...</code> | <p>Filters out all information that is related to the given e events from Debug Analyzer database. This option applies only to flow recording.</p> |
| <code>-record_e_events[= ...]</code> | <p>This option enables the recording of e events. Note that this might have a big impact on recording performance.</p> <p>Please consider using this option along with <code>-ignore_e_events</code> to ignore the unneeded events or <code>-e_event</code> to select only the needed events</p> <p>For example, <code>ida_probe -flow -record_e_events -e_events="my_protocol_monitor.*"</code></p> |
| <code>-e_units = ...</code> | <p>Filters in all information that is related to the specified e unit subtree from Debug Analyzer database. This option applies only to flow recording.</p> <p>For example, <code>ida_probe -flow -e_units="sys.apb_subsystem.gpio sys.config"</code></p> <p>You must specify a full path to the unit starting from <code>sys</code>.</p> |
| <code>-ignore_e_units = ...</code> | <p>Filters out all information that is related to the given e unit subtree from Debug Analyzer database. This option applies only to flow recording.</p> <p>For example, <code>ida_probe -flow -ignore_e_units="sys.apb_subsystem.cov_mgr sys.topology"</code></p> |
| <code>-e_packages = ...</code> | <p>Lists of e packages to be recorded to Debug Analyzer database (all other packages are not recorded). This option applies only to flow recording.</p> <p>Example: <code>ida_probe -flow -e_packages="cdn_uart cdn_apb"</code></p> |

| | |
|--|--|
| <code>-ignore_e_packages = "pkgs"</code> | <p>Lists of e packages to be excluded from Debug Analyzer record. This option applies only to flow recording.</p> <p>For example, <code>ida_probe -flow -ignore_e_packages="cdn_gpio cdn_mem"</code></p> |
| <code>-ignore_e_methods = "methods"</code> | <p>Filters out all information that is related to the given e methods from Debug Analyzer database.</p> <p>This option applies only to flow recording. When specifying multiple methods, enclose the list in quotes and separate the method names with spaces.</p> <p>You can also use wildcard characters anywhere in the name. This example ignores the <code>get_clk</code> method in the <code>cdn_master_monitor_u</code> struct:</p> <pre>ida_probe -flow -ignore_e_methods="cdn_master_monitor_u.get_clk"</pre> <p>This example ignores any method that has <code>reset</code> in it's name:</p> <pre>ida_probe -flow -ignore_e_method="*.reset"</pre> |
| <code>-ignore_e_struct = "structs"</code> | <p>Filters out all information from specified e structs in the Debug Analyzer database. This option applies only to flow recording.</p> <p>When specifying multiple units, enclose the list in quotes and separate the unit names with spaces. You can also use wildcard characters anywhere in the name.</p> <p>For example, <code>ida_probe -flow -ignore_e_structs="cdn_master_transfer_s cdn_uart_monitor_u"</code></p> |
| <code>-encrypted[= ...]</code> | <p>Dumps encrypted flow information to Debug Analyzer database. This option applies only to flow recording.</p> <p>By default flow information of encrypted code is not saved. This option records all data items created inside an encrypted code.</p> <p>Use this option if you have encrypted e code in your environment.</p> <p>For example, <code>ida_probe -flow -encrypted</code></p> |
| <code>-f = <filename></code> | <p>Specifies a file containing the addition option for the <code>ida_probe</code> command.</p> <p>For example, <code>ida_probe -flow -f=input.txt</code></p> |
| <code>-save_files = directory</code> | <p>Specifies a <i>directory</i> location to save source files.</p> |
| <code>-flow = [on off]</code> | <p>Starts or stops recording flow information.</p> <p>Captures the order in which lines of code were executed during simulation.</p> <p>Use this option with other options to reduce the size of the database. The default is on.</p> |
| <code>-flow_on_error[= ...]</code> | <p>Dumps Specman flow information on the cycle where a <code>dut_error</code> is encountered. This allows you to see the information of the error scope and the error stack in IDA.</p> |



Only the errors are recorded. All other cycles are not recorded with flow information. This option is supported for **e** only.

| | |
|---|---|
| <code>-gen</code> | <p>Records specman generation information to the database.</p> <p>This enables tracking generated variables back to the generation reduction steps, and also enables an easy way to debug contradictions.</p> <p>This option applies only to flow recording, for example: <code>ida_probe -flow -gen</code></p> |
| <code>- record_ignored_files _field_assignment[= ...]</code> | <p>Records assignment to fields in ignored files. Default is FALSE.</p> <p>This option records any value change to fields even if the file where the assignment occurred is not recorded.</p> <p>This option creates an accurate view of the variables table values even for fields that are not recorded.</p> |
| <code>-start_time = time</code> | <p>Sets the beginning time in which probe activity is to be enabled. The time argument must include the time units with no space between the time and the time units, as in 100ns.</p> <p>Each <code>ida_probe</code> command can define only one time range. However, you can issue multiple <code>ida_probe</code> commands to record multiple time ranges.</p> <p>This option applies to the <code>-log</code>, <code>-flow</code>, <code>-sv_flow</code>, and <code>-wave</code> options specified to <code>ida_probe</code>.</p> <p>For example, <code>ida_probe -sv_flow -start_time=550ns</code></p> <p>The following line records SystemVerilog flow information for the specified time window:</p> <pre>ida_probe -sv_flow -ignore_sv_files="*cdn_parallel_per_cb.sv" - start_time=... -end_time=...</pre> |



You can specify multiple `ida_probe` commands to record multiple time windows.

`-end_time = time`

Stops recording at the specified time. The *time* argument must include the time units with no space between the time and the time units, as in 100ns.

Each `ida_probe` command can define only one time range. However, you can issue multiple `ida_probe` commands to record multiple time ranges.

This option applies to the `-log`, `-flow`, and `-sv_flow` options specified to `ida_probe`. It does not apply to `-wave`. For example, in the following line, Xcelium starts processing the log and flow options at time 550ns and turns them off at time 900ns. However, when probing for waveform information, Xcelium starts from time 550ns and records to the end of the simulation:

```
ida_probe -flow -start_time=550ns -end_time=900ns
-ignore_sv_files="*cdn_parallel_per_cb.sv" -wave -
wave_probe_args="top -depth all"
```

If you would like to turn on and off the `ida_probe` option selectively, you need to break the `ida_probe` command into segments. For example:

- The following line records smartlog and wave information for the entire run:
`ida_probe -log -wave -wave_probe_args="top -depth all"`
- The following line records flow information for the specified time window:
`ida_probe -sv_flow -ignore_sv_files="*cdn_parallel_per_cb.sv" -
start_time=... -end_time=...`



You can specify multiple `ida_probe` commands to record multiple time windows.

`-statement[= ...]`

Dumps wave information to Debug Analyzer database with statement assignments information (this improves the accuracy of the HDL Cause Analysis).

This option has a big effect on recording performance and is only valid when the design is compiled with `-linedebug` option.

This option is valid only when opening the wave DB (the first `ida_probe` command with `-wave` option, or the `ida_database` command).

`-status`

Shows current Debug Analyzer recording configuration.

`-sv_files`

Enables probing of SystemVerilog files.

For example, to create a flow probe where the source files includes System Verilog files:

```
ida_probe -sv_flow -sv_files="file1.sv file2.sv"
```

`-ignore_sv_files`
`= "files"`

Lists of System Verilog files to be excluded from Debug Analyzer record. This option applies only to flow information.

When specifying multiple files, enclose the list in quotes and separate the file names with spaces. You can also use wildcard characters anywhere in the name.

For example, `ida_probe -sv_flow -ignore_sv_functions="func1 func2"`

`-sv_flow[= ...]`

Records SystemVerilog flow information to the database.

When probing Systemverilog objects to the Debug Analyzer database, you must specify the following options:

- **-sv_flow**: Specifies that the test environment contains SystemVerilog code.
- **-wave_probe_args**: Specifies the top level of the design hierarchy and the depth of the hierarchy to be probed.
- **-sv_files**: If your test environment contains pure Verilog code, you must also specify the `-sv_files` option, because pure Verilog files are not recorded by default.

`-sv_functions = ...`

Lists of System Verilog functions/tasks to be recorded by Debug Analyzer record.

This option applies only to flow recording.

For example, `ida_probe -sv_flow -sv_functions="func1 func2"`

`-ignore_sv_functions`
`= "functs"`

Lists of System Verilog functions/tasks to be excluded from Debug Analyzer record. This option applies only to flow recording.

When specifying multiple functions, enclose the list in quotes and separate the function names with spaces. You can also use wildcard characters anywhere in the name.

The following example filters all functions named bar:

```
ida_probe -sv_flow -log -ignore_sv_functions="bar"
```

You can use `B: :bar` for filtering only the `bar` declared for `B`.

```
class A;
    task foo();
        int i = 55;
        #100;
        i++;
        $display ("in A.foo");
    endtask
    task bar();
        int j = 66;
        #100;
        j++;
        $display("in A.bar");
    endtask
endclass
class B;
    function foo();
        int i = 55;
        i++;
        $display ("in B.foo");
    endfunction
    function bar();
        int j = 66;
        j++;
        $display("in B.bar");
    endfunction
endclass
module top;
    initial begin
        A a = new();
        B b = new();
        #100;
        a.foo();
        #100;
        a.bar();
        #100;
        b.foo();
        #100;
        b.bar();
    end
endmodule
```

`-sv_instances = ...`

Lists of System Verilog instances (including all scopes in the hierarchy below the given instances) to be recorded by Debug Analyzer record. This option applies only to flow recording.

For example, `ida_probe -sv_flow -sv_instances="top.a top.b"`

| | |
|---|--|
| <code>-ignore_sv_instances</code> <code>= "insts"</code> | <p>Lists of System Verilog instances (including all scopes in the hierarchy below the given instances) to be excluded from Debug Analyzer record. This option applies only to flow recording.</p> <p>When specifying multiple instances, enclose the list in quotes and separate the instance names with spaces. You can also use wildcard characters anywhere in the name.</p> <p>You can force the App to ignore any activity under a specified design hierarchy, for example:</p> <pre>ida_probe -sv_flow -ignore_sv_instances="top.router"</pre> <p>(This is supported only if <code>top_router</code> is an HDL module instance.)</p> |
| <code>-include_build_phase[</code> <code>= ...]</code> | <p>Records SystemVerilog flow information for the build phase. This information is not recorded by default. This option applies only to flow recording.</p> <p>For example, <code>ida_probe -sv_flow -include_build_phase</code></p> |
| <code>-sv_all_logs[= ...]</code> | <p>Traces all SV messages to smartlog, including messages that are written to different destination files.</p> |
| <code>-sv_modules[= ...]</code> | <p>Records SystemVerilog module information to the database. This option applies only to flow information.</p> <p>SystemVerilog module information is not recorded by default.</p> <p>For example, <code>ida_probe -sv_flow -sv_modules</code></p> |
| <code>sv_packages = "pkgs"</code> | <p>Records information for the specific SystemVerilog packages.</p> <p>The default (<code>-sv_packages</code> with no <code>"pkgs"</code> argument) includes all packages except the uvm package, which is masked.</p> <p>This option applies only to flow information. When specifying multiple packages, enclose the list in quotes and separate the package names with spaces. You can also use wildcard characters anywhere in the name.</p> <p>For example, <code>ida_probe -sv_flow -sv_packages="cdn_gpio cdn_mem"</code></p> |
| <code>-ignore_sv_packages</code> <code>= "pkgs"</code> | <p>Lists of System Verilog packages to be excluded from Debug Analyzer record. This option applies only to flow information.</p> <p>When specifying multiple packages, enclose the list in quotes and separate the package names with spaces. You can also use wildcard characters anywhere in the name.</p> <p>For example, <code>ida_probe -sv_flow -ignore_sv_packages="cdn_gpio cdn_mem"</code></p> |
| <code>-ovm[= ...]</code> | <p>Records flow information for the OVM package (<code>ovm_pkg</code>). The default is FALSE.</p> <p>If you use this option, you must also use the <code>-linedebug</code> option when you compile the package.</p> |
| <code>-uvm[= ...]</code> | <p>Records flow information for the UVM package (<code>uvm_pkg</code>). The default is FALSE.</p> <p>If you use this option, you must also use the <code>-linedebug</code> option when you compile the package.</p> |
| <code>-uvm_reg[= ...]</code> | <p>Records flow information for the UVM registers. The default is FALSE.</p> <p>This flag has effect only if the UVM package was compiled with the <code>-linedebug</code> compilation switch.</p> |

| | |
|---|--|
| <code>-wave = [on off]</code> | <p>Starts or stops recording waveform information to the IDA database. The default is <code>on</code>.</p> <p>This option uses the default Xcelium simulator probe command options, which include the recording of statements, memories, functions, and tasks.</p> <p>For large designs, you should use the <code>-wave_probe_args</code> option to reduce the size of the database.</p> |
| <code>-wave_probe_args = "args"</code> | <p>Specifies arguments to pass to the Xcelium simulator <code>probe</code> command. When specifying multiple arguments, enclose the list in quotes and separate them with spaces.</p> <p>For example, <code>ida_probe -wave -wave_probe_args="dut_top -depth all"</code></p> |
| <code>- wave_glitch_recordin g[= ...]</code> | <p>Dumps wave information to the IDA database with <code>-event</code> configuration (this increases the accuracy of the HDL Cause Analysis).</p> <p>This option affects recording performance.</p> |

input

The Tcl `input` command pushes the specified script file(s) onto the standard input queue so that *xmsim* reads and executes the scripted commands at the next `xcelium>` Tcl prompt. The simulator executes the commands that are contained in the script file(s) as if you had typed them at the prompt.

You can also execute commands in a script file by using the `-input` option when you invoke the simulator. Like the `input` command, the `-input` option takes a script file as an argument and queues the commands in the file so that the simulator executes them when it issues its first prompt.

You can specify more than one script file with the `input` command. If you specify more than one file, the files are executed in the order that they appear on the command line.

A script file that you specify with the `input` command can also contain other `input` commands. When *xmsim* executes an embedded `input` command, it pushes the contents of the embedded input command file to the front of the input queue so that the commands are executed before the remaining commands from the original script file. That is, the simulator executes the commands in an embedded script file as if you included those commands in the original script file.

You can also execute Tcl commands in a script file by using the `source` command. However, because the commands that are pushed onto the input queue by the `input` command are not read until a prompt is issued, `input` commands that are embedded in a script file do not take effect until after the `source` command finishes and another prompt is issued. *xmsim* pushes the scripts from all of the `input` commands that it encounters when executing the `source` command so that they are executed in the order that they were encountered.

The `input` command also differs from the `source` command in the following ways:

- With the `source` command, execution of the commands in the script stops if a command generates an error. With the `input` command, the contents of the file are read in place of standard input at the next Tcl prompt, as if you had typed the commands at the command-line prompt. This means that errors

do not stop the execution of commands in the script.

- The `source` command displays the output of only the last command in the file. The `input` command, on the other hand, echoes commands to the screen as they are executed, along with any command output or error messages.

input Command Syntax

```
input [-quiet] script_file [script_file ...]
```

input Command Option

This section describes the options that you can use with the Tcl `input` command.

| | |
|---------------------|---|
| <code>-quiet</code> | Suppresses all output during the running of the specified script. |
|---------------------|---|

input Command Examples

In the following example, the snapshot is loaded into `xmsim` and then an `input` command is issued to execute the commands in the file `tcl.inp1`. This is the same as using the `-input` option on the `xmsim` command line when you invoke the simulator. The file `tcl.inp1` contains the following Tcl commands:

```
stop -create -line 27
run
value data
run 100 ns
value data
run 100 ns
value data
```

You can include the `-quiet` option on the `input` command line to suppress output when the script is running.

```
% xmsim -nocopyright -tcl worklib.harddrive
Loading snapshot worklib.harddrive:module ..... Done
xcelium> input tcl.inp1
xcelium> stop -create -line 27
Created stop 1
xcelium> run
0 FS + 0 (stop 1: ./harddrive.v:27)
./harddrive.v:27 repeat (2)
xcelium> value data
4'hx
```

```
xcelium> run 100 ns
Ran until 100 NS + 0
xcelium> value data
4'h0
xcelium> run 100 ns
Ran until 200 NS + 0
xcelium> value data
4'h1
xcelium>
```

In the following example, an `input` command is issued to execute the commands in the file `tcl.inp2`. The file `tcl.inp2` contains the following Tcl commands:

```
database -open -shm waves -default
probe -create -database waves clk
probe -create -database waves nonexistent_signal
probe -create -database waves data

% xmsim -nocopyright -tcl worklib.hardrive
Loading snapshot worklib.hardrive:module ..... Done
xcelium> run 50 ns
Ran until 50 NS + 0
xcelium> input tcl.inp2
xcelium> database -open -shm waves -default
Created default SHM database waves
xcelium> probe -create -database waves clk
Created probe 1
xcelium> probe -create -database waves nonexistent_signal
xmsim: *E,PNOOBJ: Path element could not be found: non_existent_signal.
```

Notice that, unlike the `source` command, errors do not stop the execution of commands in the script. The simulator generates an error message for the second `probe` command, and then the following command is executed:

```
xcelium> probe -create -database waves data
Created probe 2
xcelium>
```

logfile

The Tcl `logfile` command saves the output of some Tcl commands to a specified log file rather than to the default log file (`xmsim.log` or `xrun.log`, or the logfile specified with the `xmsim -logfile` or `xrun -log_xmsim` options). This command has two modifiers:

- **-set:** Lets you switch from the default log file to a specified log file, and vice versa. This modifier is the default and is optional.

- **-show**: Displays the name of the current log file. If the `-default` option is included, displays the name of the default log file.

logfile Command Syntax

logfile

```
[-set] {logfile_name | -default}  
    -append logfile_name  
    -overwrite logfile_name  
-show [-default] [perfstat]
```

logfile Command Options

This section describes the options that you can use with the Tcl `logfile` command.

| | |
|---|--|
| <code>-set {logfile_name -default}</code> | <p>Changes the log file being used for simulator output.</p> <p>The <code>-set</code> modifier is optional.</p> <p>You must specify either:</p> <ul style="list-style-type: none">• The name of the log file you want to switch to. For example: xcelium> logfile newlog.log• The <code>-default</code> option. This changes the log file back to the default log file. For example: xcelium> logfile -default <p>When you switch back to the default log file with a logfile command, the output is appended to the default log file (that is, <code>-append</code> is implied). You cannot include the <code>-overwrite</code> option to overwrite the default log file.</p> <p>If the simulator was invoked with the <code>-nolog</code> option, no default simulation log file is created. In this case, a Tcl logfile command turns to log on. Reverting to the default means going back to not having a log file.</p> |
| <code>-append logfile_name</code> | <p>Appends to the specified log file.</p> <p>When the specified file already exists, you must use either the <code>-append</code> or <code>-overwrite</code> option to specify if you want to append the output to the existing file or to overwrite the file. An error message is generated if neither option is used.</p> |
| <code>-overwrite logfile_name</code> | <p>Overwrites the contents of the specified log file.</p> <p>When the specified file already exists, you must use either the <code>-append</code> or <code>-overwrite</code> option to specify if you want to append the output to the existing file or to overwrite the file. An error message is generated if neither option is used.</p> |
| <code>-show [-default] [perfstat]</code> | <p>Displays the name of the default log file.</p> <p>If you include the <code>-default</code> option, the name of the default log file is displayed. If you include the <code>-perfstat</code> option, the name of the perfstat file is displayed.</p> |

logfile Command Examples

In the following examples, which use *xrun*, the default log file is *xrun.log*:

```
;# Output of the following commands goes to xrun.log.
xcelium> logfile -show
The current log file is: xrun.log

;# Switch to a log file called xrun_cov.log.
xcelium> logfile xrun_cov.log
xcelium>

;# Output of the following commands goes to xrun_cov.log.
xcelium> coverage -functional -list top.vcomp0.arb_fairness top.vcomp0.grnt_asserted

;# Switch to default logfile, xrun.log.
xcelium> logfile -default
xcelium>

;# Output of the following commands is appended to xrun.log.
xcelium> run
```

The following example illustrates the effect of logfile command options. The default log file in this example is *xmsim.log*:

```
;# The output of the following commands goes to xmsim.log.
run 1
value @1
run 100
describe @1

;# Switch to a log file called new.log.
logfile new.log

;# Output of the following commands goes to new.log.
run 500
scope -describe @1
value top.test1.Base@1_1.it

;# Display the name of the current log file (new.log).
logfile -show

;# Switch to the default log file (xmsim.log).
logfile -default

;# Output of the following commands goes to xmsim.log.
run 100
value top.test1.Base@1_1.it

;# Switch to log file new.log.
logfile -append new.log
```

```
;# Output of the following commands is appended to new.log.
run 100
value top.test1.Base@1_1.it

;# Switch to log file new2.log.
logfile new2.lo

;# Output of the following commands goes to new2.log.
run 100
value top.test1.Base@1_1.it

;# Display the name of the default log file (xmsim.log).
logfile -show -default

;# Switch to the default log file.
logfile -default

;# Output of the following commands is appended to xmsim.log.
run 100
value top.test1.Base@1_1.it

;# The following command generates an error because new2.log already exists.
;# Must use either -append or -overwrite to open the file.
logfile new2.log

;# Switch to new2.log and overwrite it.
logfile -overwrite new2.log

;# Output of the following commands goes to new2.log.
reset
run 100
exit
```

loopvar

The Tcl `loopvar` command enables you to:

- Display the value of VHDL for-loop variables (`-value`).
- Deposit a value into a VHDL for-loop variable (`-deposit`).
- Describe VHDL for-loop variables (`-describe`).

You must compile the source code with the `xmvhdl -linedebug` command-line option to use the `loopvar` command. The simulator generates an error if you try to access a loop variable in a design unit that has not been compiled with the `-linedebug` option. This option is the default with the VHDL Desktop simulator.

The current release does not include GUI support for accessing loop variables. If you are using the SimVision analysis environment, you must enter the `loopvar` command in the I/O region of the Console window.

Loop variables are "alive" only when the associated loop is executing. Therefore, you can query the value of a loop variable or deposit it into a loop variable only while the loop is executing. You can, however, describe the loop variables even when the loop is not executing.

The text in the following sections refers to variables whose loops are currently executing as being *active*.

Note that, if a for-loop has a wait statement, and the loop is currently executing the `wait` statement, the loop and the loop variable are both active.

Loop labels can be used in the loop variable name. For example:

```
xcelium> loopvar L1.i
```

Each `loopvar` command option has a `-scope` option that you can use to specify the scope. In a mixed-language design, you can access a loop variable from a Verilog scope by specifying the VHDL scope with this option.

Setting object breakpoints on loop variables are not supported. However, you can set a conditional line breakpoint (`-line line_number -if tcl_expression`).

You cannot use the standard VHDL path naming conventions to access a loop variable. For example, you cannot access a loop variable with the following full path:

```
:P1:L1:i
```

loopvar Command Syntax

```
loopvar [-deposit loopvar_id [=] value] [-scope scope_name]  
        [-describe [loopvar_id ...] [-scope scope_name]  
        [-value] [loopvar_id ...] [-scope scope_name]
```

loopvar Command Options

| | |
|--|--|
| <code>-scope <scope_name></code> | <p>Specifies a scope. The <code>scope_name</code> argument must be a process or a process stack frame. Use the following format when naming a stack frame:</p> <pre>process_path_name[call_frame_level]</pre> <p>For instance:</p> <pre>-scope :P1[1]</pre> <p>If you do not specify a scope, the simulator assumes the current debug scope. And, an error results if the current debug scope is not a process or a subprogram scope.</p> |
| <code>-value</code> <code>[loopvar_id ...] [-</code> <code>scope scope_name]</code> | <p>Displays the value of the specified loop variable(s).</p> <p>The <code>-value</code> option is the default for the <code>loopvar</code> command. For example, the following two commands are identical.</p> <pre>xcelium> loopvar -value i xcelium> loopvar i</pre> <p>If you do not specify a <code>loopvar_id</code> argument, the value of all active loop variables in the current scope is displayed.</p> |
| <code>-describe</code> <code>[loopvar_id ...] [-</code> <code>scope scope_name]</code> | <p>Describes the specified loop variable(s).</p> <p>Example:</p> <pre>xcelium> loopvar -describe i \$LOOP_000.i.....loop variable : INTEGER = 3 (Line 16)</pre> <p>The <code>-describe</code> option can be used for both active and inactive loop variables. The value is printed only if the loop is active.</p> <p>If you do not specify a <code>loopvar_id</code> argument, all loop variables in the current scope are described.</p> |
| <code>-</code> <code>deposit loopvar_id [</code> <code>=] value [-</code> <code>scopescope_name]</code> | <p>Deposits the specified value to the loop variable.</p> <p>Example:</p> <pre>xcelium> loopvar -deposit i 3</pre> <p>You can only deposit a value to an active loop variable.</p> <p>Only valid values as defined by the range specified in the for-loop statement are allowed with the <code>-deposit</code> option. For example, the following command for <code>-loop</code> statement defines a variable <code>i</code> with a range of 1 to 10. You cannot deposit a value greater than 10 or less than 1 to this variable.</p> <pre>L1: for i in 1 to 10</pre> |

loopvar Command Examples

Example 1

The following VHDL source file is used for this example. The source file has been compiled with the `-linedebug` command-line option.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_arith.all;
use IEEE.Std_Logic_unsigned.all;
entity entity_object is
end entity_object;
architecture arch_object of entity_object is
    signal counter_sig : integer:=0;
begin
    P : process
        variable counter_var : integer:=0;
        begin
            for i in 0 to 9
                loop
                    counter_sig <= counter_sig + 1;    -- This is line 18
                    counter_var := counter_var + 1;
                end loop;
            wait;
        end process;
    end arch_object;
```

The following `scope` command sets the scope to `:P`. Because the loop in this process is not currently executing, you cannot use the `loopvar -value` command to see the value of the loop variable `i`, or `loopvar -deposit` to deposit a value to the loop variable. You can, however, describe the loop variable with `loopvar -describe`.

```
xcelium> scope -set :P
xcelium> loopvar -value
xmsim: *W,LVNOA: No active loop variables found in scope ':P'.
xcelium> loopvar -deposit i 4
xmsim: *W,LVINA: Loop variable i is currently not active.
xcelium> loopvar -describe
$LOOP_000.i.....loop variable : INTEGER ( inactive )    ( Line 16 )
```

The following commands set a line breakpoint on line 18 and then run the simulation. The scope is now an active process with an active for loop:

```
xcelium> stop -create -line 18
Created stop 1
xcelium> run
0 FS + 0 (stop 1: ./test_design.vhd:18)
./test_design.vhd:18                counter_sig <= counter_sig + 1;
```

The following two commands are identical. They display the value of all active loop variables in the current debug scope:

```
xcelium> loopvar -value
$LOOP_000.i = 0    ( Line 16 )
```

```
xcelium> loopvar
$LOOP_000.i = 0    ( Line 16 )
```

The following command displays the value of loop variable `i` in the current debug scope:

```
xcelium> loopvar i
0
```

The following command displays the value of all loop variables in scope `:P`:

```
xcelium> loopvar -scope :P
$LOOP_000.i = 0    ( Line 16 )
```

The following two commands are identical. They display the value of loop variable `i` in scope `:P`:

```
xcelium> loopvar -value i -scope :P
0
xcelium> loopvar i -scope :P
0
```

In this example the loop is not labelled. The simulator generates a name for the loop variable (`$LOOP_000.i`). You can use this name with the `loopvar` command as follows:

```
xcelium> loopvar \${loop_000.i}
0
```

The following command describes all loop variables in scope `:P`:

```
xcelium> loopvar -describe -scope :P
$LOOP_000.i.....loop variable : INTEGER = 0          ( Line 16 )
```

The following command sets loop variable `i` in scope `:P` to 3.

```
xcelium> loopvar -deposit i 3 -scope :P
xcelium> loopvar i -scope :P
3
```

The following command describes loop variable `i` in the current debug scope:

```
xcelium> loopvar -describe i
$LOOP_000.i.....loop variable : INTEGER = 3          ( Line 16 )
```

Example 2

The VHDL source file used for the following examples contains a nested for loop. The loops are labelled `L1` and `L2`.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_arith.all;
use IEEE.Std_Logic_unsigned.all;
entity entity_object is
end entity_object;
architecture arch_object of entity_object is
    signal counter_sig : integer:=0;
begin
    P : process
        variable counter_var : integer:=0;
        begin
            L1:for i in 0 to 9
                loop
                    L2:for i in 0 to 9
                        loop
                            counter_sig <= counter_sig + 1;    -- This is line 20
                            counter_var := counter_var + 1;
                        end loop L2;
                    end loop L1;
                wait;
            end process;
        end arch_object;
```

The following commands set a line breakpoint on line 20 and then run the simulation. The current debug scope is now :P, and the current execution point is in loop L2.

```
xcelium> stop -create -line 20
Created stop 1
xcelium> run
0 FS + 0 (stop 1: ./test_design.vhd:20)
./test_design.vhd:20          counter_sig <= counter_sig + 1;
```

The following command displays the values of all active loop variables in the current debug scope:

```
xcelium> loopvar
L1.i = 0
L2.i = 0
```

If there are multiple active loop variables with the same name, the `loopvar` command displays the value of the variable in the innermost loop. The following command displays the value of `i` of loop L2:

```
xcelium> loopvar i
0
```

A loop variable name can include the loop label. The following command displays the value of variable `i` of loop L1:

```
xcelium> loopvar L1.i
```


0

The following command displays the value of variable `i` of loop `L1` and `i` of loop `L2`:

```
xcelium> loopvar L1.i L2.i
L1.i = 0
L2.i = 0
```

The following command deposits the value `3` into variable `i` of loop `L2`:

```
xcelium> loopvar -deposit i 3
```

The following command deposits the value `4` into variable `i` of loop `L1`:

```
xcelium> loopvar -deposit L1.i 4
```

The following command describes all active loop variables in the current debug scope:

```
xcelium> loopvar -describe
L1.i.....loop variable : INTEGER = 4
L2.i.....loop variable : INTEGER = 3
```

Setting an object breakpoint on a loop variable is not supported. However, you can set a conditional line breakpoint as shown in the following command:

```
xcelium> stop -line 20 -if {[loopvar -value i] = 1}
Created stop 2
```

maptypes

The Tcl `maptype` command invokes the mapping process to generate data types in a target language based on data types in a source language. In general, this command should be called after the necessary configuration commands have been executed. These data types can then be used as data types in ML UVM transactions using ML UVM TLM ports.

maptypes Command Syntax

```
maptypes
    -to_lang lang
    -from_type type_name
    -base_name base_name
```

maptypes Command Options

This section describes the options that you can use with the Tcl `maptypes` command.

-to_lang *lang*

The name of the target language. The accepted language names are:

- **e** (or **sn**)
- **sv**
- **sc**

The language names are case insensitive. This parameter is mandatory and can be repeated.

When types are mapped to two or more target languages, Cadence recommends doing so in a single invocation.

-from_type *type-*
name

The name of the source type. The *type-name* must be the full name (including package/namespace qualifiers) and must contain the language prefix (**e**, **SV**, **SC**).

This parameter is mandatory and may repeat. Examples:

```
-from_type e:my_pkg::my_struct  
-from_type sv:package1::my_data_item
```

-base_name *base-*
name

The basename for the output files. This parameter is mandatory.

The basename can contain a directory path to a folder. For example:

```
-base_name my_types  
-base_name my_dir/my_dt
```

maptypes Command Examples

The following command generates type definitions in SystemVerilog for the **e** type `ahb_packet`:

```
mltypemap> maptypes -from_type e:ahb::ahb_packet \  
-to_lang sv \  
-base_name ahb_datatypes
```

This command generates three files: `ahb_datatypes.svh`, `ahb_datatypes_ser.sv`, and `ahb_datatypes_ser.sv`.

Related Topic

- [Mapping Types with mltypemap](#)

memory

The Tcl `memory` command loads a Verilog or VHDL memory or dumps the contents of a specified memory object to a specified memory file. This command has two modifiers: `-load` and `-dump`.

The following are common tasks you can perform with `memory` commands:

- [Loading Memory](#)

- [Dumping Memory](#)
- [Specifying Memory File Format](#)

For more information on declaring Verilog and VHDL memories in your HDL source, see [Memory Objects](#).

memory Command Syntax

```
memory -load memory_object -file mem_filename -start start_address \  
    [-end end_address] [-addressfmt address_format] [-datafmt data_format] \  
    [-defval default_value]  
  
-dump [output_format] [-start start_address] [-end end_address] \  
memory_object -file mem_filename
```

memory Command Options

This section describes the options that you can use with the Tcl `memory` command.

`-load`

Loads a memory object. You can use the following options:

- **-file *mem_filename***: Specifies the name of the memory image file.

If you are using the memory `-load` command, this option specifies the name of the memory image file that contains the data to be loaded. If you are using the memory `-dump` command, this option specifies the name of the output file.

- **-start *start_address***: Specifies the address where the load should start.

If the memory image file is in data-only format, you must include the `-start start_address` option to specify the address where the load should start. The `-end` option is not required. If the memory image file is in address/data format, and you include the `-start` and `-end` options, the addresses specified with the options are used.

- **-end *end_address***: Specifies the address where the load should end.

If the memory image file is in data-only format, you must include the `-start start_address` option to specify the address where the load should start. The `-end` option is not required.

If the memory image file is in address/data format, `-start` and `-end` are both optional. If you include the options, the addresses specified with the options are used.

- **-addressfmt *address_format***: Specifies the format of the addresses to be loaded. The *address_format* can be B or b (Binary), O or o (Octal), D or d (Decimal), or H or h (Hex).

If you do not specify the address format on the command line, the format specified in the memory file with the \$ADDRESSFMT directive is used. If the \$ADDRESSFMT directive is not specified, the default address format is hexadecimal.

- **-datafmt *data_format***: Specifies the format of the data in the memory image file. The *data_format* can be B or b (Binary), O or o (Octal), D or d (Decimal), or H or h (Hex).

If you do not specify the data format on the command line, the format specified in the memory file with the \$DATAFMT directive is used. If the \$DATAFMT directive is not specified, the default data format is hexadecimal.

- **-defval *default_value***: Specifies a default value for unspecified addresses. The default value can be:
 - 0, 1, X, Z for Verilog
 - 0, 1, X, Z, W, H, L, U, - for VHDL

If you do not specify a default value on the command line, the value specified in the memory file with the \$DEFAULTVALUE directive is used. If a default value is not specified in the memory file, only the memory addresses that are specified are filled. Other addresses remain unchanged.

-dump

Dumps the contents of memory into a memory file. You can specify an output format. The options are: %h (hexadecimal), %d (decimal), %o (octal), or %b (binary). %h is the default.

You can use the following options:

- **-file *mem_filename***: Specifies the name of the memory image file.

If you are using the memory -load command, this option specifies the name of the memory image file that contains the data to be loaded. If you are using the memory -dump command, this option specifies the name of the output file.

- **-start *start_address***: Specifies the address where the load should start.

If the memory image file is in data-only format, you must include the `-start start_address` option to specify the address where the dump should start. The `-end` option is not required.

If the memory image file is in address/data format, and you include the `-start` and `-end` options, the addresses specified with the options are used.

The arguments to the `-start` and `-end` options, which specify the address where the dump should start and the address where the dump should end, are integers, which you can specify in decimal, hexadecimal, or octal notation.

- **-end *end_address***: Specifies the address where the load should end.

If the memory image file is in data-only format, you must include the `-start start_address` option to specify the address where the dump should start. The `-end` option is not required.

If the memory image file is in address/data format, `-start` and `-end` are both optional. If you include the options, the addresses specified with the options are used.

The arguments to the `-start` and `-end` options, which specify the address where the dump should start and the address where the dump should end, are integers, which you can specify in decimal, hexadecimal, or octal notation.

memory Command Examples

Example 1: Loading memory with a value specified on the command line

In the following VHDL example, the signal `MEM_OBJ` is defined as follows:

```
type MEMTYPE is array (NATURAL range <>) of std_logic_vector (3 downto 0);  
signal MEM_OBJ: MEMTYPE (0 to 15);
```

The following command loads all elements of `MEM_OBJ` with a value of 1:

```
xcelium> memory -load :MEM_OBJ -defval 1  
xcelium> value :MEM_OBJ  
("1111", "1111", "1111", "1111", "1111", "1111", "1111", "1111", "1111", "1111", "1111", "1111",  
"1111", "1111", "1111", "1111")
```

Example 2: Loading memory with a file in data-only format

In the following VHDL example, the signal `MEM_OBJ` is defined as follows:

```
type MEMTYPE is array (NATURAL range <>) of std_logic_vector (3 downto 0);  
signal MEM_OBJ: MEMTYPE (0 to 15);
```

The memory image file, called `mem_load.dat`, is as follows:

```
# Addresses specified with -start/-end are in decimal format  
$ADDRESSFMT D  
# Data is in hexadecimal format  
$DATAFMT H  
9  
8  
7  
8  
Z;Z;Z  
W  
L  
7  
6  
5  
4;4  
1;1
```

To load this memory image file, you must include the `-start` option and, optionally, the `-end` option.

The following command specifies that the load is to start with address 0 and end with address 15.

The `$ADDRESSFMT` variable in the data file specifies that the arguments to `-start` and `-end` are in decimal format:

```
xcelium> memory -load :MEM_OBJ -file mem_load.dat -start 0 -end 15  
xcelium> value :MEM_OBJ  
("1001", "1000", "0111", "1000", "ZZZZ", "ZZZZ", "ZZZZ", "WWWW", "LLLL", "0111", "0110",  
"0101", "0100", "0100", "0001", "0001")
```

The following command specifies that the load is to start with address 15 and end with address 0:

```
xcelium> memory -load :MEM_OBJ -file mem_load.dat -start 15 -end 0  
xcelium> value :MEM_OBJ  
("0001", "0001", "0100", "0100", "0101", "0110", "0111", "LLLL", "WWWW", "ZZZZ", "ZZZZ",  
"ZZZZ", "1000", "0111", "1000", "1001")
```

The following command specifies that the load is to start with address 2 and end with address 13.

Because a default value is not specified with the `-defval` option or with the `$DEFAULTVALUE` directive, the unspecified addresses (0, 1, 14, and 15) remain unchanged:

```
xcelium> memory -load :MEM_OBJ -file mem_load.dat -start 2 -end 13  
xcelium> value :MEM_OBJ
```

```
("UUUU", "UUUU", "1001", "1000", "0111", "1000", "ZZZZ", "ZZZZ", "ZZZZ", "WWWW", "LLLL",  
"0111", "0110", "0101", "UUUU", "UUUU")
```

The following command includes the `-start` option, but the `-end` option is omitted. The load starts with address 2 and ends with the highest address (15):

```
xcelium> memory -load :MEM_OBJ -file mem_load.dat -start 2  
xcelium> value :MEM_OBJ  
("UUUU", "UUUU", "1001", "1000", "0111", "1000", "ZZZZ", "ZZZZ", "ZZZZ", "WWWW", "LLLL",  
"0111", "0110", "0101", "0100", "0100")
```

A `$DEFAULTVALUE` directive is not included in the memory file to specify a default value. The following command includes the `-defval` option to specify a default value of 0 for unspecified addresses.

```
xcelium> memory -load :MEM_OBJ -file mem_load.dat -start 4 -defval 0  
xcelium> value :MEM_OBJ  
("0000", "0000", "0000", "0000", "1001", "1000", "0111", "1000", "ZZZZ", "ZZZZ", "ZZZZ",  
"WWWW", "LLLL", "0111", "0110", "0101")
```

The `$DATAFMT` directive in the memory file specifies that the format for the data is hexadecimal. The following command includes the `-datafmt` option to specify that the format is decimal. This overrides the specification in the memory file.

```
xcelium> memory -load :MEM_OBJ -file mem_load.dat -start 0 -end 10 \  
-datafmt d -defval 0  
xcelium> value :MEM_OBJ  
("1001", "1000", "0111", "1000", "ZZZZ", "ZZZZ", "ZZZZ", "WWWW", "LLLL", "0111", "0110",  
"0000", "0000", "0000", "0000", "0000")
```

Example 3: Loading memory with a file in address/data format

In the following VHDL example, the signal `MEM_OBJ` is defined as follows:

```
type MEMTYPE is array (NATURAL range <>) of std_logic_vector (3 downto 0);  
signal MEM_OBJ: MEMTYPE (0 to 15);
```

The memory image file, called `mem_load.dat`, is as follows. In this example, the `$ADDRESSFMT` and `$DATAFMT` variables are not used to specify the address and data format.

```
# Address format is not specified with $ADDRESSFMT. Default is hexadecimal.  
# Data format is not specified with $DATAFMT. Default is hexadecimal.  
# Single address, single data  
0/9  
1/8  
2/7  
3/8  
#Multiple address, single data  
4:6/Z  
#Single address, multiple data
```

```
7/W;L
9/7
a/6
b/5
c:d/4
e:f/1
```

If you execute a memory `-load` command without the `-start` and `-end` options, the addresses specified in the file are used.

The following command loads the memory starting with address 0 and ending with address 15:

```
xcelium> memory -load :MEM_OBJ -file mem_load.dat
xcelium> value :MEM_OBJ
("1001", "1000", "0111", "1000", "ZZZZ", "ZZZZ", "ZZZZ", "WWWW", "LLLL", "0111", "0110",
"0101", "0100", "0100", "0001", "0001")
```

For lines with a single address and multiple data, such as the line 7/W;L in the memory file shown above, the direction of the load is determined by the direction of the signal definition, which is ascending (0 to 15) in this example. W is loaded into address 7, and L is loaded into address 8.

If you execute a memory `-load` command and include the `-start` and `-end` options, the addresses specified in the file are ignored, and the start and end addresses specified with the options are used. The direction of the load (ascending or descending) is determined from the `-start` and `-end` options.

For example, if the memory `-load` command includes `-start` and `-end` options, the addresses in the memory file, `mem_load.dat`, shown above are ignored. The file is read as follows:

```
# Address format is not specified with $ADDRESSFMT. Default is hexadecimal.
# Data format is not specified with $DATAFMT. Default is hexadecimal.
9
8
7
8
Z
W;L
7
6
5
4
1
```

The following command specifies that the load is to start with address 0 and end with address 15. Notice that the last data element is loaded into address 11.

```
xcelium> memory -load :MEM_OBJ -file mem_load.dat -start 0 -end 15
xcelium> value :MEM_OBJ
("1001", "1000", "0111", "1000", "ZZZZ", "WWWW", "LLLL", "0111", "0110", "0101", "0100",
```



```
% xrun -access +rwc -tcl asso.sv
file: asso.sv
    module worklib.assoc:sv
        errors: 0, warnings: 0
    Elaborating the design hierarchy:
    ...
    ...
    Writing initial simulation snapshot: worklib.assoc:sv
Loading snapshot worklib.assoc:sv ..... Done
xcelium> value mem[0]
256'hxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xcelium> memory -load assoc.mem ./preload.data
xcelium> value mem[0]
256'h0000000000000000000000000000000000000000000000000000000000004012
xcelium> run 50
Ran until 50 NS + 0
xcelium> memory -dump -file memdump.log %h assoc.mem
xcelium> exit
% cat memdump.log
# Memory Dump.
# Version - TOOL:          xcelium    11.10-p020
# Memory Object - assoc.mem
$ADDRESSFMT H
$DATAFMT h
0000000000000000/000000000000000000000000000000000000000000000000000000000004012
0000000000000002/00000000000000000000000008000000c136201c0006801823228025d6de802568128
0000000000000004/0000000000ff03000008000000c136201c0006801823228025d6de8abd12354
0000000000000008/0000000000ff03000008000000c136201c0006801823228025d6de8abd12354
...
...
# End of Memory Dump
```

Memory Objects

Before you can load or dump a memory object using the Tcl [memory](#) command, you must first declare that object. To declare Verilog and VHDL memories with Xcelium, you define such objects in the source HDL using the methods described below.

- **Verilog Memory Objects**

Verilog memories are declared as:

```
reg memlbit [15:0]          // Memory memlbit with 16 1-bit words
reg [7:0] membyte [15:0]    // Memory membyte with 16 8-bit words
```

The memory must be a one-dimensional array.

SystemVerilog associative arrays are also supported, with the following restrictions:

- The associative array must be a simple associative array that is declared inside the module.

For example:

```
module top;

    // Use associative array as storage
    reg [255:0] fuse_array[[63:0]];

    ...

    ...

endmodule
```

- The array index must be of integral type. Other types (for example, string or class handle) are not supported.
- You cannot load an associative array with a value specified on the command line. Data must be loaded from a file in [Address/Data format](#). Only single address/single data format is supported.
- You cannot specify a start address with the `-start` option or specify an end address with the `-end` option when you load or dump the memory.
- Specifying a default value for associative array elements is not allowed.

See [Example 5: Loading and Dumping a SystemVerilog Associative Array](#) for an example.

• VHDL Memory Objects

For VHDL, the array object must be a one-dimensional array, and its index subtype must be an integer type. The element type of the variable must also be a one-dimensional array. The type of the array elements must be an enumeration type that contains the literals 0 and 1, and that can contain any of the enumeration literals in `std_logic`. Here is an example of a VHDL array that satisfies these conditions:

```
type MVL4 is (X, 0, 1, Z);
type MVL4_VECTOR is array (NATURAL range <>) of MVL4;
type MEMTYPE is array (NATURAL range <>) of MVL4_VECTOR (31 downto 0);
variable MEM: MEMTYPE (0 to 1023);
```

The memory object must have write access. Use the `-access +w` option when you elaborate the design (`xmelab -access +w` or `xrun -access +w`).

Memories up to 4096 bits wide can be loaded with `memory -load`.

Related Topic

- [Specifying Memory File Format](#)

Loading Memory

Use the `-load` option with the `memory` command to load memory. You can:

- Load a memory object with a value specified on the memory command line. See [Loading Memory With a Value Specified on the Command Line](#).
- Load a memory object with data specified in a memory file. See [Loading Memory From a Memory Image File](#).

If the memory image file is in data-only format, you must include the `-start` option to specify the address where the load should start. The `-end` option, which specifies the address where the load should end, is optional. If the memory image file is in address/data format, `-start` and `-end` are optional. If you include them, the addresses specified in the file are ignored.

If the memory image file has fewer elements than the memory, the corresponding elements (addresses) of the memory are changed. The rest of the memory elements are either set to the default value that you specify with the `-defval` option or the `$DEFAULTVALUE` directive, or they remain unchanged if no default value is specified.

If the memory image file has an address that does not exist in the memory, the simulator issues a warning message and ignores the elements in the memory image file.

The `memory -load` command fills each element of the memory from LSB to MSB. If the width of the memory element is smaller than the width of the memory image file element, the most significant (left-most) bits of the memory image file element are lost. If the memory image file has fewer elements than the memory, the corresponding bits of the element are changed. The rest of the bits are either set to the default value specified with the `-defval` option or the `$DEFAULTVALUE` directive, or remain unchanged if no default value is specified.

Syntax

```
memory -load memory_object -file mem_filename -start start_address \  
    [-end end_address] [-addressfmt address_format] [-datafmt data_format] \  
    [-defval default_value]
```

Options

`-load`

Loads a memory object. You can use the following options:

- **`-file mem_filename`:** Specifies the name of the memory image file.

If you are using the `memory -load` command, this option specifies the name of the memory image file that contains the data to be loaded. If you are using the `memory -dump` command, this option specifies the name of the output file.

- **-start *start_address***: Specifies the address where the load should start.

If the memory image file is in data-only format, you must include the **-start *start_address*** option to specify the address where the load should start. The **-end** option is not required. If the memory image file is in address/data format, and you include the **-start** and **-end** options, the addresses specified with the options are used.

- **-end *end_address***: Specifies the address where the load should end.

If the memory image file is in data-only format, you must include the **-start *start_address*** option to specify the address where the load should start. The **-end** option is not required.

If the memory image file is in address/data format, **-start** and **-end** are both optional. If you include the options, the addresses specified with the options are used.

- **-addressfmt *address_format***: Specifies the format of the addresses to be loaded. The *address_format* can be B or b (Binary), O or o (Octal), D or d (Decimal), or H or h (Hex).

If you do not specify the address format on the command line, the format specified in the memory file with the `$ADDRESSFMT` directive is used. If the `$ADDRESSFMT` directive is not specified, the default address format is hexadecimal.

- **-datafmt *data_format***: Specifies the format of the data in the memory image file. The *data_format* can be B or b (Binary), O or o (Octal), D or d (Decimal), or H or h (Hex).

If you do not specify the data format on the command line, the format specified in the memory file with the `$DATAFMT` directive is used. If the `$DATAFMT` directive is not specified, the default data format is hexadecimal.

- **-defval *default_value***: Specifies a default value for unspecified addresses. The default value can be:
 - 0, 1, X, Z for Verilog
 - 0, 1, X, Z, W, H, L, U, - for VHDL

If you do not specify a default value on the command line, the value specified in the memory file with the `$DEFAULTVALUE` directive is used. If a default value is not specified in the memory file, only the memory addresses that are specified are filled. Other addresses remain unchanged.

Loading Memory With a Value Specified on the Command Line

You can load all addresses in a memory object with a value specified on the command line. The command is as follows:

```
memory -load memory_object -defval value
```

The value can be:

- 0, 1, X, Z for Verilog
- 0, 1, X, Z, W, H, L, U, - for VHDL

For example:

```
memory -load data -defval 0
```

See the [Loading memory with a value specified on the command line](#) example.

Loading Memory From a Memory Image File

You can load a memory using information contained in a memory image file.

A memory image file contains:

- Optional directives for specifying the address format and the data format.
 - `$ADDRESSFMT address_format`
The *address_format* can be B or b (Binary), O or o (Octal), D or d (Decimal), or H or h (Hex). If the `$ADDRESSFMT` directive is not specified in the data file, it can be specified on the command line with the `-addressfmt` command-line option. If neither is specified, the default address format is hexadecimal.
 - `$DATAFMT data_format`
The *data_format* can be B or b (Binary), O or o (Octal), D or d (Decimal), or H or h (Hex). If the `$DATAFMT` directive is not specified in the data file, it can be specified on the command line with the `-datafmt` command-line option. If neither is specified, the default data format is hexadecimal.
- An optional directive for specifying a default value for unspecified addresses.
`$DEFAULTVALUE default_value`

The default value can be:

- 0, 1, X, Z for Verilog
- 0, 1, X, Z, W, H, L, U, - for VHDL

If the `$DEFAULTVALUE` directive is not specified in the data file, it can be specified on the command line with the `-defval` command-line option. If neither is specified, only the memory

addresses that are specified are filled. Other addresses remain unchanged.

- One or more lines that specify:
 - The data to be loaded ([Data-Only Format](#)). You can specify the data in binary, octal, decimal, or hexadecimal format.
 - The addresses to be filled and the data to be loaded ([Address/Data Format](#)). You can specify the address and data in binary, octal, decimal, or hexadecimal format.

The memory image file can also contain comments, which begin with a pound (#) sign.

Loading Memory from a Memory Image File in data-only Format

```
memory -load memory_object -file mem_filename -start start_address \  
    [-end end_address] [-addressfmt address_format] -datafmt data_format] \  
    [-defval default_value]
```

Loading Memory from a Memory Image File in address/data Format

```
memory -load memory_object -file mem_filename \  
    [-start start_address -end end_address] \  
    [-addressfmt address_format] [-datafmt data_format] \  
    [-defval default_value]
```

Related Topics

- [Loading memory with a file in data-only format](#)
- [Loading memory with a file in address/data format](#)

Dumping Memory

Use the `-dump` option with the `memory` command to dump the contents of memory to a memory file.

You can read back the memory image file produced with `memory -dump` to load the memory. This file does not contain a `$DEFAULTVALUE` directive. The simulator dumps only the addresses that you specify so that a load in the same simulation session cannot destroy other memory locations.

Syntax

```
memory -dump [output_format] [-start start_address] [-end end_address] \  
    memory_object -file mem_filename
```

Options

`-dump`

Dumps the contents of memory into a memory file. You can specify an output format. The options are: `%h` (hexadecimal), `%d` (decimal), `%o` (octal), or `%b` (binary). `%h` is the default.

You can use the following options:

- **`-file mem_filename`**: Specifies the name of the memory image file.

If you are using the memory `-load` command, this option specifies the name of the memory image file that contains the data to be loaded. If you are using the memory `-dump` command, this option specifies the name of the output file.

- **`-start start_address`**: Specifies the address where the load should start.

If the memory image file is in data-only format, you must include the `-start start_address` option to specify the address where the dump should start. The `-end` option is not required.

If the memory image file is in address/data format, and you include the `-start` and `-end` options, the addresses specified with the options are used.

The arguments to the `-start` and `-end` options, which specify the address where the dump should start and the address where the dump should end, are integers, which you can specify in decimal, hexadecimal, or octal notation.

- **`-end end_address`**: Specifies the address where the load should end.

If the memory image file is in data-only format, you must include the `-start start_address` option to specify the address where the dump should start. The `-end` option is not required.

If the memory image file is in address/data format, `-start` and `-end` are both optional. If you include the options, the addresses specified with the options are used.

The arguments to the `-start` and `-end` options, which specify the address where the dump should start and the address where the dump should end, are integers, which you can specify in decimal, hexadecimal, or octal notation.

Examples

In the following sequence of commands, `:MEM_OBJ` is loaded using the data file. A memory `-dump` command is then executed to dump the contents of the memory to a file called `memfile.mem`.

```
xcelium> memory -load :MEM_OBJ -file mem_load.dat
xcelium> value :MEM_OBJ
```



```
("1001", "1000", "0111", "1000", "ZZZZ", "ZZZZ", "ZZZZ", "WWW", "LLLL", "0111", "0110",
"0101", "0100", "0100", "0001", "0001")
xcelium> memory -dump :MEM_OBJ -file memfile.mem
xcelium> more memfile.mem
# Memory Dump.
# Version - TOOL:          xcelium    10.20-b100
# Memory Object - :MEM_OBJ
$ADDRESSFMT H
$DATAFMT h
0/9
1/8
2/7
3/8
4/Z
5/Z
6/Z
7/X
8/0
9/7
a/6
b/5
c/4
d/4
e/1
f/1
# End of Memory Dump
```

You can include the `-start` and `-end` options to dump part of the memory to a file. For example:

```
xcelium> memory -load :MEM_OBJ -file mem_load.dat
xcelium> memory -dump :MEM_OBJ -file memfile.mem -start 0 -end 5
xcelium> more memfile.mem
# Memory Dump.
# Version - TOOL:          xcelium    10.20-b100
# Memory Object - :MEM_OBJ
$ADDRESSFMT H
$DATAFMT h
0/9
1/8
2/7
3/8
4/Z
5/Z
# End of Memory Dump
xcelium>
```

Specifying Memory File Format

You can use the `memory` command to load a Verilog or VHDL memory or to dump the contents of a specified memory object to a specified memory file. For the memory file that is generated using the `memory` command, you can specify the following types of formats:

- [Data-Only Format](#)
- [Address/Data Format](#)

Data-Only Format

In this format, the lines in the memory file specify only the data. The format is:

```
data [; data ...]
```

If the memory file is in data-only format, you must use the `-start` option to specify the first address to load. You can also specify an end address using the `-end` option.

The following is an example memory image file in data-only format:

```
# Address format
$ADDRESSFMT D
# Data format
$DATAFMT B
# Default value for unspecified addresses.
# If default value is omitted, only the memory addresses that are
# specified are filled. Other addresses remain unchanged.
$DEFAULTVALUE XXXX
# The following line loads value 0000 at the address specified with -start.
0000
# The following line loads values XXXX, 1111, and 0000 at the next three addresses.
XXXX;1111;0000
# Values for other addresses are not specified, and will be set to XXXX, the
# default value.
```

Address/Data Format

A memory image file in address/data format contains lines that specify both the addresses to be filled and data to be loaded.

The format is:

```
start_address [: end_address ]/ data [; data ...]
```

As shown by the syntax, three cases are supported:

- Single address/single data
3/X

- Single address/multiple data

3/X;1;0

- Multiple address/single data

3:5/X

The following is an example of the memory image file in the address/data formats:

```
# Address format
$ADDRESSFMT H
# Data format
$DATAFMT H
# Default value for unspecified addresses.
# If default value is omitted, only the memory addresses that are
# specified are filled. Other addresses remain unchanged.
$DEFAULTVALUE X
# The following line loads value 0 at address 0.
0/0
# The following line loads values a, b, c, d, and e at addresses 1 to 5.
1/a;b;c;d;e
# The following line loads value f at addresses a to 1f.
a:1f/f
# Values for address 6 to 9 are not specified, and will be set to X, the
# default value.
```

Related Topic

- [Memory Objects](#)

omi

The Tcl `omi` command lets you display information about model managers and instances controlled by model managers. It also lets you pass the OMI model manager run-time commands to model managers that support this capability. Some model managers provide special capabilities that enhance the usability of the models under its control. For example, a model manager might let you load the contents of a memory viewport from a file or let you dynamically control the collection of simulation data.

Use the `omi` command to:

- Display information on the model managers and model instances for the current simulation session (`-list`).
- Send commands to model managers and model instances (`-send`).

omi Command Syntax

omi

```
-list
    [-all]
    [-instance model_manager]
    [-manager]

-send
    [-all] command
    [-instance instance_name command]
    [-manager model_manager command]
```

omi Command Options

This section describes the options that you can use with the Tcl `omi` command.

- | | |
|--------------------|---|
| <code>-list</code> | Displays a list of model managers. You can use the following options: <ul style="list-style-type: none">• -all: Displays a list of all model managers and the instances managed by each model manager. The model manager aliases are also listed.• -manager: Displays a list of the model managers. The list includes the model manager aliases and the corresponding names given by the model managers.• -instance <i>[model_manager]</i>: Displays a list of all OMI instances. If you include a model manager alias, only the instances controlled by that model manager are listed. |
| <code>-send</code> | Sends the command to model managers. You can use the following options: <ul style="list-style-type: none">• -all: Sends the command to all model managers.• -manager <i>model_manager</i>: Sends the command to the specified model manager.• -instance <i>instance_name</i>: Sends the command to the specified model instance. |

omi Command Examples

The following command displays a list of the OMI model managers and instances information for the current simulation session:

```
xcelium> omi -list -all
```

The following command displays a list of the model managers. The list includes model manager aliases and the corresponding names given by the model managers:

```
xcelium> omi -list -manager
```

The following command displays a list of all OMI instances:

```
xcelium> omi -list -instance
```

The following command displays a list of all OMI instances controlled by the model manager `mm:1`:

```
xcelium> omi -list -instance mm:1
```

The following command sends the specified command to all model managers for all model instances:

```
xcelium> omi -send -all "dump mem"
```

The following command sends the command to model instances controlled by the specified model manager:

```
xcelium> omi -send -manager mm:1 "dump mem"
```

The following command sends the command to the specified model instance only:

```
xcelium> omi -send -instance top.pl "loadmem mem ./mem_file"
```

pause

The Tcl `pause` command interrupts the execution of a Tcl command script and returns control to the Tcl interpreter prompt.

This command supports two use models:

- By specifying options at the Tcl command prompt. You can:
 - Resume the execution of the last interrupted script (`pause -resume`).
 - Abort the execution of a script (`pause -abort`).
 - Display which scripts have been interrupted, and at which line number (`pause -status`).
- By specifying a Tcl command script. The simulator ignores this command if it is specified together with command options in the script. Additionally, the script containing the `pause` command must be invoked with either the Tcl `input` command or `xmsim -input` on the command line. For example:

```
xcelium> input script.tcl
```

Or:

```
% xmsim -input script.tcl snapshot_name
```

If the script is executed with the Tcl `source` command, the simulator ignores the `pause` command in the script without generating an error or warning. Additionally, the simulator ignores the `pause` command if it is the last command in a script.

When a script is interrupted by the `pause` command, control is returned to the Tcl interpreter. The command prompt includes a reminder that the script has been interrupted. The prompt looks like the following:

```
xmsim(pause 1)>
```


You can then enter Tcl commands, `set` or `unset` environment variables, invoke another script, and so on. If you invoke another script, that script may also be paused and another script invoked, up to a nesting level of 50 scripts. The prompt includes a number that indicates the current nesting level:

```
xmsim(pause 4)>
```

pause Command Syntax

```
pause -abort [<number_levels>|all]
      -resume
      -status
```

```
pause
```

 The simulator supports the `pause` command, without Tcl command options, only in a Tcl command script.

pause Command Options

This section describes the options that you can use with the Tcl `pause` command.

```
-abort
[<number_levels>|all]
```

Stops the execution of a Tcl script that has been interrupted with a `pause` command.

If the Tcl scripts are nested, you can:

- Use `-abort` (with no arguments) to abort the last script only.
- Use `-abort <number_levels>` to abort the specified number of nesting levels. The `<number_levels>` argument is an integer. The default is `-abort 1`.
- Use `-abort all` to abort all scripts.

The `abort` option has no effect if there are no scripts to abort.

| | |
|----------------------|---|
| <code>-resume</code> | Resumes execution of the last Tcl script that was paused. The <code>-resume</code> option has no effect if there are no scripts that are paused. |
| <code>-status</code> | Lists all the currently paused Tcl scripts. This list shows the name of the interrupted script(s) and the line number at which the script was interrupted. |

pause Command Limitations

- You cannot use a `pause` command in a `foreach` Tcl construct. The following message is displayed when `pause` is used inside a `foreach` construct:

```
xmsim: *N,NOPAUS: Pause-resume functionality is not supported for 'foreach' construct.
```

- You cannot use a `pause` command in a Tcl `proc`. For example, the following is not supported:

```
proc sum {arg1 arg2} {  
  puts "calling sum procedure \n";  
  pause  
  set x [expr $arg1 + $arg2];  
  return $x }  
}
```

The following message is displayed when `pause` is used inside a Tcl `proc`:

```
xmsim: *N,NOPAUS: Pause-resume functionality is not supported for 'proc' construct.
```

pause Command Examples

Example 1

The following Tcl script (`script.tcl`) contains a `pause` command to interrupt the execution of the script.

```
script.tcl  
-----  
puts "HELLO WORLD"  
pause  
puts "HELLO UNIVERSE"
```

Executing the script with the `input` command generates the following output:

```
xcelium> input script.tcl  
xcelium> puts "HELLO WORLD"  
HELLO WORLD
```

```
xcelium> pause
xmsim(pause 1)>
```

Use `pause -resume` to continue the script.

```
xmsim(pause 1)> pause -resume
xcelium> puts "HELLO UNIVERSE"
HELLO UNIVERSE
xcelium>
```

Example 2

The following example illustrates a nested pause. In this example, the simulator is invoked with the `-input` option to invoke the script `start.tcl`.

The `start.tcl` script invokes another script, called `script1.tcl`, which contains a `pause` command. At the command-line prompt, an `input` command is executed to invoke the third script, `script2.tcl`. This script is also interrupted with a `pause` command. A `pause -abort` command stops the execution of `script2.tcl`, and then a `pause -resume` command resumes the execution of `script1.tcl`.

| | |
|--|---|
| <pre>start.tcl ----- puts "START Script" input script1.tcl puts "END Script" script1.tcl ----- puts "First command in script1" pause puts "Last command in script1"</pre> | <pre>script2.tcl ----- puts "First command in script2" pause puts "Last command in script2"</pre> |
|--|---|

```
% xmsim -nocopyright -input start.tcl worklib.top
xcelium> puts "START Script"
START Script
xcelium> input script1.tcl
xcelium> puts "First command in script1"
First command in script1
xcelium> pause
xmsim(pause 1)> input script2.tcl
xmsim (pause 1)> puts "First command in script2"
First command in script2
xmsim (pause 1)> pause
xmsim(pause 2)> pause -status
      Macro 'script2.tcl' paused at line '2' (Current Macro)
      Macro 'script1.tcl' paused at line '2'
xmsim(pause 2)> pause -abort
xmsim(pause 1)> pause -status
```


Macro 'script1.tcl' paused at line '2' (Current Macro)

```
xmsim(pause 1)> pause -resume
xcelium> puts "Last command in script1"
Last command in script1
xcelium> puts "END Script"
END Script
xcelium>
```

Example 3

In this example, the Tcl script runs the simulation for 1000 ns and then pauses. Various Tcl commands are entered at the prompt before resuming the script with a `pause -resume` command.

```
% xmsim -nocopyright -input test.script worklib.top
Loading snapshot worklib.top:module ..... Done
xcelium> run 1000 ns
Ran until 1 US + 0
xcelium> pause
xmsim (pause 1)> value count
4'ha
xmsim (pause 1)> drivers count
count.....wire [3:0]
    count[3] (wire/tri) = St1
        St1 <- (top.counter.d) output port 1, bit 0 (./counter.v:19)
    count[2] (wire/tri) = St0
        St0 <- (top.counter.c) output port 1, bit 0 (./counter.v:18)
    count[1] (wire/tri) = St1
        St1 <- (top.counter.b) output port 1, bit 0 (./counter.v:17)
    count[0] (wire/tri) = St0
        St0 <- (top.counter.a) output port 1, bit 0 (./counter.v:16)
xmsim (pause 1)> stop -object count
Created stop 1
xmsim (pause 1)> pause -resume
xcelium> run
1060 NS + 1 (stop 1: top.count = b)
xcelium>
```

power

The Tcl `power` command has two modifiers `-show` and `-find`. The default action is `power -show`, although the `-show` modifier is not required. Using the default action, the following are common tasks that you can perform with this command:

- [Displaying Information About a Power Domain](#)
- [Displaying Information About a Specified Power State Table](#)
- [Displaying a List of Power Domains Within a Specified Power Mode](#)

When specifying Tcl `power -find` commands, note that only one object type option is allowed for each `power -find` specification. This alternative action enables you to locate and list all low-power objects of the specified type in the current scope. Searchable object types include power domains, supply nets, supply sets, power switches, and power state tables.

power Command Syntax

```
power -find
    [-domain |
     -aon_domain |
     -default_domain |
     -switchable_domain |
     -supply_net |
     -supply_set |
     -power_switch]
    [-pst]
        [-state]
    -object <hdl_object>
        [-domain |
         -isolation_control |
         -sr_save_control |
         -sr_restore_control]
    -pd_name <domain_name>
        [[-in_port
         -out_port] |
         -isolation_rule]
    [-recursive]
    [-newline]
    [-quiet]

power [-show] {-object <hdl_object> [<hdl_object> ...] |
             -pdname <domain_name> [<domain_name> ...]}
             [-instances]
             [-isolation_ports]
```

```
        [-sr_variables]
        [-state]
        [-verbose]

power [-show] -pst <table_name>
        [-active]
        [-state]
        [-verbose]

power [-show] -pwr_mode <mode_name> [<mode_name> ...]
```

power Command Options

This section describes the options that you can use with the Tcl `power` command.

- | | |
|--------------------|--|
| <code>-find</code> | <p>Locates and lists all low-power objects of the specified type in the current scope.</p> <p>You locate low-power objects by searching power domains or HDL objects. Searchable object types include power domains, supply nets, supply sets, power switches, or power state tables. You can choose only one object type for each power <code>-find</code> specification.</p> <p>The searchable object types are defined as follows:</p> <ul style="list-style-type: none">• <code>-domain</code>: Lists all power domains.• <code>-aon_domain</code>: Lists all domains without an explicit control condition (always on domains).• <code>-switchable_domain</code>: Lists all domains with an explicit control condition (switchable domains).• <code>-supply_net</code>: Lists all supply nets found in the IEEE 1801 design.• <code>-supply_set</code>: Lists all supply sets found in the IEEE 1801 design.• <code>-power_switch</code>: List all power switches in the low-power design. |
|--------------------|--|

- **-object <hdl_object>**: List all low-power information associated with the specified HDL object. Note that you must specify either an HDL object or a power domain name when listing the low-power information for a design. Wildcards are not supported.

The following sub-options are available when searching HDL objects using a `power -find -object` command:

- **-domain**: Lists the domain associated with the HDL object.
 - **-isolation_control**: Lists the isolation control expression associated with the HDL object.
 - **-sr_save_control**: Lists the state retention control expression associated with the HDL object.
 - **-sr_restore_control**: Lists the state retention restore control expression associated with the HDL object.
- **-pdname <domain_name>**: Lists all low-power information associated with the specified power domain. Note that you must specify either a power domain or an HDL object when listing the low-power information for a design. Wildcards are not supported.

The following sub-options are available when searching power domains using a `power -find -pdname` command:

- **-in_port**: Lists the isolated input ports in the specified power domain.
- **-out_port**: Lists the isolated output ports in the specified power domain.
- **-isolation_rule**: Lists all isolation rules associated with the power domain.

- **-pst**: Lists all power state tables in the current scope of the IEEE 1801 design.

The following sub-option is available when searching power state tables using a `power -find -pst` command:

- **-state**: Lists the state information of each power state table in the current scope.
- **-recursive**: Lists all low-power objects in the current scope, as well as those scopes below the current scope (the child scopes).
- **-newline**: Lists each found object on a separate line.
- **-quiet**: Disables warning messages when no low-power objects are found.

-show

Displays information on a particular power domain, HDL object, power state table (PST), or power mode.

Specifying this option is not required. In other words, the following power commands are identical:

```
xcelium> power -show -object inst.alu_inst.aui.br
```

and

```
xcelium> power -object inst.alu_inst.aui.br
```

-object <hdl_object>

Specifies the name of one or more HDL objects in the design. Information is displayed for the parent power domain that contains the object(s).

-pdname <domain_name>

Specifies the name of one or more power domains.

By default, with no other options, a `power -pdname` command displays the name of the specified power domain and the top instances in that domain.

The following sub-options are available when you specify a power domain with the `power [-show] -pdname` command or when you search for a power domain that contains an HDL object with the `power [-show] -object` command:

- **-instances:** Displays the name of the power domain and the top instances in that power domain. This is the default if no other option is specified.
- **-isolation_ports:** Displays the isolation ports in the power domain, the isolation status (not enabled or enabled), and the line number of the corresponding:
 - `create_isolation_rule` command in the CPF file
 - `set_isolation` command in the IEEE 1801 file
- **-sr_variables:** Displays the state retention registers and variables in the power domain, their retention status (not retained or retained), and the line number of the corresponding:
 - `create_state_retention_rule` command in the CPF file
 - `set_retention` command in the IEEE 1801 file

- **-state:** Displays the current state information of the power domain.
This includes:
 - The current state of the power domain
The state of the power domain is:
 - **ON** if the domain is in a nominal condition specified as on
(`create_nominal_condition -name name -voltage voltage -state on`).
 - **UNINITIALIZED** if the simulation is being controlled by active state conditions, and when the power domain is on but there are no active state conditions enabled to specify the nominal condition to which the domain should transition.
 - **OFF** if the domain is in a nominal condition specified as off
(`create_nominal_condition -name name -voltage voltage -state off`).
 - **STANDBY** if the domain is in a nominal condition specified as standby (`create_nominal_condition -name name -voltage voltage -state standby`).
 - **TRANSITIONING** if the domain is currently in transition from one nominal condition to another nominal condition.
 - The current nominal condition of the domain
The nominal condition of the power domain can be:
 - The name of the nominal condition if the domain is in one of the nominal conditions defined as on.
 - **SHUTOFF** if the power domain is in the OFF state.
 - **UNINITIALIZED** if the nominal condition is undefined. This can occur when the simulation is being controlled by active state conditions, and when the power domain is on but there are no active state conditions enabled to specify the nominal condition to which the domain should transition.
 - The current voltage of the domain

If the `-sr_variables` option has been used to include information on the state retention registers and variables, the saved values of any retained variables are included in the output.

- **-verbose:** Adds the following information to the output:
 - The IEEE or CPF file, and line number of the `create_power_domain` command for the power domain.
 - The names of any mapped domains for the specified power domain, and the IEEE 1801 or CPF files and line numbers where the mapped domains are defined.
 - The names of any base domains of the specified power domain, and the IEEE 1801 or CPF files and line numbers where the base domains are defined.
 - The name of the power mode control group for the specified power domain, and the IEEE 1801 or CPF file and line number where the control group is defined.
 - The slope defined for the power domain. The slope is displayed as volts per time unit, where the time unit is the time resolution of the simulation.
 - If both a minimum and maximum slope are defined, the output shows the current, minimum, and maximum slope.
 - If only the maximum slope is defined, only the current slope is displayed.
 - If no slope is defined, the current slope is shown as "infinite".

`-pst <table_name>`

Displays information for the specified power state table (PST). The specified table name is specific to the current scope. It can be a full or relative pathname.

The following sub-options are available when you specify a PST with the `power [-show] -pst` command:

- **-active:** Displays only the active states for the specified PST, and the corresponding voltage values for each active state.
- **-state:** Displays the current state information for the defined power states.

- **-verbose**: Lists the file and line number where each state is defined in the IEEE 1801 file, as well as a file and line number reference for the supplies associated with the specified PST.

`-pwr_mode <mode_name>`

Specifies the name of a power mode. This option displays a list of power domains within the specified power mode, along with their corresponding nominal condition names and voltage values.

power Command Examples

The following command displays the top instances for the power domain `PD_compress`.

```
xcelium> scope dut
xcelium> power -pdname PD_compress
Power Domain tb.dut.PD_compress
Top instances:
    tb.dut.inst_compress
```

The following command displays the top instances for the power domains `PD_default` and `PD_compress`.

```
xcelium> power -pdname PD_default PD_compress
Power Domain tb.dut.PD_default
    Power Domain is a default domain
    Top instances:
        tb.dut

Power Domain tb.dut.PD_compress
    Top instances:
        tb.dut.inst_compress
```

The following command displays the top instances for all power domains in the current scope.

```
xcelium> power -pdname *
```

The following command displays all of the information for the power domain `PD_compress`.

```
xcelium> power -pdname PD_compress -instances -isolation_ports -sr_variables -state -verbose
Power Domain tb.dut.PD_compress is ON, nominal condition is NC_08, voltage = 0.800000
Power domain rule: file ./test_mvs.cpf line 31
Base Domains:
    tb.dut.PD_default
        Power domain rule: file ./test_mvs.cpf line 27
Control Group:
    tb.dut (default)
        Control group rule: file ./test_mvs.cpf line 9
Current Slope: 0.200000 volt/ns
```

```
Top instances:
  tb.dut.inst_compress
Isolation ports:
  tb.dut.inst_compress.out4
    Status is: not enabled
    Isolation rule: file ./test_mvs.cpf line 63
  tb.dut.inst_compress.out3
    Status is: not enabled
    Isolation rule: file ./test_mvs.cpf line 63
  tb.dut.inst_compress.out2
    Status is: not enabled
    Isolation rule: file ./test_mvs.cpf line 63
  tb.dut.inst_compress.out1
    Status is: not enabled
    Isolation rule: file ./test_mvs.cpf line 63
State Retention variables and registers:
  tb.dut.inst_compress.out4 (saved value = 1'h0)
    State Retention rule: file ./test_mvs.cpf line 69
    Status is: retained
  tb.dut.inst_compress.out3 (saved value = 1'h0)
    State Retention rule: file ./test_mvs.cpf line 69
    Status is: retained
  tb.dut.inst_compress.out2 (saved value = 1'h1)
...

```

The following command displays the power domain that contains the object `TESTBENCH.inst.alu_inst.aui.br` and the top instances in the power domain. The `-instances` option is the default if no other option is specified. This command is the same as `power -object TESTBENCH.inst.alu_inst.aui.br`.

```
xcelium> power -object -instances TESTBENCH.inst.alu_inst.aui.br
Power Domain TESTBENCH.inst.PDau
Top instances:
  TESTBENCH.inst.alu_inst.aui

```

The following command displays the power domain that contains the objects `inst.alu_inst.aui.br` and `inst.alu_inst.aui.z`.

```
xcelium> power -object inst.alu_inst.aui.br inst.alu_inst.aui.z
Power Domain TESTBENCH.inst.PDau
Top instances:
  TESTBENCH.inst.alu_inst.aui

```

The following command displays the power domain that contains the object and the isolation ports for that power domain. The isolation status and the line number of the corresponding `create_isolation_rule` command in the CPF file is displayed.

```
xcelium> power -object TESTBENCH.inst.alu_inst.aui.br -isolation_ports

```

```
Power Domain TESTBENCH.inst.PDau
Isolation ports:
  TESTBENCH.inst.alu_inst.aui.z
    Status is: not enabled
    Isolation rule: file nano.cpf line 54
```

The following command includes the `-sr_variables` option, which displays the state retention registers and variables in the power domain, their retention status, and the line number of the corresponding `create_state_retention_rule` command in the CPF file.

```
xcelium> power -object TESTBENCH.inst.alu_inst.aui.br -sr_variables
Power Domain TESTBENCH.inst.PDau
State Retention variables and registers:
  TESTBENCH.inst.alu_inst.aui.z
    State Retention rule: file nano.cpf line 44
    Status is: not retained
```

Related Topic

- [Displaying and Listing Low-Power Information](#)

Displaying Information About a Power Domain

Use the `-object` or `-pdname` modifier with the `power [-show]` command to display information on a particular power domain.

- Specifying `-object` searches for a power domain that contains the given HDL object.
- Specifying `-pdname` searches for a power domain that matches the given hierarchical path.

Syntax

This section lists the `power` command syntax to display information about a domain.

```
power [-show] {-object <hdl_object> [<hdl_object> ...] |
  -pdname <domain_name> [<domain_name> ...]}
  [-instances]
  [-isolation_ports]
  [-sr_variables]
  [-state]
  [-verbose]
```

Options

`-show`

Displays information on a particular power domain, HDL object, power state table (PST), or power mode.

Specifying this option is not required. In other words, the following power commands are identical:

```
xcelium> power -show -object inst.alu_inst.aui.br
```

and

```
xcelium> power -object inst.alu_inst.aui.br
```

`-object <hdl_object>`

Specifies the name of one or more HDL objects in the design. Information is displayed for the parent power domain that contains the object(s).

`-pdname <domain_name>`

Specifies the name of one or more power domains.

By default, with no other options, a `power -pdname` command displays the name of the specified power domain and the top instances in that domain.

The following sub-options are available when you specify a power domain with the `power [-show] -pdname` command or when you search for a power domain that contains an HDL object with the `power [-show] -object` command:

- **-instances:** Displays the name of the power domain and the top instances in that power domain. This is the default if no other option is specified.
- **-isolation_ports:** Displays the isolation ports in the power domain, the isolation status (not enabled or enabled), and the line number of the corresponding:
 - `create_isolation_rule` command in the CPF file
 - `set_isolation` command in the IEEE 1801 file
- **-sr_variables:** Displays the state retention registers and variables in the power domain, their retention status (not retained or retained), and the line number of the corresponding:
 - `create_state_retention_rule` command in the CPF file
 - `set_retention` command in the IEEE 1801 file

- **-state**: Displays the current state information of the power domain.
This includes:
 - The current state of the power domain
The state of the power domain is:
 - **ON** if the domain is in a nominal condition specified as on
(`create_nominal_condition -name name -voltage voltage -state on`).
 - **UNINITIALIZED** if the simulation is being controlled by active state conditions, and when the power domain is on but there are no active state conditions enabled to specify the nominal condition to which the domain should transition.
 - **OFF** if the domain is in a nominal condition specified as off
(`create_nominal_condition -name name -voltage voltage -state off`).
 - **STANDBY** if the domain is in a nominal condition specified as standby (`create_nominal_condition -name name -voltage voltage -state standby`).
 - **TRANSITIONING** if the domain is currently in transition from one nominal condition to another nominal condition.
 - The current nominal condition of the domain
The nominal condition of the power domain can be:
 - The name of the nominal condition if the domain is in one of the nominal conditions defined as on.
 - **SHUTOFF** if the power domain is in the OFF state.
 - **UNINITIALIZED** if the nominal condition is undefined. This can occur when the simulation is being controlled by active state conditions, and when the power domain is on but there are no active state conditions enabled to specify the nominal condition to which the domain should transition.
 - The current voltage of the domain

If the `-sr_variables` option has been used to include information on the state retention registers and variables, the saved values of any retained variables are included in the output.

- **-verbose:** Adds the following information to the output:
 - The IEEE or CPF file, and line number of the `create_power_domain` command for the power domain.
 - The names of any mapped domains for the specified power domain, and the IEEE 1801 or CPF files and line numbers where the mapped domains are defined.
 - The names of any base domains of the specified power domain, and the IEEE 1801 or CPF files and line numbers where the base domains are defined.
 - The name of the power mode control group for the specified power domain, and the IEEE 1801 or CPF file and line number where the control group is defined.
 - The slope defined for the power domain. The slope is displayed as volts per time unit, where the time unit is the time resolution of the simulation.
 - If both a minimum and maximum slope are defined, the output shows the current, minimum, and maximum slope.
 - If only the maximum slope is defined, only the current slope is displayed.
 - If no slope is defined, the current slope is shown as "infinite".

Examples

Consider a design with a top-level module `tb_top` that instantiates a `dut` HDL object. You might use the Tcl `power` command below to search for a power domain within this `dut` object and display information on that power domain.

```
xcelium> power -object dut
Power Domain tb_top.dut.domain_aon_top1
Power Domain is a default domain
Top instances:
  tb_top.dut
```

This next example adds the `-verbose` option to the previous command, which displays the file and line number referencing the location to the `create_power_domain` command that defines `domain_aon_top1`.

```
xcelium> power -object dut -verbose
Power Domain tb_top.dut.domain_aon_top1
```

```
Power domain rule: file ../upf/top.upf line 24
Power Domain is a default domain
Top instances:
    tb_top.dut
```

This example is similar to the one above but uses the `-pdname` option to search for the domain using a full hierarchical path argument.

```
xcelium> power -pdname tb_top.dut.domain_aon_top1 -verbose
Power Domain tb_top.dut.domain_aon_top1
Power Domain is a default domain
Top instances:
    tb_top.dut
```

Displaying Information About a Specified Power State Table

Use the `-pst` option with the `power [-show]` command to display information for the specified power state table (PST).

This option also lists the supplies associated with the PST, each power state name, and the state of each power state.

Syntax

This section lists the `power` command syntax to display information about a PST.

```
power [-show] -pst <table_name>
        [-active]
        [-state]
        [-verbose]
```

Options

`-show`

Displays information on a particular power domain, HDL object, power state table (PST), or power mode.

Specifying this option is not required. In other words, the following power commands are identical:

```
xcelium> power -show -object inst.alu_inst.aui.br
```

and

```
xcelium> power -object inst.alu_inst.aui.br
```

`-pst <table_name>`

Displays information for the specified power state table (PST). The specified table name is specific to the current scope. It can be a full or relative pathname.

The following sub-options are available when you specify a PST with the `power [-show] -pst` command:

- **-active:** Displays only the active states for the specified PST, and the corresponding voltage values for each active state.
- **-state:** Displays the current state information for the defined power states.
- **-verbose:** Lists the file and line number where each state is defined in the IEEE 1801 file, as well as a file and line number reference for the supplies associated with the specified PST.

Examples

In the first example, a `Tcl scope` command sets the debug scope to `tb_top.dut`. The `power -pst` command then displays general information on `PST1`, the specified power state table.

```
xcelium> scope -set tb_top.dut
xcelium> power -pst PST1
Power State Table tb_top.dut.PST1
Supplies:
    VDD1 VDD1_SW VDD2 VSS
State SON:
    v1_08 v1_08 v1_08 v0
State S_some_on:
    v1_08 voff v1_08 v0
State SOFF:
    voff voff voff v0
```

This next example specifies `tb_top.dut.PST1`, the full hierarchical path to `PST1`. The `-verbose` option enables the simulator to list the 1801 file and line number for the supplies and power states shown.

```
xcelium> power -pst -verbose tb_top.dut.PST1
Power State Table tb_top.dut.PST1
Supplies: file ../upf/top.upf line 100
    VDD1 VDD1_SW VDD2 VSS
State SON: file ../upf/top.upf line 102
    v1_08 v1_08 v1_08 v0
State S_some_on: file ../upf/top.upf line 103
```



```
v1_08 voff v1_08 v0
State SOFF: file ../upf/top.upf line 104
voff voff voff v0
```

In this example, adding the `-state` option to the `power -pst -verbose` command displays the current status of the power states `SON`, `S_some_on`, and `SOFF`. These states are defined as a part of `PST1` using the 1801 `add_pst_state` command.

```
xcelium> power -pst -verbose -state tb_top.dut.PST1
Power State Table tb_top.dut.PST1
Supplies: file ../upf/top.upf line 100
VDD1 VDD1_SW VDD2 VSS
State SON: file ../upf/top.upf line 102
Status is: Active
v1_08 v1_08 v1_08 v0
State S_some_on: file ../upf/top.upf line 103
Status is: Off
v1_08 voff v1_08 v0
State SOFF: file ../upf/top.upf line 104
Status is: Off
voff voff voff v0
```

In this example, adding the `-active` option to the `power -pst` command displays only the power states in `PST1` that are currently active. In this case, the power state `SON` is shown. The voltage values corresponding to this power state are also listed for the supply nets `VDD1`, `VDD1_SW`, `VDD2`, and `VSS`.

```
xcelium> power -pst -active tb_top.dut.PST1
Power State Table tb_top.dut.PST1
Supplies:
VDD1 VDD1_SW VDD2 VSS
State SON:
{1.080000, 1.080000, 1.080000, 0.000000}
```

If the `-verbose` option is used with this `power -pst -active` command, the simulator lists the 1801 file references for the supply nets and the active power state `SON`. The output for `SON` includes a list of the supply net state values as well as the corresponding voltage values.

```
xcelium> power -pst -active -verbose tb_top.dut.PST1
Power State Table tb_top.dut.PST1
Supplies: file ../upf/top.upf line 100
VDD1 VDD1_SW VDD2 VSS
State SON: file ../upf/top.upf line 102
{v1_08 = 1.080000, v1_08 = 1.080000, v1_08 = 1.080000, v0 = 0.000000}
```

Displaying a List of Power Domains Within a Specified Power Mode

Use the `-pwr_mode` option with the `power [-show]` command when simulating a CPF design to display a list of power domains within the specified power mode(s), along with their corresponding nominal condition names and voltage values.

Syntax

This section lists the `power` command syntax to display domain information on the specified power mode(s).

```
power [-show] -pwr_mode <mode_name> [<mode_name> ...]
```

Options

`-show`

Displays information on a particular power domain, HDL object, power state table (PST), or power mode.

Specifying this option is not required. In other words, the following power commands are identical:

```
xcelium> power -show -object inst.alu_inst.aui.br
```

and

```
xcelium> power -object inst.alu_inst.aui.br
```

`-pwr_mode <mode_name>`

Specifies the name of a power mode. This option displays a list of power domains within the specified power mode, along with their corresponding nominal condition names and voltage values.

Example

In the following example, the CPF file includes a `create_power_mode` command that defines a power mode called `M3`. This power mode consists of power domain `pdT` at nominal condition `NC_12` (1.2 V), power domain `pdA` at nominal condition `NC_12` (1.2 V), and power domain `pdB` at nominal condition `NC_08` (.8 V).

```
create_power_mode -name M3 \  
                  -default \  
                  -domain_conditions {pdT@NC_12 pdA@NC_12 pdB@NC_08}
```

```
xcelium> power -pwr_mode M3
```

```
Power mode top.M3
```

```
Domains:
```

```
pdT
```

```
Nominal condition is (NC_12): voltage = 1.200000
pdA
Nominal condition is (NC_12): voltage = 1.200000
pdB
Nominal condition is (NC_08): voltage = 0.800000
```

probe

The Tcl `probe` command controls the values of certain simulation objects that you can then save to a Simulation History Manager (SHM), Value Change Dump (VCD), or Extended Value Change Dump (EVCD) database. Verilog, VHDL, and mixed-language designs are supported.

The objects that you probe must have read access. If you specify an object as an argument to the `probe` command, and that object does not have read access, the simulator prints an error message. If you specify a scope as an argument to the `probe` command, the objects within that scope that do not have read access are excluded from the probe, and the simulator prints a warning message.

If you do not specify an argument, the current debug scope is assumed by the simulator. In this case, you must include an additional option that specifies the objects to include in the trace (`-all`, `-inputs`, `-outputs`, or `-ports`).

Be aware that, for Verilog, you can set an EVCD probe only on scopes. You cannot set a probe on specific Verilog ports. Because the top-level scope in Verilog does not have ports, you must specify a scope as the argument.

If the scope name and a signal name are the same, and you specify that name, the `probe` command probes the signal rather than signals in the scope.

The following are other common tasks that you can perform using this command:

- [Deleting a Probe](#)
- [Deactivating a Probe](#)
- [Enabling a Probe](#)
- [Displaying Probe Information](#)

probe Command Syntax

```
probe [-create] [{<object>|<scope_name>}...]
      [{ -shm |
        -vcd |
        -evcd
          [{splitio | simple}]
          [[-mode {lfcompat | lfvectcompat}]] |
```

```

        [-evcdfformat <format_number>]] |
        [-evcd_supply_strength]] |
    -database <dbase_name>}}
        [-activations [-depth {n | all}] [-future_processes]
[-aicms]
[-all [-memories] [-variables] [-generics] [-sc_processes]] [-dynamic]
[-assertions {-assertdebug [-format <format>]] [-failure] [-state] [-signals]]
[-depth {<n> | all | to_cells}]
[-disable_packed_trans]
[-domain {analog | digital}]
[-emptyok]
[-enable_packed_trans]
[-exclude {<object_name> | <scope_name>}]
[-exclude_list <filename> ]
[-exclude_type {buffer , inout}]
[-flow]
[-functions]
[-inhconn_signal <global_signal>]
[-inputs]
[-name <probe_name>]
[-outputs]
[-packed <limit>]
[-ports]
[-power]
[-pwr_mode]
[-qda_max_elem <num|all|0>]
[-screen [-format <format_string>] [-redirect <file>] [-append] <objects>]
[-scope_list <filename>]
[-sr_save]
[-sr_all_save]
[-string_max_len <size>]
[-tasks]
[-transaction]
[-unpacked limit]
[-uvm]
[-vspice_cell]
[-waveform]

-delete <probe_name> [<probe_name>...]
-disable <probe_name> [<probe_name>...]
-enable <probe_name> [<probe_name>...]
-save [<filename>]
-show [<probe_name>...] [-database <dbase_name>]


```

probe Command Options

This section describes the options that you can use with the Tcl `probe` command.

`-create [{<object>|<scope_name>}...]`

Places the values of the specified simulation objects in a database. The simulation objects must have read access.

 This modifier is optional when using the `probe` command.

If you are probing to an SHM or VCD database, or if you are probing VHDL objects to an EVCD database, you should specify one of the two supported arguments.

- `<object>`: Indicates the object(s) to be traced.
- `<scope_name>`: Indicates the scope(s) to be traced.

It is legal to specify multiple arguments of the same type or a combination of object(s) and scope(s). If you do not specify an argument, the current debug scope is assumed.

You can also use this option to record statement trace data into an SHM database that was opened with a `database -statement` command. All statements within a specified scope are traced. After writing statement trace data to the SHM database, use the *Explore - Go To - Cause* command in the SimVision waveform viewer to track down the cause of a signal transition with either the Source Code Browser or the Trace Signals sidebar. See [Finding the Cause of a Signal Transition](#) for more information.

If you do not specify an `object` or `scope_name` argument, you must use one of the following options: `-inputs`, `-outputs`, `-ports`, or `-all`. You must specify the database type (`-shm`, `-vcd`, `-evcd`) or the database name, if more than one database is open.

You can use wildcard characters in the argument to a `probe -create` command if the argument is a Verilog or VHDL object name.

- The asterisk (`*`) matches any number of characters.
- The question mark (`?`) matches any one character.

You cannot use wildcard characters in scope specifiers or inside escaped names.

`-shm`

Sends the probe to the default SHM database if more than one database is open.

If no default database exists, the simulator opens a default database called `xcelium.shm` and saves the file to the current working directory.

If you are running the simulator on a machine with multiple CPUs, you can improve the performance of waveform dumping and decrease peak memory consumption by using the `-mcdump` option when you invoke the simulator (`xrun -mcdump` or `xmsim -mcdump`). In multi-process mode, the simulator forks off a separate executable called `xmdump`, which performs some of the processing for waveform dumping. The `xmdump` process runs in parallel on a separate CPU until the main process exits.

`-vcd`

Sends the probe to the default VCD database if more than one database is open.

If no default database exists, the simulator opens a default database called `xcelium.vcd`. and saves the file to the current working directory.

```
-evcd [{splitio|simple}]  
[[-mode {lfcompat|lfvectcompat}]] |  
[-evcdformat <format_number>]] |  
[-evcd_supply_strength]]
```

Sends the probe to the default EVCD database if more than one database is open.

By default, the simulator dumps final value changes in extended format for both vector and scalar ports. For vector ports, the simulator dumps value changes for the entire vector, not for individual bits of the vector. If no default database exists, the simulator opens a default database called `xcelium.evcd` and saves the file to the current working directory.

There are two arguments and three sub-options that you can use with this to control the EVCD dump. These are defined below:

- **splitio:** This argument dumps the value of the input port if any driver into the design under test changes value; it dumps the value of the output port if any driver within the design under test changes value. If both inside and outside drivers change, it dumps two values: one corresponding to the input and one corresponding to the output.
- **simple:** This argument dumps resolved signal values on every transaction on the signal connected to a given primary port in simple 0x1z format.
- **-mode {lfcompat|lfvectcompat}:** This sub-option applies to VHDL only and is available for backward compatibility.

There are two possible arguments to choose from:

- **lfcompat:** Dumps vector ports as individual elements and the strength mapping is the same as it was in legacy releases. You must use `-mode lfcompat` if you want to dump a subelement of a compressed VHDL signal.
- **lfvectcompat:** Dumps vector ports as complete vectors, and the strength mapping is the same as it was in legacy releases. For more information on generating an EVCD file for VHDL and for details on VHDL strength mapping, see [The Extended Value Change Dump \(EVCD\) File](#).

If the `-mode` option and the `-evcdformat` option are both included on the command line, a warning message is issued. The VHDL scope is dumped in the format specified with the `-mode` option and the Verilog scope is dumped in the format specified with `-evcdformat`.

- **-evcdformat <format_number>:** Specifies a format for

the EVCD database.

The *format_number* argument can be 0, 1, 2, or 3.

- 0: The default behavior. Reports the strengths for both the zero and one components of the value if the strengths are the same. If the strengths are different, report only the winning strength. That is, the two strength values either match (for example, `pA 5 5 !`) or the winning strength is shown and the other is zero (for example, `pH 0 5 !`).
- 1: Keeps the losing value. Reports the strengths for both the zero and one components of the value (for example, `pD 6 5 !`).
- 2: Generates output according to the IEEE 1364-2001 standard.

The IEEE standard states that the values 0 (both input and output are active with value 0) and 1 (both input and output are active with value 1) are conflicting states. The standard then defines two strength ranges:

Strong: strengths 7, 6, and 5

Weak: strengths 4, 3, 2, 1

The rules for resolving conflicts are:

- If the input and output are driving with the same range of strength, the resolved value is 0 or 1, and the strength is the stronger of the two.
- If the input is driving a strong strength and the output is driving a weak strength, the resolved value is d or u, and the strength is the strength of the input.
- If the input is driving a weak strength and the output is driving a strong strength, the resolved value is l or h, and the strength is the strength of

the output

This format flag also changes the default behavior so that range information is included for non-scalar ports. If you do not specify this value, only size information is included. For example, suppose you have the following declaration:

```
input [31:1] my_wire;
```

In pre-2001 mode (that is, without specifying the flag), the EVCD output is:

```
$var port 31 ! my_wire $end
```

In 2001 mode, the output is:

```
$var port [31:1] ! my_wire $end
```

- **3:** Do both 1 and 2. Generate output according to the IEEE 1364-2001 standard, and also keep the losing strength.
- **-evcd_supply_strength:** Considers the driver as unidirectional to the port if the port is driven by a driver with supply strength.

If the `-evcd_supply_strength` switch is used without the parent option `-evcd`, the tool throws an error (PRBEVSU).

`-database <dbase_name>`

Sends the probe to the specified database.

If more than one database is open, you must specify the database into which you want to dump values. The `dbase_name` argument is the logical name of the database in which you want to save the probe. This database must already exist and must be open. This option does not create a database for you. If only one database is open, you do not need to specify a database.

```
-activations [-depth {<n>|all}]  
[-future_processes]
```

Creates a summary probe of the SystemC processes in the specified scope(s), or in the current debug scope if no scope is specified.

This option records the names of the SystemC processes into an SHM database in the order of their activations. The probe values are stored in sequence time. See the section "Thread Manager in Tcl Command Mode" in the chapter "Debugging SystemC Models" in the SystemC Simulation User Guide for more information.

The following sub-options are supported:

- **-depth {<n>|all}**: Specifies the number of hierarchical levels to traverse to collect the SystemC processes. The default value is 1.
- **-future_processes**: Includes the new SystemC processes created in the scope(s) after the probe is created.

```
-all [-memories] [-variables]  
[-generics] [-sc_processes]  
[-dynamic]
```

Specifies that all the declared objects within a scope, except for Verilog memories, VHDL variables, SystemC `ncsc_viewable` objects, and SystemC processes are to be included in the probe. This applies to:

- The scope(s) named as the probe argument(s).
- The current debug scope if no scope or object is named.
- The subscopes that you specify with the -depth option

For EVCD, use this option (or the `-ports` option) to probe all the declared primary ports within a scope.

The following sub-options are supported:

- **-memories:** Use `-all -memories` to include Verilog memories in the probe. You can also probe a memory by:
 - Including the name of the memory as an argument to the `probe` command. Probing of individual memory elements or ranges of memory elements is supported.
 - Using the `$shm_probe` system task in your Verilog code. See [Managing the SHM Database](#) for details. Verilog memories cannot be dumped to EVCD databases. The simulator generates an error message if you use the `-memories` option when creating a probe to an EVCD database.
- **-variables:** Use `-all -variables` to include VHDL variables or SystemC `ncsc_viewable` objects in the probe.

This probes all objects, including variables, in the specified scope and in all sub-processes of the probed scope. When probing a VHDL process, the `-variables` option is not required because you can only probe variables in this scenario. In other words, the behavior of the following two commands is identical:

```
xcelium> probe -create {:$PROCESS_000} -all -  
variables -database waves  
xcelium> probe -create {:$PROCESS_000} -all -  
database waves
```

VHDL variables cannot be dumped to EVCD databases. In this case, only the primary ports of the component instances are probed.

- **-generics:** Use `-all -generics` to include VHDL generics in the probe.

Generics of a composite type are not included when using this sub-option.

- **-sc_processes:** Use `-all -sc_processes` to include SystemC processes in the probe.
- **-dynamic:** Use `-all -dynamic` to enable probing of SV dynamic types like classes and QDA.

This is disabled by default to prevent increasing the size of the created SHM data. Additionally, in case you have a dynamic object and have not specified this flag, a notification is printed that the `-all` command excludes dynamic types by default.

```
-assertions {-assertdebug -format  
<format|string>} [-failure] [-  
signals] [-state]
```

Creates probes on assertions.

If you have PSL assertions in your design and have compiled with the `-assert` option, or SystemVerilog assertions and have compiled with the `-sv` option, you can use the Tcl `probe` command with this option to create different kinds of assertion probes. Be aware that you can only specify one type of assertion probe when using the Tcl `probe` command. The `-failure`, `-state`, and `-transaction` options are mutually exclusive.

The following sub-options enable related features and different assertion probe types::

- **-assertdebug:** Enables additional debug for assertions with transactions for all attempts and local variables wherever applicable.

To control the resulting output, add the following option to the Tcl command line:

- **-format:** Formats the output to be printed. The format can be specified as a simple *format* specification or as a *string* that contains both text and specifiers to determine the display of objects, Valid formats for the screen are `%c`, `%s`, `%d`, `%x`, `%o`, `%C`, `%D`, and `%T`. Valid formats for `-assertdebug` are `%o`, `%b`, `%d`, and `%x`.

- **-failure:** Records assertion failures.
- **-signals:** Creates a signal probe, which records the signals that are referenced by the assertions. If assertion signals are not probed, the Waveform Window does not show any data for these signals when running the simulation using the GUI.
- **-state:** Creates a state probe, which records changes in the assertion state for the probed assertions. This is the default if `-failure` or `-transaction` is not used.

For instance, the following command records all assertions in a hierarchy as state probes without having to name each assertion:

```
xcelium> probe -create -shm -assertions -state  
-depth all
```

`-database <dbase_name>`

Saves the probe into the specified database.

If only one database is open, you do not need to specify a database. If more than one database is open, you must specify the database into which you want to dump values. You can do this by specifying a database name with this option. The `dbase_name` argument references the logical name of the database in which you want to save the probe and this database must be open. This option does not create a database for you.


You can also specify the database into which you want to dump values with the `-shm`, `-vcd`, or `-evcd` option. These options specify that you want to save the probe to the default SHM, VCD, or EVCD database, respectively.

`-depth {<n>|all|to_cells}`

Specifies how many scope levels to descend when searching for objects to probe if you have specified a scope.

When using this option, you must specify one of the following arguments:

- **<n>**: Descends the specified number of scope levels. For instance, `-depth 1` means include only the given scope, `-depth 2` means include the given scope and its subscopes, and so on. The default is 1.
- **all**: Includes all scopes in the hierarchy below the specified scope(s).
- **to_cells**: Includes all scopes in the hierarchy below the specified scope(s), but stop at cells (Verilog modules with ``celldefine` or VITAL entities with VITAL `Level0` attribute).

 If you are running the simulator in multi-step invocation mode, you must use the `-libcell` option to tag Verilog module instances as cell instances. However, if you are running the simulator in single-step invocation mode using the `xrun` command, this `-libcell` functionality is switched on by default to tag modules extracted from libraries (from `-y`, `-v`, or ``uselib`) as cells, as if the source code contained a ``celldefine` directive. You can use the `-nolibcell` option to override this behavior.

`-domain {analog|digital}`

Specifies whether probes should be set up on analog or digital signals.

By default, both analog and digital signals are probed. Choose one of the following arguments to change the default behavior:

- **analog**: Probes analog signals only.
- **digital**: Probes digital signals only.

`-emptyok`

Turns off the PRNONE error message issued when there is nothing for a `probe -create` command to probe in the specified scope.

`-exclude {object_name|scope_name}`

Excludes the specified object or the specified scope from the probe. This option works with SHM, VCD, EVCD, or screen probes.

In certain cases, writing a probe command to probe all of the objects that you want to probe (without probing unwanted objects) can be difficult. This option provides a convenient way to exclude certain objects or scopes when creating a probe. Instead of specifying all of the objects or scopes that you want to probe, you can specify the objects or scopes *not* to probe. For example, you might probe all objects in the design except objects in certain instances, or probe all levels of a certain instance, but not certain instances within those levels.

The argument to the `-exclude` option is either an *object_name* or a *scope_name*. You can use multiple `-exclude` options to exclude multiple objects or scopes. For example:

```
xcelium> probe -vcd muxdff -depth all -exclude  
dff_g1 -exclude mux_m1:d*
```

You can also exclude multiple objects or scopes by enclosing the arguments in curly braces. For example:

```
xcelium> probe -vcd muxdff -depth all -exclude  
{dff_g1 mux_m1:d*}
```

When a scope is excluded, all objects inside the specified scope and in all of its subscopes are excluded.

The wildcard characters `*` and `?` are allowed for object names only and not for scope names. In order to exclude a scope from the probe, you must specify the complete path of the scope. For example, the following command probes everything except objects inside scope `muxdff`.

```
xcelium> probe -evcd -depth all -all : -exclude  
:tb:muxdff
```

The following command probes everything except objects whose name start with `tb`. However, any scope name starting with `tb` is probed.


```
xcelium> probe -vcd -depth all -all : -exclude tb*
```

Another method to exclude objects or scopes from a probe is by setting the `probe_exclude_patterns` variable. This variable takes a *value* argument that defines a list of patterns to exclude from subsequent probe commands. For example:

```
set probe_exclude_patterns {_zy*}
```

Setting this variable provides an easy way to filter out all objects with names that follow a well-defined pattern, regardless of where the objects are in the hierarchy.

```
-exclude_list <filename>
```

Specifies a list of design objects or scope instances that are *not* be probed as part of the current `probe -create` command.

The specified file includes one *design object* per line.

`-exclude_type {buffer, inout}`

Excludes the specified object type from the probe. This option works with SHM, VCD, or EVCD probes.

In larger designs, using the Tcl `probe` command to dump all input and output ports along with all inout ports and buffers can become onerous. This option provides a way to exclude particular object types when creating a probe. Currently, this option only supports the following object types:

- VHDL buffers
VHDL and Verilog inout ports

Use the two arguments, `buffer` and `inout`, to exclude an object type. You can use multiple `-exclude_type` options to exclude multiple types on one line. For example:

```
xcelium> probe: -shm -all -variables -exclude_type  
buffer -exclude_type inout
```

You can also exclude multiple types by enclosing both arguments in curly braces:

```
xcelium> probe: -shm -all -variables -exclude_type  
{buffer inout}
```

If you use multiple options with the `probe` command, then the type specified by this option is excluded regardless of the other options specified on the command line. For example, the following probes all signals and ports except buffers:

```
xcelium> probe: -shm -all -variables -exclude_type  
buffer
```

`-flow`

Specifies that the probe is for currents, rather than voltage.


Specifying `-flow -all` saves port currents and all other values and quantities covered by the `-all` option (but not including probing of currents though inherited connections). Specifying `-flow` with `-ports`, `-inputs`, or `-outputs` probes currents in the specified objects of the scope.

See the *Virtuoso AMS Designer Simulator User Guide*, Appendix B ("Tcl-Based Debugging") for more details on this option.

`-functions`

FUNCTIONS

Includes objects declared within Verilog function scopes. By default, these objects are not included in the probe.

 If a Verilog function scope is specified as the argument to the `probe` command (either explicitly or implicitly as the current debug scope), this option is not required.

- The `probe -create` command uses an existing `-functions` option to indicate that all static functions encountered while traversing the design for this command invocation, registers a callback against the 'subprogram exit' snare providing for the capture of return values and calling scope information.
- The `probe -delete` command removes any callbacks registered against the 'subprogram exit' snare which are associated with the labeled probe.
- The `probe -enable` and `probe -disable` commands support function return values.

The `-functions` option also allows you to probe the return value of functions into an SST2 database when the database is opened in `-event` mode and `xmvlog` is invoked with `-linedebug`. Every TCL probe command that uses the `-function` option, and does not have the corresponding database opened in `-event` mode, generates a warning. Similarly, if the function definition is compiled without `-linedebug`, this also generates a warning.

The use of `-event` when opening a new database ensures that multiple calls to the same function within a given time slice are recorded as separate transactions. Return Value information is written to the SST2 database in the form of a `dwComposite` variable whose elements are variables with types defined using a `dwRecordType` that contains:

- The function return value (datatype determined per function)
- The `dwScopeHandle` of the calling scope

(dwScopeHandle)

- The line number of the function call (unsigned int)
- The line position of the function call (unsigned int)

Limitations:

- The `probe -create` command supports this feature for SHM probes only.
- Only static functions are supported.

`-inhconn_signal <global_signal>`

Returns the total current drawn from the `global_signal` through inherited connections by the specified instance.

This option must be specified together with the `-flow` option on the `probe` command line.

See the *Virtuoso AMS Designer Simulator User Guide*, Appendix B ("Tcl-Based Debugging") for more details on this option.

`-inputs`

Specifies that all inputs within a scope are to be included in the probe. This applies to:

- The current debug scope (if no scope(s) or object(s) is named in an argument)
- The scope(s) named in the argument
- Subscopes that you specify with the `-depth` option

`-name <probe_name>`

Specifies a user-defined name for the probe. You can then use the name that you assign to a probe with the `-disable`, `-enable`, `-delete`, and `-show` modifiers.

If you do not use this option to name your probes, the simulator gives every probe that you create a sequential number.

`-outputs`

Specifies that all outputs within a scope are to be included in the probe. This applies to:

- The current debug scope (if no scope(s) or object(s) is named in an argument)
- The scope(s) named in the argument
- Subscopes that you specify with the `-depth` option

`-packed <limit>`

Specifies the limit for packed arrays.

This option applies only to SystemVerilog static arrays. Dynamic arrays, queues, and queues of dynamic arrays (QDAs) are not supported.

This option sets a size limit to packed arrays and enables the display of a warning for arrays that exceed this limit. The *limit* argument is a required, unsigned decimal number that specifies the maximum array size to probe. The *limit* for packed arrays is measured in bits. A *limit* of 0 implies the Xcelium Simulator maximum, which is 2^{32} for packed arrays.

The Tcl variable `probe_packed_limit` defines the *limit* for packed arrays at the start of each simulation run to 4×1024 . You can modify this variable using the Tcl `set` command or by using the `-packed` option with `probe -create` to override the default *limit* when probing a packed array.

`-ports`

Specifies that all ports within a scope are to be included in the probe. This applies to:

- The current debug scope (if no scope(s) or object(s) is named in an argument)
- The scope(s) named in the argument
- Subscopes that you specify with the `-depth` option

For EVCD, you can use `-ports` or `-all` to include all of the declared ports within a scope in the probe.

`-power`

Probes all low-power simulation control expressions to the SHM database.

Only SHM probes are supported. VCD, EVCD, and screen probes (using `-screen`) are not supported. Including the `-shm` option is not required.

You can probe either IEEE 1801 or CPF control expressions. For IEEE 1801, you can also use `probe -power` to trace the table states for a power state table (PST). In this scenario, each PST is probed to the SHM database as a structure with each field being a state of the PST. Each state is either `Active` or `Off`.

`-pwr_mode`

Probes power mode information for all power domains.

Only SHM probes are supported. VCD, EVCD, and screen probes (using `-screen`) are not supported. Including the `-shm` option is not required.

The only other `probe` command option that can be used with `-pwr_mode` is the `-name` option. You cannot specify other objects with the `probe -pwr_mode` command.


The following information is saved in the SHM database for each power domain:

- The power domain name
- The name of the power mode the domain is currently in
- The state of the power domain (ON, OFF, TRANSITIONING, STANDBY, UNINITIALIZED)
- The nominal condition name and current voltage of the power domain

`-qda_max_elem <num|all|0>`

Limits the probed elements in QDA.

The number of QDA elements that get probed by default is limited to a fixed size of elements, thereby improving the SHM probe performance and the waveform performance. The default limit is set to 1000 elements. When QDA exceeds this limit, only the first 1000 elements are probed.

 The index order of AA that is used to determine the first 1000 is the simulator serial order.

The QDA probe limit configuration is explained below:

`probe -shm -create [-qda_max_elem <num|all|0>]`

- **num** : A digit indicating the number of elements to probe for QDAs
- **all** : All QDA elements are to be probed.
- **0** : No elements are to be probed. Only the size of the QDA is probed.

When specifying the `probe` command more than once for the same item with different `-qda_max_elem` configurations, the max number that is used for `-qda_max_elem` is used for the actual probe.

`-screen [-format <format_string>] [-redirect <file>] [-append] <objects>`

Monitors the value changes on the specified object(s) and displays the values on the screen.

The argument to this option must be one or more *objects*. This can include Verilog memories, memory ranges, and single memory elements. You cannot specify a scope as an argument to this option. Additionally, you cannot use the `-screen` option with any other probe command-line options. You can use the following modifiers:

- Include `-format` to specify the output format.
The `format_string` argument can contain both text and format specifiers. Variables and objects are

paired sequentially with specifiers. Valid formats are:

- **%b**: Binary format. The argument must be an object name that is either a scalar object or a logic vector.
- **%d**: Decimal format. The argument must be an object name whose type is integer, physical, or enumeration.
- **%f**: Real number in floating-point notation. The argument must be an object whose base type is real.
- **%o**: Unsigned octal object. The argument must be a scalar object or a logic vector.
- **%s**: Substitute. The argument is substituted on an as-is basis.
- **%x**: Unsigned hex object. The argument is an object name that is either a scalar object or a logic vector.
- **%v**: Default value format. The argument must be a signal or a variable name. The value of the object is formatted appropriately, according to its type.
- **\t**: Inserts a horizontal tab.
- **\n**: Inserts a carriage return.
- **\c**: Used as the last character, this suppresses the default line feed.
- **%c**: Prints the cycle count.
- **%D**: Prints the current delta cycle count.
- **%T**: Prints the current simulation time.
- Include `-redirect` to redirect the output to a file.
- Include `-append` to append the results of another `probe` `-screen` command to the file specified with `-redirect`.

`-scope_list <filename>`

Reads in a list of design scopes along with the corresponding depth information.

Each line of the specified *filename* holds one scope and one depth indicator in the following ASCII format:

<depth> <design_scope>

Where *depth* can be:

{an_unsigned_number | 'all'}

`-sr_save`

Probes the saved value of state retention variables.

In a low-power simulation, you can define the rules for replacing registers in switchable power domains with state retention registers so that the values of the registers are saved before power-down and restored after power-up. The command you use is dependent on whether or not you are following the IEEE 1801 or CPF flow.

- For IEEE 1801, use the `set_retention` command
- For CPF, use the `create_state_retention_rule` command

Use the `-sr_save` option to probe both the value of state retention variables and the saved value of the variables. Only SHM probes are supported. VCD, EVCD, and screen probes (using `-screen`) are not supported. Including the `-shm` option is not required. You can specify either a scope or variable name.

```
probe -create scope_name [ -shm] -sr_save
```

Or:

```
probe -create variable_name [ -shm] -sr_save
```

If you specify a *scope_name*, you can use the `-all` and `-memories` options (for Verilog designs) or the `-all` and `-variables` options (for VHDL designs). If you specify a *variable_name*, that variable name is saved as *variable_name_sr*. Using the `-depth` option together with `-sr_save` is not supported.

`-sr_all_save`

Probes the values of all state retention shadow registers, regardless of scope.

In a low-power simulation, each state retention register contains a shadow register that can preserve the register's state during power-down and restore that state at power-up. Use the `-sr_all_save` option to probe the values of all shadow registers in your low-power design, regardless of scope, and save these values to the SHM database.

Only SHM probes are supported. VCD, EVCD, and screen probes (using `-screen`) are not supported. Including the `-shm` option is not required.

You can specify this option by itself or together with other SHM probe options:

```
probe [ -shm] -sr_all_save
```

Or:

```
probe -power [ -shm] -sr_all_save
```

If the `probe` command does not explicitly specify an existing SHM database, the Xcelium Simulator uses the default SHM database.

`-string_max_len <size>`

Adjusts the limit of dynamic string values by specifying a *size* in bytes.

This option can take a maximum string limit of up to 1MB. The *size* value must be specified as a positive integer.


By default, Xcelium limits probes of dynamic string values to 1K in size. To set a custom limit:

- Use the Tcl `probe` command option `-string_max_len`.
- Set the environment variable `SHM_STRING_MAX_LEN`.

After implementing one of the supported methods above, if a declared object has a length greater than the set string limit, then Xcelium displays a warning, truncates the string to the set limit, and probes the string. If the limit is set to a value greater than 1MB (1048576 bytes), then Xcelium outputs a warning and automatically set the maximum limit to 1MB.

`-tasks`

Includes objects declared within Verilog task scopes. By default, these objects are not included in the probe.

 Including this option is not necessary if a Verilog task scope is specified as the argument to the `probe` command (either explicitly or implicitly as the current debug scope).

`-transaction`

Records the value change trace as a transaction.

Transaction probes are created for only those objects whose value change can be recorded as a transaction. If you specify one or more scopes, a transaction probe is created for the objects that support transactions, and other objects in the scope(s) are ignored. If you specify an object, a transaction probe is created if the object supports transactions. If the object does not support transactions, the object is ignored with a warning.

You can use this option with `-assertions` to probe assertion objects as transactions. If the `-signals` option is also included, the signals contributing to the assertions are also included as probed signals. Other objects that can be recorded as transactions are not included in the probe. If there are multiple `probe -assertions -transaction` commands, all of the transaction records for assertions are written to the database used in the first `probe` command, even if the subsequent `probe` commands explicitly specify another database with the `-database` option.

In contrast, SystemC transaction probes can write transaction records to multiple databases. Each `probe -transaction` command uses the database specified for the `probe` command.

`-unpacked <limit>`

Specifies the limit for unpacked arrays.

The `-unpacked` option applies only to SystemVerilog static arrays. Dynamic arrays, queues, and queues of dynamic arrays (QDAs) are not supported.

This option sets a size limit to unpacked arrays and enables the display of a warning for arrays that exceed this limit. The `limit` argument is a required, unsigned decimal number that specifies the maximum array size to probe. The `limit` for unpacked arrays is measured in terms of the actual array elements. A `limit` of 0 implies the Xcelium Simulator maximum, which is 2^{22} for unpacked arrays.

The Tcl variable `probe_unpacked_limit` defines the `limit` for unpacked arrays at the start of each simulation run to 16×1024 . You can modify this variable using the Tcl `set` command, or by using the `-unpacked` option with `probe -create` to override the default `limit` when probing unpacked arrays.

`-uvm`

Includes UVM (or OVM) base class objects with class object probes on SHM databases.

By default, the Tcl `probe -create` command filters out probes to design elements that do not provide debug value in order to limit the size of SHM probes. This option is required in order to probe UVM/OVM base classes.

`-vspice_cell`

Includes voltage, current, or both on the ports of the spice instance instantiated at Verilog/SPICE or VHDL/SPICE boundary, without saving the internal nets and the sub hierarchies.

For example, the following command probes voltage on the SPICE ports on Verilog/SPICE boundary.

```
xcelium> probe -database ams_database -vspice_cell
```

You can use the `-flow` and `-all` options with the `-vspice_cell` option, to probe both voltage and current on the SPICE ports on Verilog/SPICE boundary. For example:

```
xcelium> probe -database ams_database -vspice_cell  
-flow
```

`-waveform`

Causes the objects in the probe to be added to the SimVision waveform display if the simulator is running in GUI mode. If the waveform viewer is not running, it is invoked and the objects are then added to the display.

This option has no effect if the simulator is not in GUI mode.

`-delete <probe_name>`
`[<probe_name>...]`

Deletes the probe(s) specified by the argument. The argument can be a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

`-disable <probe_name>`
`[<probe_name>...]`

Disables the probe(s) specified by the argument. The argument can be a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

`-enable <probe_name>`
`[<probe_name>...]`

Resumes the probe(s) specified by the argument. The argument can be a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

`-save <filename>`

Creates a Tcl script that you can execute to re-create the current databases and probes. If you do not specify a *filename* argument, the script is printed to the screen.

```
-show [<probe_name>...]  
[-database <dbase_name>]
```

Outputs information about the probe(s) specified in the argument. The argument can be a probe name or a list of probe names. You can use the two wildcard characters (* and ?) in the probe name argument.

Add the `-database` sub-option to print information on probe(s) in the specified database.

probe Command Examples

Create Probes

Consider a `database` command that opens a default SHM database called `waves`.

```
xcelium> database -open -shm waves -default  
Created default SHM database waves
```

The following command creates a probe on all objects in the current debug scope. Data is sent to the default SHM database. In this case, the `-create` modifier is not required; however, `-all` is required because no `object` or `scope_name` is specified. Additionally, since only one database (a default SHM database) is open, `-shm` is optional.

```
xcelium> probe -create -shm -all  
Created probe 1
```

The following command creates a probe on all objects in the current debug scope. Here, no default SHM database exists, so the simulator creates a default database called `xcelium.shm`:

```
xcelium> probe -create -shm -all  
Created default SHM database xcelium.shm  
Created probe 1
```

The following command creates a probe on all objects in scope `top.u1` and sends data to the default SHM database (creating one called `xcelium.shm` , if a default database does not exist):

```
xcelium> probe -shm top.u1
```

The following command creates probes on all objects in the current scope that have names beginning with `rst` and ending with `p5`:

```
xcelium> probe -shm rst*p5
```

The following command creates probes on all objects in the scope `top` that have two letters and that have names that start with the letter `c`:

```
xcelium> probe -shm top.c?
```

The following command creates a probe on all objects in scopes `top.u1` and `top.u2`:

```
xcelium> probe -shm top.u1 top.u2
```

The following command creates a probe on the signal `sum` in the current debug scope and sends data to the default SHM database:

```
xcelium> probe -shm sum
```

The following command creates a probe on `sum` and `c_out` in the current debug scope and sends data to the default SHM database:

```
xcelium> probe -shm sum c_out
```

The following command creates a probe on `sum` in scope `u1`, sending data to the default SHM database:

```
xcelium> probe -shm u1.sum
```

The following command creates a probe on all ports in scope `u1`:

```
xcelium> probe -shm u1 -ports
```

The following command creates a probe on all ports in scope `u1` and its subscopes:

```
xcelium> probe -shm u1 -ports -depth 2
```

The following command creates a probe on all ports in scope `u1` and all scopes below `u1`:

```
xcelium> probe -shm u1 -ports -depth all
```

The following command creates a probe on all ports in scope `u1` and all scopes below `u1`, stopping at modules with a ``celldefine` directive:

```
xcelium> probe -shm u1 -ports -depth to_cells
```

By default, Verilog memories are not included when you probe objects in a scope. To probe Verilog memories in a scope, use `-all -memories`, as shown in the following example:

```
xcelium> probe -shm -all -memories u1
```

The following command creates a probe on the Verilog memory called `mem` in the current scope:

```
xcelium> probe -shm mem
```

You can also probe individual memory elements or memory ranges, as shown in the following two commands:

```
xcelium> probe -shm mem[255]
```

```
xcelium> probe -shm mem[255:128]
```

By default, VHDL variables are not included when you probe objects in a scope. To probe VHDL variables in a scope, use `-all -variables`, as shown in the following example:

```
xcelium> probe -shm -all -variables :u1
```

The following command creates a probe called `peek`:

```
xcelium> probe -shm sum -name peek
```

The following example shows a Verilog module that contains a generate-loop. Each for-generate instance is a scope. The name of the scope is the name of the generate block (`bit`, in this example), plus an instance-select, which is the value of the `generate` parameter. In this example, the scopes are `bit[0]`, `bit[1]`, `bit[2]`, and `bit[3]`. See [Generate Constructs](#) for details on generated instantiations.

```
module addergen1 (co, sum, a, b, ci);
  parameter SIZE = 4;
  output [SIZE-1:0] sum;
  output co;
  input [SIZE-1:0] a, b;
  input ci;
  wire [SIZE:0] c;
  genvar i;
  assign c[0] = ci;
  generate
    for(i = 0; i < SIZE; i = i + 1)
      begin : gen_blk
        wire t1, t2, t3; //internal nets
        xor g1 (t1, a[i], b[i]);
        xor g2 (sum[i], t1, c[i]);
        and g3 (t2, a[i], b[i]);
        and g4 (t3, t1, c[i]);
        or g5 (c[i+1], t2, t3);
      end
  endgenerate
  assign co = c[SIZE];
endmodule
```

The name of a for-generate, without an index, is treated as a reference to each of the generate scopes. The following two commands are equivalent.

```
xcelium> probe -shm gen_blk
xcelium> probe -shm gen_blk[0] gen_blk[1] gen_blk[2] gen_blk[3]
```

You can probe SystemVerilog dynamic objects to an SHM database. A dynamic object can be probed before it comes into existence (that is, before the "new" that creates the dynamic object occurs). You can probe a class as a whole, with all its objects being captured, or you can probe a member (property) of a class instance. For example:


```
class c1;
  struct {
    int i;
  } s1;
endclass
module top;
  c1 o1;
  initial
    #1 o1 = new;
endmodule
```

```
% xrun -access +rwc -tcl test.sv
```

```
...
```

```
...
```

```
xcelium> probe -shm o1           # Probe the class instance
```

```
Created default SHM database xcelium.shm
```

```
Created probe 1
```

```
xcelium> probe -shm o1.s1.i      # Probe a member of the class instance
```

```
Created probe 2
```

```
xcelium> probe -show
```

```
1      Enabled      top.o1 (database: xcelium.shm) -shm
```

```
Number of objects probed : 1
```

```
2      Enabled      top.o1.s1.i (database: xcelium.shm) -shm
```

```
Number of objects probed : 1
```

```
xcelium> run
```

```
xmsim: *W,RNQUIE: Simulation is complete.
```

```
xcelium> probe -show
```

```
1      Enabled      top.o1 (database: xcelium.shm) -shm
```

```
Number of objects probed : 1
```

```
2      Enabled      top.o1.s1.i (database: xcelium.shm) -shm
```

```
Number of objects probed : 2
```

In the following example, the `database` command opens a default VCD database called `test_vcd`. The `probe` command creates a probe on all ports in the scope `counter`. Data is sent to the default VCD database.

```
xcelium> database -open -vcd test_vcd -default
```

```
Created default VCD database test_vcd
```

```
xcelium> probe -create -vcd top.counter -ports
```

```
Created probe 1
```

The following `probe -vcd` command creates a probe on all objects in the current debug scope. In this example, no default VCD database exists, so the simulator creates a default database called `xcelium.vcd`.

```
xcelium> probe -create -vcd -all
```

```
Created default VCD database xcelium.vcd
```

Created probe 1

When generating an EVCD database for Verilog, you must specify a scope(s) as the argument to the `probe` command. You cannot specify specific ports. In the following example, the `database` command opens a default database called `test_evcd`. The associated filename is `test_evcd.evcd`. The `probe` command creates a probe on all primary ports in the scope `test_bench.dut`.

```
xcelium> database -open test_evcd -evcd -default
Created default EVCD database test_evcd
xcelium> probe -create test_bench.dut -evcd
```

In the following example, the `database` command opens an EVCD database called `test_evcd`. The `-into` option specifies that the filename of the output file is `test.evcd`. The `probe` command creates a probe on all primary ports in the scope `test_bench.dut`. Data is sent to the EVCD database called `test_evcd`.

```
xcelium> database -open test_evcd -evcd -into test.evcd
xcelium> probe -create test_bench.dut -evcd -database test_evcd
```

In the following example, the `probe` command includes the `-evcd splitio` option. This option dumps the value of the input port if any driver into the scope `test_bench.dut` changes value, and dumps the value of the output port if any driver within the scope changes value. If both inside and outside drivers change, two messages are dumped: one corresponding to the input, and one corresponding to the output.

```
xcelium> database -open -evcd test_evcd -into test.evcd
xcelium> probe -create test_bench.dut -evcd splitio -database test_evcd
```

The `probe` command in the following example includes the `-evcd -evcdfORMAT` option. The argument to the `-evcdfORMAT` option is 1, which specifies that both the zero and one component of the value is to be dumped (for example, `pD 6 5 <0`).

```
xcelium> database -open -evcd test_evcd -into test.evcd
xcelium> probe -create test_bench.dut -evcd -evcdfORMAT 1 -database test_evcd
```

In the following example, the `database` command opens a default EVCD database called `test_evcd`. The `probe` command creates a probe on all primary ports in the VHDL scope `:u1`. Data is sent to the default EVCD database.

```
xcelium> database -open -evcd test_evcd -default
Created default EVCD database test_evcd
xcelium> probe -create -evcd -all :u1
Created probe 1
```

In the following example, the `probe` command includes the `-evcd -mode lfcompat` option. This option applies only to VHDL, and is provided for backward compatibility. It specifies that vector ports are to be dumped as individual elements, and that the strength mapping is that used in legacy releases.

```
xcelium> database -open test_evcd -evcd -into test.evcd -timescale ns
xcelium> probe -create :i1:i1:d -evcd -mode lfcompat -database test_evcd
```

The following `probe` command also includes the `-evcd -mode lfcompat` option. This option is required if

you want to dump a subelement of a compressed VHDL signal to an EVCD database.

```
xcelium> probe -create :top:cans(0) -evcd -mode lfcompat -database test_evcd
```

Set Packed Array Limits

The following command limits the number of elements probed in the packed array named `my_array` to 200 bits.

```
xcelium> probe -create my_array -packed 200
```

If an array is encountered that exceeds the size specified by `probe_packed_limit`, a warning is issued. This informs you of the *location*, *size*, and *limit* of the array, and shows you the correct format of the `probe` command that you should use to access the array. The warning has the following format:

```
xmsim: *W,PRPASZ: Packed array at < pathname > of < size > bits exceeds limit of < limit > -  
not probed
```

```
xmsim: *W,PRPASZ: Use 'probe -create -packed < size > < pathname >'
```

where:

- **pathname** : is the full hierarchical name of the array. For example: the pathname for `my_array` in instance `b`, within instance `a` would be: `a.b.my_array`.
- **size** : is the specified size of the array in bits.
- **limit** : is the value set using the `-packed` option.

If you specify the `-unpacked` option to limit the number of elements probed in an unpacked array, and if the encountered array exceeds the size specified by `probe_unpacked_limit`, the PRUASZ warning is printed to the display. This warning is similar to PRPASZ but is specific to unpacked arrays.

Set the Maximum String Length

The following command creates a probe on all objects in scope `top.u1` and sets a maximum string length limit of 512. The data is sent to the default SHM database.

```
xcelium> probe -shm top.u1 -string_max_len 512
```

This command creates a probe on all objects in scope `top.u2` and the hierarchy below, and sets a maximum string length limit of 1.

```
xcelium> probe -shm -depth all top.u2 -string_max_len 1
```

In this scenario, the string length limit results in the following STRLENMSG warning:

```
xmsim: *W,STRLENMSG: String of length 11 exceeds limit. Dumped value is truncated to 1
```

character

This command specifies a number value of 9999999 for the option `-string_max_len`, which is greater than 1MB.

```
xcelium> probe -create -shm top.ul -string_max_len 9999999
```

This results in an STRMAXPRER warning.

```
xmsim: *W,STRMAXPRER: String length limit should be less than "1048576" for probe -
string_max_len, got "9999999"
```

Exclude Objects from a Probe

The following command probes all objects in the current debug scope, except `clk`:

```
xcelium> probe -shm * -exclude clk
```

The following command probes all objects in the current debug scope, except `clk` and `rst`:

```
xcelium> probe -shm * -exclude clk -exclude rst
```

or:

```
xcelium> probe -shm * -exclude {clk rst}
```

The following command probes all objects in the current debug scope, except objects whose name starts with `d`:

```
xcelium> probe -shm * -exclude d*
```

The following command probes all objects, except objects inside scope `mux_m1` and its subscopes:

```
xcelium> probe -vcd muxdff -depth all -exclude mux_m1
```

The `-exclude` option supports wildcards, but the wildcards are applicable only to object names, not scope names. For example, the following command probes all objects in the design except those objects inside scope `mux_m1` whose name starts with `d`.

```
xcelium> probe -vcd muxdff -depth all -exclude mux_m1:d*
```

The following command probes all objects in scope `muxdff` and its subscopes, except for:

- Objects in instance `dff_g1`
- Objects in scope `mux_m1` whose name starts with `d`

```
xcelium> probe -vcd muxdff -depth all -exclude dff_g1 -exclude mux_m1:d*
```

```
Created default VCD database xcelium.vcd
```

```
Created probe 1
```

```
xcelium> probe -show
```

```
1      Enabled      muxdff (database: xcelium.vcd) -vcd -depth all -exclude
                        {mux_m1:d*} -exclude dff_g1
```

```
Number of objects probed : 10
```

Output Verbose Probe Information

The following command displays the state of all probes:

```
xcelium> probe -show
```

The following command displays the state of the probe called `peek`:

```
xcelium> probe -show peek
```

Deactivate Probes

The following command deactivates the probe called `peek`:

```
xcelium> probe -disable peek
```

Enable Probes

The following command enables the probe called `peek`, which was disabled in the previous command:

```
xcelium> probe -enable peek
```

Monitor Signal Values and Display Changes on the Screen

The following `probe -evcd` command creates a probe on all primary ports in the current debug scope. In this example, no default EVCD database exists, so the simulator creates a default database called `xcelium.evcd`.

```
xcelium> probe -create -evcd -all
Created default EVCD database xcelium.evcd
Created probe 1
```

The following command monitors value changes on signals `clock` and `count`. When either of these signals changes value, the simulator displays output on the screen.

```
xcelium> probe -screen clock count
Created probe 1
xcelium> run 10 ns
Time: 5 NS: board.clock = 1'h1 : board.count = 4'hx
Ran until 10 NS + 0
```

In the following command, the `-format` option is included to format the output of `probe -screen`:

```
xcelium> probe -screen -format "clock = %d \ncount = %b" clock count
Created probe 1
xcelium> run 10 ns
```

```
clock = 1'd1
count = 4'bxxxx
Ran until 10 NS + 0
```

The following example illustrates the simulator output when you use `probe -screen` to monitor signal value changes and then deactivate the probe at some later time:

```
xcelium> probe -screen clock count
Created probe 1
xcelium> run 10 ns
Time: 5 NS: board.clock = 1'h1: board.count = 4'hx
Ran until 10 NS + 0
xcelium> probe -disable 1
xcelium> run 10 ns
Time: 10 NS: board.clock = <disabled> : board.count = <disabled>
Ran until 20 NS + 0
```

Redirect Value Change Information to a File

The following command redirects the output of the signal `sum` to a file named `myprobe.txt`:

```
xcelium> probe -screen -redirect myprobe.txt sum
```

The following command probes a signal `board.count` and redirects the results to a file named `myprobe.txt`. Then, another command probes the signal `board.clock` and appends these results to the same file, `myprobe.txt`.

```
xcelium> probe -screen -redirect myprobe.txt board.count
Created probe 1
xcelium> run 500 ns
Ran until 500 NS + 0
xcelium> probe -screen -redirect myprobe.txt -append board.clock
Created probe 2
xcelium> run 500 ns
```

- If a `probe -screen` command results in multiple signals, a discrete line of output is generated for each signal by default.

```
Time: 10 NS: top.u1.a4 = 4'hf
Time: 10 NS: top.u1.a32 = 32'hffffffff
```

- If you want to generate multiple signal events on the same line, set the value of the `probe_screen_format` variable to 1:

```
Time: 10 NS: top.u1.a4 = 4'hf; top.u1.a32 = 32'hffffffff;
```

Probe Low-Power Control Expressions

The following probe command includes the `-power` option to probe all low-power simulation control expressions to the SHM database. Only SHM probes are supported. VCD, EVCD, and screen probes (using `-screen`) are not supported. Including the `-shm` option is not required.

```
xcelium> database -open -shm -into waves.shm waves -default
Created default SHM database waves
xcelium> probe -power -database waves
Created probe 1
xcelium> probe -show 1
1      Enabled      TESTBENCH.inst.pcu_inst.pau[2] (database: waves) -shm
                        TESTBENCH.inst.pcu_inst.plu[2]
                        TESTBENCH.inst.pcu_inst.palu[2]
                        TESTBENCH.inst.pcu_inst.prf[2]
                        TESTBENCH.inst.pcu_inst.prf[1]
                        TESTBENCH.inst.pcu_inst.pau[1]
                        TESTBENCH.inst.pcu_inst.pau[0]
                        TESTBENCH.inst.pcu_inst.plu[0]
                        TESTBENCH.inst.pcu_inst.palu[0]
                        TESTBENCH.inst.pcu_inst.prf[0]
                        Number of objects probed : 10
xcelium>
```

In the following example, the `probe -pwr_mode` command probes power mode information for the three power domains in the design:

```
xcelium> database -open -shm -into waves.shm waves -default
Created default SHM database waves
xcelium> probe -create -pwr_mode -database waves
Created probe 1
xcelium> probe -show
1      Enabled      LPS (database: waves) -shm
                        Number of objects probed : 3
xcelium>
```

UVM Probes

```
# Create a default filtered SHM database
xcelium> database -shm -open filtered -default

# Create probe 1
xcelium> probe -shm -memories -all -depth all -packed 32768 uvm_pkg::uvm_top top *

# After running the simulation, generate the following report:
xcelium> probe -show -verbose
1 Enabled uvm_pkg::uvm_top (database: filtered) -shm -all -memories -depth all
```

```
top
top.go
Number of objects probed : 3
probe_packed_limit: 32768    probe_unpacked_limit: 16384
Individual probes count : 17
...
```

Alternatively, you can run the same simulation using the `-uvm` option:

```
# Create a default unfiltered SHM database
xcelium> database -shm -open unfiltered -default

# Create probe 2
xcelium> probe -shm -memories -all -depth all -packed 32768 uvm_pkg::uvm_top top * -uvm

# Generate a new report:
xcelium> probe -show -verbose
2 Enabled uvm_pkg::uvm_top (database: unfiltered) -shm -all -memories
    -depth all -uvm
    top
    top.go
    Number of objects probed : 3
probe_packed_limit: 32768    probe_unpacked_limit: 16384
Individual probes count : 3611
...
```

This results in a difference of 3594 unique probes.

Probe with Wildcards

The following sequence of commands illustrates the use of wildcard characters in probe name arguments. Two probes called `peek1` and `peek2` are created. Both probes are then deactivated using the `*` wildcard character. Both probes are then deleted using the `?` wildcard character.

```
xcelium> probe -database waves clock -name peek1
Created probe peek1
xcelium> probe -database waves count -name peek2
Created probe peek2
xcelium> probe -show
peek1 Enabled board.clock (database: waves) -shm
      Number of objects probed : 1

peek2 Enabled board.count (database: waves) -shm
      Number of objects probed : 1
xcelium> probe -disable pe*
xcelium> probe -show
peek1 Disabled board.clock (database: waves) -shm
      Number of objects probed : 1
```



```
peek2    Disabled          board.count (database: waves) -shm
          Number of objects probed : 1
xcelium> probe -delete peek?
xcelium> probe -show
No probes set
xcelium>
```

Probe with the Same Module/Signal Name

The following command creates a probe on all signals in the scope `top`. Data is sent to the database `waves2`. This database must already exist.

```
xcelium> probe -database waves2 top
```

Now consider the following code, where the module name is `interrupr`, and the module contains a signal, also called `interrupr`.

```
module interrupr (clk, rst_n, error, clear_error, mask, interrupr);
    input clk;
    input rst_n;
    input[5:0] error;
    input[5:0] clear_error;
    input[5:0] mask;
    output interrupr;

    wire interrupr;
    ...
    ...
endmodule
```

In this case, the following command probes the signal `interrupr`, rather than all signals of the scope.

```
xcelium> probe interrupr -all -depth all -shm
Created probe 1
xcelium> probe -show 1
1    Enabled    interrupr.interrupr (database: xcelium.shm) -shm -all -depth all
          Number of objects probed : 1
```

If you want to probe the scope `interrupr`, you can add `$root` to the path. This `$root` allows explicit access to the top of the instantiation tree and provides a mechanism to disambiguate a local path (which takes precedence) from the rooted path with the same name. For example:

```
xcelium> probe \ $root.interrupr -all -depth all -shm
Created probe 2
xcelium> probe -show 2
2    Enabled    interrupr (database: xcelium.shm) -shm -all -depth all
          Number of objects probed : 8
```

If you are probing statement trace information into a database opened with a `database -statement` command, the argument must be a scope. All statements within the scope are traced. You can probe specific objects, inputs, outputs, or ports to a database opened with `-statement`. However, statements are not traced if you specify an object or use the `-inputs`, `-outputs`, or `-ports` options.

Probe with No Connectivity Access

The following command shows the error message that is displayed if you run in the default "regression" mode (no read, write, or connectivity access to simulation objects) and then probe an object that does not have read access:

```
xcelium> probe -shm r
xmsim: *E,OBJACC: Object must have read access: top.r.

xcelium> probe -shm w
xmsim: *E,OBJACC: Object must have read access: top.w.
```

Related Topics

- [probe_screen_format](#)
- [Probing Different Simulation Objects](#)
- [Access to Simulation Objects](#)
- [Wildcards Characters in Tcl Commands](#)

Setting a Probe

You can probe objects to the following kinds of databases:

- SHM (for Verilog, VHDL, or mixed-language)
- VCD (for Verilog, VHDL, or mixed-language)
- EVCD (for Verilog or VHDL)

Use the Tcl `probe` command with the optional `-create` modifier to create probes.

The basic syntax of the `probe` command is as follows:

```
probe [-create] [{object | scope_name} ...]
      {-shm | -vcd | -evcd | -database dbase_name}
```

The `-create` modifier can be followed by an argument that specifies:

- The object(s) to be traced

- The scope(s) to be traced
- A combination of object(s) and scope(s) to be traced

If you do not specify an argument, the current debug scope is assumed, but you must include an option that specifies the objects you want to include in the trace (`-all`, `-inputs`, `-outputs`, or `-ports`).

You must include an option to specify the database into which values are dumped. Use one of the following options:

- `-database dbase_name`

Send the probe to the specified database. The database must already exist.

- `-shm`

Send the probe to the default SHM database. If no default database is open, *xcelium* opens a default database called `xcelium.shm`.

- `-vcd`

Send the probe to the default VCD database. If no default database is open, *xcelium* opens a default database called `xcelium.vcd`.

- `-evcd`

Send the probe to the default EVCD database. If no default database is open, *xcelium* opens a default database called `xcelium.evcd`.

Only objects that have read access are probed.

Related Topic

- [Access to Simulation Objects](#)

Displaying Probe Information

Use the `-show` option with the `probe` command to display information about a probe.

The argument to `-show` is a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

- The asterisk (`*`) matches any number of characters.
- The question mark (`?`) matches any one character.

If you do not include an argument, the simulator displays information on all probes.

For VCD and EVCD, the number of objects displayed is the number of objects that appear in the header of

the generated VCD or EVCD file.

For SHM, the probe count is the number of objects displayed when the SHM database is opened with the SimVision waveform viewer. This also applies to probes opened with `probe -screen`.

Syntax

```
probe -show [<probe_name>...] [-database <dbase_name>]
```

Options

```
-show [<probe_name>...]  
[-database <dbase_name>]
```

Outputs information about the probe(s) specified in the argument. The argument can be a probe name or a list of probe names. You can use the two wildcard characters (*) and ?) in the probe name argument.

Add the `-database` sub-option to print information on probe(s) in the specified database.

Examples

The following command displays the state of all probes. The number of objects probed is included.

```
xcelium> probe -show  
1      Enabled      board (database: waves) -shm -depth all  
                        Number of objects probed : 21  
2      Enabled      board (database: xcelium.vcd) -vcd  
                        Number of objects probed : 4
```

The following command displays the state of the probe named `peek`.

```
xcelium> probe -disable peek
```

Consider the `probe` commands below that set up separate counts for the same `waves` database.

```
xcelium> probe count[1:0] -database waves  
xcelium> probe count[2] count[3] -database waves
```

The following `probe -show` command displays this separate count, even though the output goes to the same database.

```
xcelium> probe -show  
1      Enabled      board.count[1:0] (database: waves) -shm  
                        Number of objects probed : 1  
2      Enabled      board.count[2] (database: waves) -shm  
                        board.count[3]  
                        Number of objects probed : 2
```

Deactivating a Probe

Use the `-disable` option with the `probe` command to deactivate a probe.

While the probe is deactivated, values for the objects in that probe are not written to the database. To resume probing, use the `-enable` modifier.

The argument to `-disable` is a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

- The asterisk (`*`) matches any number of characters
- The question mark (`?`) matches any one character

You can deactivate SHM probes individually at any time.

You cannot deactivate VCD or EVCD probes individually. Use `database -disable` to deactivate all VCD or EVCD probes. See the `database` command.

You cannot deactivate probes that record statement trace data into a database opened with a `database -statement` command.

Syntax

```
probe -disable <probe_name> [<probe_name>...]
```

Options

```
-disable <probe_name>  
[<probe_name>...]
```

Disables the probe(s) specified by the argument. The argument can be a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

Example

The following command deactivates the probe named `peek`.

```
xcelium> probe -disable peek
```

Probing Different Simulation Objects

Xcelium supports probing [Verilog](#), [VHDL](#), and [SystemC](#) objects that have read access.

Probing Verilog Objects

When probing Verilog objects, you can create an SHM, VCD, or EVCD database by opening a database with the Tcl `database` command and then probing signals or scopes with the Tcl `probe` command. You can also:

- Create an SHM database by using the `$shm_open` and `$shm_probe` system tasks in your Verilog source code. See [Managing the SHM Database](#) for details.

For backward compatibility with Verilog code that contains calls to Signalscan tasks for recording data (`$recordvars`, `$recordfile`, `$recordsetup`, and so on), Cadence has implemented these tasks as system tasks native to the simulator. See [Using \\$recordvars and Related Tasks](#) for details.

- Create a VCD database by using value change dump system tasks (`$dumpfile`, `$dumpvars`, `$dumpall`, and so on) in your Verilog source code. See [The Value Change Dump \(VCD\) File](#) for more information.
- Create an EVCD database by using the `$dumpports` system task. See [The Extended Value Change Dump \(EVCD\) File](#) for details.

You can only specify scopes when probing Verilog objects to an EVCD database. You cannot probe specific signals.

For Verilog, you cannot probe arrays of variable data types to a VCD database. This includes Verilog memories, which are one-dimensional arrays of type `reg`. You cannot probe variables declared as multi-dimensional arrays.

Probing VHDL Objects

When probing VHDL objects, you cannot probe VHDL signals, ports, variables, and generics that are declared inside a subprogram. If VHDL objects are *not* declared inside subprograms, Xcelium allows to probe such objects to an SHM database as long as their type does not fall into one of the following categories:

- Non-standard integer types whose bounds require more than 32 bits to represent
- Access and file types
- Any composite type that contains one of the above types

For VHDL, the VCD file uses syntax and format conventions identical to those used for Verilog. You can dump all signals, ports, and variables, including those declared as multi-dimensional arrays, to a VCD database, with the following limitations:

- The signal, port, or variable must be of type `std_ulogic`, `bit`, `integer`, `real`, or any user-defined type that is a subset of `std_ulogic`.

- Objects that are declared inside a subprogram cannot be probed.
- Signals and variables that correspond to records are not dumped into the VCD file.
- The type of the object cannot be:
 - A non-standard integer type whose bounds requires more than 32 bits to represent
 - Access and file types
 - Any composite type that contains one of the above types

Signals and ports in VHDL map to wires in Verilog, and variables map to registers.

The values that can be dumped to a VCD file are 0, 1, z, and x. Other values are mapped as follows:

- U -> X
- W -> X
- L -> 0
- H -> 1
- - -> X

You can also create a VCD database and probe VHDL objects to the database by using the Tcl `call` command. This command calls predefined CFC routines, which are part of the simulator C interface and is a feature that has been retained for backward compatibility. The recommended method of generating a VCD file is to open a database with the `database -open -vcd` command and probe objects to the database with the `probe -vcd` command. See [Generating a VCD File Using CFC Routines](#) for details on this alternate method of generating a VCD file.

For an EVCD database, you can probe only the primary ports of component instances. The simulator monitors the ports for both value and drive level, and generates an output file that contains the value, direction, and strength of the primary ports of the component instances. See [The Extended Value Change Dump \(EVCD\) File](#) for more information on EVCD databases.

Probing SystemC Objects

You can apply the Tcl `probe` command to SystemC processes and the following SystemC objects:

`sc_signal`, `sc_clock`, `sc_in`, `sc_out`, `sc_inout`, `ncsc_viewable`, `sc_fifo`, `sc_fifo_in`, `sc_fifo_out`, `sc_mutex`, `sc_semaphore`, `tlm_fifo`, `tlm_req_rsp_channel`. For the templated objects in this list, this command is supported when the object is templated by a C++ data type, by a SystemC built-in data type, or by a user-defined data type that implements the `<<` operator.

For `sc_port` and `sc_export` objects, `probe` is supported if the `sc_object` bound to the port or export supports `probe`.

The `-vcd` and `-evcd` options are not supported for SystemC objects.

You can also:

- Probe SystemC PSL assertions and the content of a SystemC memory module derived from the `ncsc_memory_debug_if` class.
- Apply `probe -shm` and `probe -screen` to an arbitrary SystemC object, if that object declares support for a value and value change callback (the member methods `ncsc_supports_value()` and `ncsc_supports_value_change()`).
- Record object changes in delta times when the SHM database is an event database (that is, when the object information is stored in sequence time, which is produced by the option `-event` supplied to the `Tcl database` command). To enable viewing object changes in delta times, you need to run the simulator with the `-scSyncEveryDelta` option turned on. For mixed SystemC and HDL designs, this option is on by default; for pure SystemC designs, it must be explicitly turned on. For more information about using this option with the simulator, see [Synchronizing Delta Cycles](#).

However, if a SystemC object declares the method `ncsc_need_sequence_probe`, the object's value change is always recorded in sequence time, whether or not the database was opened with the `-event` option. For more information about the `ncsc_need_sequence_probe` and other methods to specify the supported Tcl commands for an arbitrary SystemC object, see [Specifying Supported Commands](#).

It is possible to probe ports, signals, objects derived from `ncsc_viewable`, as well as memory content, of a type that represents a user-defined C++ structure or class. When probing objects of such types, you must provide an output operator `<<` for the structure or class. The `probe` command calls this output operator and writes its output as a string value to the SHM database. For example, assuming the name of the type is `T`, the type of the output operator function is:

```
inline ostream&
operator << (ostream& os, const T& v) {
    //write out v in some format to the output    //stream os
    return os;
}
```

For instance, the command:

```
xcelium> probe scope_name
```

probes objects defined in the scope excluding `ncsc_viewable` objects; and the command:

```
xcelium> probe scope_name -all -variables
```

includes probing `ncsc_viewable` objects declared in the scope.

Similarly, the command:


```
xcelium> probe scope_name -all -sc_processes
```

includes probing `sc_process` objects in the scope.

Related Topics

- [Using the Memory Viewer](#)
- [Debugging SystemC Objects and TLM Ports](#)
- [Tcl Commands for Debugging Processes](#)

Enabling a Probe

Use the `-enable` option with the `probe` command to resume previously disabled probe(s).

As soon as the probe resumes, all objects in the probe have their values written to the database.

The argument to `-enable` is a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

- The asterisk (`*`) matches any number of characters.
- The question mark (`?`) matches any one character.

Syntax

```
probe -enable <probe_name> [<probe_name>...]
```

Options

```
-enable <probe_name>  
[<probe_name>...]
```

Resumes the probe(s) specified by the argument. The argument can be a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

Example

The following command enables the probe named `peek`, which was previously disabled.

```
xcelium> probe -enable peek
```

Deleting a Probe

Use the `-delete` option with the `probe` command to delete a probe.

The argument to `-delete` is a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

- The asterisk (`*`) matches any number of characters.
- The question mark (`?`) matches any one character.

You can delete SHM probes at any time.

You can delete VCD or EVCD probes only at the time the VCD or EVCD database is created. Once the simulation has advanced, the VCD or EVCD header is written to the file and no modifications to the probes are possible.

You cannot delete probes that record statement trace data into a database opened with a `database -statement` command.

Syntax

```
probe -delete <probe_name> [<probe_name>...]
```

Options

```
-delete <probe_name>  
[<probe_name>...]
```

Deletes the probe(s) specified by the argument. The argument can be a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

Examples

The following command deletes the probe named `peek1`.

```
xcelium> probe -delete peek1
```

The following command uses the `?` wildcard to search for and delete the matching probe names, like `peek2` and `peek3`, or `peekA` and `peekB`.

```
xcelium> probe -delete peek?
```

process

The Tcl `process` command displays information about the processes that are currently executing or that are scheduled to execute at the current simulation time. You can use this command to display the processes in VHDL, in Verilog, or in mixed VHDL/Verilog designs.

For VHDL, the following constructs qualify as processes:

- VHDL process statements
- Postponed processes
- Concurrent statements (implicit process)
- Foreign processes (FMI)

For Verilog, the following constructs qualify as processes:

- `initial` and `always` statement blocks
- Continuous assignments
- Implicit continuous assignments (port connections)
- Non-blocking assignments
- `$monitor` statements

You must compile the source code with the `-linedebug` option (non-optimization mode) in order for the `process` command to display accurate information.

The `process` command can apply to SystemC processes if specified with `-current`, `-next`, or `-all`. In this case, it reports only the full paths of any running and scheduled SystemC processes, and their effective sensitivity. This command does not report any source code information for SystemC processes.

process Command Syntax

```
process
  [-all [<count>]]
  [-current]
  [-eot [<count>]]
  [-forked [<count>]]
  [-next [<count>]]
```

process Command Options

This section describes the options that you can use with the Tcl `process` command.

| | |
|--------------------------------------|--|
| <code>-all [<count>]</code> | <p>Lists all of the processes that are scheduled to execute at the current simulation time. This includes:</p> <ul style="list-style-type: none">• The process that is currently executing (<code>-current</code>).• All processes that are scheduled to execute in the current delta cycle (<code>-next</code>).• All process that are scheduled to execute at the end of the current simulation time (<code>-eot</code>). <p>If you specify a <code>count</code> argument, the number of processes in each of these categories is limited to the specified number.</p> |
| <code>-current</code> | <p>Lists the process that is currently executing. This is the default behavior if no <code>process</code> option is specified.</p> |
| <code>-eot [<count>]</code> | <p>Lists all of the processes that are scheduled to execute at the end of the current simulation time.</p> <p>This option shows the processes that are scheduled to execute after normal activity for the current time has stabilized. These processes include postponed processes, non-blocking assignments, and <code>\$monitor</code> statements. Postponed processes and <code>\$monitor</code> statements are guaranteed to be the last processes run in the current time, but other end-of-time process activity may cause further delta cycles.</p> <p>Include a <code>count</code> argument to limit the number of processes that are displayed.</p> |
| <code>-forked [<count>]</code> | <p>Lists all forked Verilog processes.</p> <p>Include a <code>count</code> argument to limit the number of processes that are displayed.</p> |
| <code>-next [<count>]</code> | <p>Lists all of the processes that are scheduled to execute at the current simulation time.</p> <p>Include a <code>count</code> argument to limit the number of processes that are displayed.</p> |

process Command Examples

Verilog Examples

The Verilog source code used in the examples in this section is shown below. This source code was compiled with the `-linedebug` option.

```
module top_module;
  reg[1:0]x, y, z;
  wire [1:0]sum, co;
  initial
    $monitor($time,, "x=%b y=%b z=%b sum=%b co=%b", x, y, z, sum, co);
  initial
    begin
      x=2'b00;
      y=2'b00;
      z=2'b00;
    end
  always
    x=#10 x+1;
  always
    y=#40 y+1;
  always
    z=#160 z+1;
endmodule
```

The following command with no command-line option is the same as `process -current`. This displays the full pathname of the executing process, as well as the file name and the source code line number.

```
% xmsim worklib.top_module
xcelium> process
Executing Process:
[./test.v, 6, top_module]      initial
```

The following command uses the `-all` option to show the currently executing process and all scheduled processes, including those that are scheduled at the end of the current simulation time.

```
xcelium> run -process
./test.v:11      x=2'b00;
xcelium> process -all
Executing Process:
[./test.v, 9, top_module]      initial

Scheduled Process(es):
[./test.v, 16, top_module]     always
[./test.v, 19, top_module]     always
[./test.v, 22, top_module]     always

End-of-time Process(es):
```

```
[./test.v, 7, top_module]      $monitor($time,, "x=%b y=%b z=%b sum=%b      co=%b",  
x, y, z, sum, co)
```

The following command line includes a *count* argument to limit the number of processes that are displayed.

```
xcelium> process -all 1  
Executing Process:  
[./test.v, 9, top_module]      initial  
  
Scheduled Process(es):  
[./test.v, 16, top_module]     always  
  
End-of-time Process(es):  
[./test.v, 7, top_module]      $monitor($time,, "x=%b y=%b z=%b sum=%b co=%b", x, y,  
z, sum, co)
```

The following command shows the next two processes that are scheduled to execute.

```
xcelium> process -next 2  
Scheduled Process(es):  
[./test.v, 16, top_module]     always  
[./test.v, 19, top_module]     always
```

The following command shows the processes that are scheduled to execute at the end of the current simulation time.

```
xcelium> process -eot  
End-of-time Process(es):  
[./test.v, 7, top_module]      $monitor($time,, "x=%b y=%b z=%b sum=%b co=%b", x, y,  
z, sum, co)
```

VHDL Example

The VHDL source code used in the examples in this section is shown below. This source code was compiled with the `-linedebug` option.

```
entity TOP_TESTBENCH is
end TOP_TESTBENCH;

architecture SCHEMATIC of TOP_TESTBENCH is
  shared variable try : integer := 0;
  signal clk : bit := '1';
  begin

  Process_Loop: process (clk)
  Begin
    assert false report "Process_Loop " severity note;
    try := try + 1;
  end process;

  Process_clk: process
  Begin
    wait for 50 ns;
    clk <= NOT clk;
  end process;

  Process_later: postponed process (clk)
    variable prevTry : integer := 1;
  Begin
    if try /= (prevTry + 5) then
      assert false report "Did not get postponed" severity note;
    end if;
    prevTry := try;
  end process;

end SCHEMATIC;
```

The following command with no command-line option is the same as `process -current`. This displays the full pathname of the executing process, as well as the file name and the source code line number.

```
xcelium> process
Executing Process:
[./test.vhd, 15, :Process_clk] Process_clk: process
```

The following command uses the `-all` option to show the currently executing process and all scheduled processes, including those that are scheduled at the end of the current simulation time.

```
xcelium> process -all
Executing Process:
[./test.vhd, 15, :Process_clk] Process_clk: process

Scheduled Process(es):
[./test1.vhd, 10, :Process_Loop]      Process_Loop: process (clk)
[./test1.vhd, 20, :Process_later]    Process_later: postponed process (clk)
```

```
End-of-time Process(es):  
None
```

The following command shows all processes that are scheduled to execute.

```
xcelium> process -next  
Scheduled Process(es):  
[./test1.vhd, 10, :Process_Loop]      Process_Loop: process (clk)  
[./test1.vhd, 20, :Process_later]    Process_Later: postponed process (clk)
```

The following command line includes a *count* argument to limit the number of processes that are displayed.

```
xcelium> process -next 1  
Scheduled Process(es):  
[./test1.vhd, 10, :Process_Loop]      Process_Loop: process (clk)
```

Mixed Verilog/VHDL Example

The following command uses the `-all` option to illustrates the output for a mixed-language design.

```
xcelium> process -all  
Executing Process:  
[File: tb.vhd, Line: 153]              :$PROCESS_001  
  
Scheduled Process(es):  
[File: tb.vhd, Line: 152]              :$PROCESS_000  
[File: tb.vhd, Line: 134]              :gen_quarters  
[File: tb.vhd, Line: 122]              :gen_dimes  
[File: tb.vhd, Line: 114]              :gen_nickels  
...  
...  
...  
[File: DRINK.vhd, Line: 356]            :top:$PROCESS_002  
[File: DRINK.vhd, Line: 355]            :top:$PROCESS_001  
[File: DRINK.vhd, Line: 354]            :top:$PROCESS_000  
module DRINK_MACHINE:  
  
[File: ./SOURCES/drink_machine.v, Line: 54]    initial $sdf_annotate("SDFFILE")  
  
End-of-time Process(es):  
No Processes Scheduled
```

Related Topic

- [Tcl Commands for Debugging Processes](#)

profile

The Tcl `profile` command invokes the profiler and controls its behavior.

This command supports the two Xcelium profilers: the Basic Profiler and the Advanced Profiler. The Basic Profiler generates a simple text file report that contains stream, module, and summary-level information. This report is generated at run time and is called `xmprof.out` by default. The Advanced Profiler measures where CPU time is spent during the simulation and tracks dynamic memory consumption at run time. It dumps a binary database once the simulation run is complete. You can then use the `xprof` utility to load the database in a UI and show the percentage grade of time consumed by each aspect of the design and verification environment through different views.

With Xcelium, you invoke the profiler in one of the following ways:

- Use the `-xprof` command-line option with `xrun` or `xmelab` to enable instrumented profiling and dump the binary database.
- Use the `-profile` command-line option with `xrun` or `xmsim` to enable basic run-time profiling and save the `xmprof.out` report.
- Use the Tcl `profile` command.

This command provides more control over the profiler behavior than the available command-line options. With the Tcl `profile` command, you can:

- Start or stop profiling at any time (`-on` and `-off`).
- Dump the profiled data to a file at any time (`-dump`).
- Clear the currently collected profiled data (`-clear`).

If profiling is turned on with the `profile` command, it remains on after a reset. A `reset` command clears the currently collected profile data but does not disable profiling.

profile Command Syntax

```
profile
  -clear
  -dump [-overwrite] [<name|dir>]
  -off
  -on
```

Multiple options are allowed when using this command, but you must specify at least one option.

profile Command Options

This section describes the options that you can use with the Tcl `profile` command.

| | |
|--|---|
| <code>-clear</code> | <p>Clears the collected profile information by clearing the internal information buffer.</p> <p>You can use this option only when the profiler is on. It does not disable profiling.</p> |
| <code>-dump [-overwrite [<name dir>]]</code> | <p>Writes the profile data at any time during a simulation.</p> <p>By default, this option uses the Basic Profiler to write information to the <code>xmprof.out</code> file. You can specify a <i>name</i> as an argument to <code>-dump</code> to customize the output filename.</p> <p>If you use the <code>-xprof</code> command-line option with <i>xrun</i> or <i>xmelab</i>, the <code>-dump</code> option instead writes its information to the Advanced Profiler database. In this case, the database is located in the directory <code>./xprof_report_dir/design/test</code>. You can choose a different <i>directory</i> as an argument to <code>-dump</code> and customize this location.</p> <p>The profiler does not overwrite existing files automatically. If an output file or database already exists, you must include the <code>-overwrite</code> option to save over the existing data.</p> |
| <code>-off</code> | <p>Turns off the profiler.</p> |
| <code>-on</code> | <p>Turns on the profiler.</p> |

profile Command Examples

The following command invokes the Basic Profiler with `profile -on`. Separate profiles are created for different simulation times.

```
% xrun -tcl -v93 -access +rwc -top test test.vhd
...
...
Loading snapshot worklib.test:arch_test ..... Done
xcelium> profile -on                ;# Enable profiling.
xcelium> run 1000 ns
xcelium> profile -dump profdata1.out ;# Write profile data to profdata1.out.
xcelium> set vital_timing_checks_on 1 ;# Set variables, execute Tcl commands, etc.
xcelium> run 1000 ns
xcelium> profile -dump profdata2.out ;# Write profile data to profdata2.out.
                                   ;# File contains profile data for 2000 ns.
```

```
xcelium> set vital_timing_checks_on 0
xcelium> run 1000 ns
xcelium> profile -dump -overwrite profdata1.out      ;# Write profile data to
                                                    ;# profdata1.out.
xcelium>                                           ;# File contains profile data
                                                    ;# for 3000 ns.
xcelium> exit
```

The following command generates profile results for a simulation run. The simulation is then reset. The value of a Tcl variable is changed, a Tcl command is executed, and the simulation is run again with the profile results dumped to a separate file.

```
xcelium> profile -on                                ;# Turn profiling on.
xcelium> run
xcelium> profile -dump file1.prof ;# Write profile results to file1.prof.
xcelium>
xcelium> reset                                     ;# Reset simulation to time 0 ns.
                                                    ;# This clears the profile data, but profiling is still on.
xcelium> set real_precision 5                      ;# Set variables, execute Tcl commands, etc.
xcelium> tcheck -off top.y1.u2
xcelium> run
xcelium> profile -dump file2.prof ;# Write profile results of second simulation run.
xcelium> profile -off                              ;# Turn profiling off.
xcelium> exit
```

The following `xrun` command enables the Advanced Profiler with the `-xprof` option. The profiled time intervals are:

- 0 NS to 1000 NS
- 2000 NS to 3000 NS

```
% xrun -tcl -v93 -access +rwc -iprof -top test test.vhd
...
...
Loading snapshot worklib.test:arch_test ..... Done
xcelium> run 1000ns
Ran until 1 US + 0
xcelium> profile -dump mytest1                      ;# Dump profile results to database
                                                    ;# in ./xprof_report_dir/mytest1/prof_db
xcelium>
xcelium> profile -off                                ;# Turn profiling off
xcelium> run 1000ns
Ran until 2 US + 0
xcelium> profile -on                                ;# Turn profiling on
xcelium> run 1000ns
Ran until 3 US + 0
xcelium> profile -dump -overwrite mytest1           ;# Dump profile results to database
```

```
;# in ./xprof_report_dir/mytest1/prof_db
```

```
xcelium> exit
```

Related Topics

- [Using the Basic Profiler](#)
- [Advanced Profiler](#)

release

The Tcl `release` command releases any force set on the specified object(s). Releasing a force causes the value to immediately return to the value that would have been present if the force hadn't been blocking transactions. Objects specified as arguments to the `release` command must have write access. See [Access to Simulation Objects](#) for details on specifying access to simulation objects. Use the `-keepvalue` option if you want to release the forced object, but retain the forced value.

This command releases any force, whether it is created by a `force` command or by a Verilog `force` procedural statement during simulation. The behavior is the same as that of a Verilog `release` statement.

The following flows have special use cases:

- For Low-Power Simulation, you must pass the `-lps` option to the `release` command to remove the force on a specified supply net. Releasing a force on a supply net without `-lps` results in a PNOOBJ (path element not found) error.
- For SystemC, you must pass the `-keepvalue` option to the `release` command. Releasing a force without the `-keepvalue` option causes the command to immediately return the object to the value that it would have if the force was not blocking transactions. If the `-keepvalue` option is used, the command keeps the forced value even after the release.

The following objects cannot be forced to a value with the `force` command and, therefore, cannot be specified as the object in a `release` command:

- memory
- memory element
- bit-select or part-select of a register
- VHDL variable

release Command Syntax

```
release object_name ...  
    [-after <timespec>]  
    [-keepvalue]  
    [-lps]
```

release Command Options

This section describes the options that you can use with the Tcl `release` command.

| | |
|--------------------------------------|--|
| <code>-after <timespec></code> | Releases the forced object after a certain time interval. You can set the <i>timespec</i> value in fs, ps, ns, us, or ms. |
| <code>-keepvalue</code> | Releases the forced object, but retains the forced value. |
| <code>-lps</code> | Releases any forces on the specified supply net (for low-power simulation). |

release Command Limitations

Normally, you can use wildcard characters in the *object_name* argument to the `release` command.

- The asterisk (`*`) matches any number of characters.
- The question mark (`?`) matches any one character.

However, you cannot use wildcard characters inside escaped names or if the *object_name* is a supply net for low-power simulation.

release Command Examples

The following command removes a force set on object `x`:

```
xcelium> release x
```

The following command removes a force set on object `:top:DISPENSE_tempsig`:

```
xcelium> release :top:DISPENSE_tempsig
```

The following command releases two objects: `w[0]` and `r`:

```
xcelium> release w[0] r
```

The following command releases all forces applied on objects in the current scope that have names that end in `td`:

```
xcelium> release *td
```

The following command includes the `-lps` option to release the force on the supply net named `tb.dut.t1.VDD`:

```
xcelium> release -lps tb.dut.t1.VDD
```

Related Topics

- [Removing the Force on a Supply Net](#)
- [Wildcards Characters in Tcl Commands](#)

reset

The Tcl `reset` command resets the currently loaded model to its original state at time zero. The time-zero snapshot, created by the elaborator, must still be available.

The Tcl debug environment remains the same as much as possible after a reset.

- Tcl variables remain as they were before the reset.
- SHM and VCD databases remain open, and probes remain set.

VCD databases created with the `$dumpvars` call in Verilog source code are closed when you reset.

- Breakpoints remain set.
- The SimVision waveform viewer window remain the same.

Forces and deposits in effect at the time you issue the `reset` command are removed.

When you run the simulation after a reset, the order in which commands are executed to set probes, set breakpoints, and so on, may be different from the order in which they were executed before the reset. However, the commands are still executed at the correct simulation time cycle.

While the `reset` command is generally supported only when models are dynamically linked against the shared object version of the SystemC libraries, an alternative implementation of the `reset` command allows you to re-invoke the simulation process and reload the simulation environment when dynamically linking SystemC libraries as an archive (`libsystemc_ar.a`) or loading SystemC models as executables. You can also use this alternative feature to reset the simulation after it has been restarted or re-invoked. To use the alternative `reset` command, set the environment variable `FORCE_RESET_BY_REINVOKE`.

When you use the alternative `reset` command, the simulator issues a message to warn you about possible differences in the simulation results, and also about the differences in resulting simulation databases due to deletion and re-addition of the probe commands.

You cannot use the alternative implementation of the `reset` command in GUI mode, that is when SimVision is attached to the simulation.

reset Command Syntax

```
reset
    -sync
```

reset Command Options

This section describes the options that you can use with the Tcl `reset` command.

| | |
|--------------------|---|
| <code>-sync</code> | <p>Syncs the snapshot to the last elaborated snapshot. Using the <code>reset</code> command with the <code>-sync</code> option removes the following:</p> <ul style="list-style-type: none">• All line, function and object breakpoints• Probes and databases (<code>shm</code>, <code>vcd</code>, <code>evcd</code>)• All callbacks• Resets the scope to top of the design <p>You must re-elaborate the design (without changing compilation and elaboration options) before issuing the <code>reset -sync</code> command.</p> <p>When using the <code>-sync</code> option with <code>reset</code> command, shared object (*.so) files are not reloaded. These include <code>librun.so</code>, <code>libvpi.so</code>, <code>libvhpi.so</code>, <code>libcfc.so</code>, <code>libdpi.so</code>, <code>sv_export.so</code>, and other user-defined shared objects, and VIP related shared objects. Hence, the changes that go into these shared objects such as code in the *.c/cpp files are not reflected after using the <code>-sync</code> option. No change should be made in the signatures of DPI export functions.</p> |
|--------------------|---|

reset Command Examples

The following command resets the currently loaded model to its original state at time zero:

```
xcelium> reset
```

restart

The Tcl `restart` command replaces the currently simulating snapshot with another snapshot of the same elaborated design.

You must specify a snapshot name with the `restart` command, and the specified snapshot must be a snapshot created by the `save` command.

The snapshot name is interpreted the same way as the snapshot name on the *xmsim* command line, with the addition that you can give only the view name preceded by a colon if you want to load a snapshot that is a view of the currently loaded cell. For example:

| | |
|-----------------------------------|---|
| <code>restart top</code> | Restarts <code>[lib.]top[:view]</code> If the view name is omitted, there must be only one snapshot of the given cell, otherwise, the snapshot name is ambiguous. In this case, an error message is issued, and a list of available snapshots is printed. |
| <code>restart top:ckpt</code> | Restarts <code>[lib.]top:ckpt</code> |
| <code>restart :ckpt</code> | Restarts <code>[lib.][cell]:ckpt</code> |

An error message is issued if the snapshot specified on the command line is not a snapshot of the design hierarchy that is currently loaded. That is, you cannot use the `restart` command to load a snapshot of a different elaborated design or one that comes from a different elaborated design. To load a different model, exit *xmsim* and then invoke it with the new snapshot.

When you restart with a saved snapshot in the same simulation session:

- SHM databases remain open and all probes remain set.
- Breakpoints set at the time that you execute the restart remain set.

If you set a breakpoint that triggers, for example, every 10 ns (that is, at time 10, 20, 30, and so on) and restarts with a snapshot saved at time 15, the breakpoint triggers at 20, 30, and so on, not at time 25, 35, and so on.

- Forces and deposits that are in effect at the time you issue a `save` command are still in effect when you restart.

If you exit the simulation and then invoke the simulator with a saved snapshot, databases are closed. Any probes and breakpoints are deleted. If you want to restore the full Tcl debug environment when you restart, make sure that you save the environment with the `save -environment` command. This command creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. You can then use the Tcl `source` command after restarting or the `-input` option when you invoke the simulator to execute the script. For example:

```
% xmsim top
xcelium> (Open a database, set probes, set breakpoints, deposits, forces, etc.)
xcelium> run 100 ns
xcelium> save worklib.top:ckpt1
```



```
xcelium> save -environment ckpt1.tcl
xcelium> exit
% xmsim -tcl worklib.top:ckpt1
xcelium> source ckpt1.tcl
```

You can save a snapshot to a particular location by using a `save -path` command. To restart this snapshot, use the `restart -path` command.

For a design containing SystemC, the `restart` command restarts the simulation snapshot previously generated with the `save` command from a simulation session. You must specify a snapshot name with the `restart` command, and the specified snapshot must be a snapshot created by the `save` command.

Use the `-savedb` option to save the existing SHM databases from being overwritten after a restart from the saved snapshot.

restart Command Syntax

```
restart <snapshot_name>
    [-path <snapshot_dir>]
    [-savedb]
    [-show]
    [-snload e_file [e_file ...]]
    [-snlogappend]
    [-snseed new_seed]
```

restart Command Options

This section describes the options that you can use with the Tcl `restart` command.

| | |
|---------------------------------|---|
| <code>-path snapshot_dir</code> | <p>Restarts the simulation with the snapshot saved at the specified location.</p> <p>This option lets you restart with a snapshot that was saved to a specific location with a <code>save -path</code> command. For example:</p> <pre>xcelium> save -path /path/to/new/snapshot/dir worklib.top:ckpt1 xcelium> [some Tcl commands] xcelium> restart -path /path/to/new/snapshot/dir worklib.top:ckpt1</pre> <p>This <code>restart</code> command first reads in the library at the location specified by the <code>-path</code> option, and then loads the specified snapshot.</p> |
| <code>-savedb</code> | <p>Saves current databases.</p> <p>This option is valid only when restarting a snapshot containing SystemC.</p> <p>The <code>-savedb</code> option is used to prevent existing SHM databases from being overwritten after a restart from the saved snapshot.</p> |

| | |
|--|--|
| <code>-show</code> | <p>Lists the names of all snapshots that can currently be used as the argument to the restart command.</p> |
| <code>-snload e_file</code> <code>[e_file ...]</code> | <p>Loads the specified e file(s) after loading the saved snapshot.</p> <p>The <code>-snload</code> option lets you:</p> <ol style="list-style-type: none">1. Run your test in a specific configuration until the DUT has been initialized or until some other point of interest has been reached.2. Save the current state of the simulation.3. Restart the saved snapshot in multiple runs with additional e files. Each run continues from the saved state, but with a different scenario. The code in the specified e files only affects the simulation from this point onwards. For example, new constraints do not affect already-generated structs. <p>Example:</p> <pre>xcelium> restart snap1 -snload file1.e file2.e</pre> <p>The additional e files specified with <code>-snload</code> are called <i>Dynamic Loadable Files</i> (DLF). These files can contain only a subset of e.</p> <p>The <code>-snload</code> option requires a Specman_Advanced_Option license, and works only when the Specman generator is set to Intelligen.</p> |
| <code>-snlogappend</code> | <p>Appends to the Specman log file any log files previously saved with the <code>-snwithlogs</code> option. See Creating a Single Log File for Multiple Simulations.</p> |
| <code>-snseed new_seed</code> | <p>Specifies a new seed for re-seeding Specman after loading the saved snapshot.</p> <p>The <code>-snseed</code> option lets you:</p> <ol style="list-style-type: none">1. Run your test in a specific configuration until the DUT has been initialized or until some other point of interest has been reached.2. Save the current state of the simulation.3. Restart the saved snapshot in multiple runs, assigning a different value to the generation seed. Each run continues from the saved state. The different seed causes different generation results, and thus different scenarios. <p>The <code>new_seed</code> argument can be a number or the word <code>random</code>.</p> <p>Examples:</p> <pre>xcelium> restart snap1 -snseed 4 xcelium> restart snap1 -snload file1.e -snseed random</pre> <p>The <code>-snseed</code> option requires a Specman_Advanced_Option license, and works only when the Specman generator is set to Intelligen.</p> |

restart Command Examples

In the following example, a `save` command is issued to save the simulation state as a view of the currently loaded cell, `top`. This snapshot can be loaded using either of the following two `restart` commands:

```
xcelium> save top:ckpt1
xcelium> restart top:ckpt1
xcelium> restart :ckpt1
```

In the following example, a `save` command is issued to save the simulation state as a view of the currently loaded cell, `top`. A second `save` command is issued to save the Tcl debug environment. If you exit the simulator, you can restart with the saved snapshot and then restore the debug settings by sourcing the script created with the `save -environment` command:

```
xcelium> save top:ckpt1
xcelium> save -environment top_ckpt1.env
xcelium> exit
% xmsim -tcl :ckpt1
xcelium> source top_ckpt1.env
```

The following command reloads the snapshot of the given cell, `top`. Because the view name is not specified, the snapshot name is ambiguous if there is more than one view, and an error message is issued:

```
xcelium> restart top
```

The following `restart` command results in an error because you are trying to replace the currently simulating snapshot (`asic1:ckpt1`) with another snapshot of a different elaborated design (`asic2:ckpt1`). You can only restart snapshots of the same elaborated design:

```
% xmsim -tcl asic1:ckpt1
xcelium> restart asic2:ckpt1
```

The following command lists all of the snapshots you can currently load with the `restart` command:

```
xcelium> restart -show
otherlib.board:module
worklib.board:ckpt1
worklib.board:ckpt2
```

Related Topics

- [Restarting SystemC Simulations](#)
- [Saving SystemC Simulations](#)

run

The Tcl `run` command starts simulation or resumes a previously halted simulation. With the `run` command you can:

- Run until an interrupt, such as a breakpoint or error, occurs or until simulation completes (the `run` command with no modifiers or arguments).
- Run one behavioral statement, stepping over subprogram calls (`-next`).
- Run one behavioral statement, stepping into subprogram calls (`-step`).
- Run until the current subprogram ends (`-return`).
- Run to a specified timepoint or for a specified length of time (`-timepoint`).
- Run to the beginning of the next delta cycle or to a specified delta cycle (`-delta`).
- Run to the beginning of the next phase of the simulation cycle (`-phase`).
- Run until the beginning of the next scheduled process or to the beginning of the next delta cycle, whichever comes first (`-process`).

run Command Syntax

```
run
  -adjacent
  -analogsolver
  -clean
  -delta [cycle_spec]
  -eoi
  -next
  -phase
  -process
  -rand_solve
  -return
  -step
  -sync
  [-timepoint] [time_spec] [-absolute | -relative]
  -vda
```

run Command Options

This section describes the options that you can use with the Tcl `run` command.

| | |
|--|--|
| <code>-adjacent</code> | <p>Runs one behavioral statement, remaining in the current process.</p> <p>The <code>run -adjacent</code> command is supported for Verilog processes only.</p> <p>A <code>run -next</code> or <code>run -step</code> command stops at the next executable line of code to be processed by the simulator. This next line of code could be anywhere in the design hierarchy.</p> <p>A <code>run -adjacent</code> command also steps to the next line of executable code, but remains within the currently executing process or thread. This is useful when debugging a complex algorithm with many conditional flows through the code because it lets you retain focus on a specific area of the code, rather than on the next scheduled event in the simulation.</p> |
| <code>-analogsolver</code> | <p>Reboots the analog solver to the exact setup you left at during the last run.</p> |
| <code>-clean</code> | <p>Runs the simulation to the next point at which it is possible to create a checkpoint snapshot with the <code>save -simulation</code> command. See save.</p> |
| <code>-delta</code> <code>[cycle_spec]</code> | <p>Runs the simulation for the specified number of delta cycles. If no <code>cycle_spec</code> argument is specified, run the simulation to the beginning of the next delta cycle. A <code>run -delta</code> command is the same as <code>run -delta 1</code>.</p> |
| <code>-eoi</code> | <p>Runs until the end of initialization.</p> <p>This option runs until all variable declaration initialization assignments (vda's) and initial blocks are complete, or until a delay/event clause is encountered.</p> <p>This command is intended to provide time 0 initialization. If a delay or event clause is found within a given initial block, that block runs up until the point of the delay or event. If there are no initial blocks to run a warning is issued.</p> <p>In the following example, an <code>initial</code> block is being used to reset a design. The <code>initial</code> blocking assignment <code>reset = 0;</code> is unencumbered by any delay or event clauses, so it would be evaluated at time 0. The subsequent statements in this block would need to be rescheduled. Consequently, they would not be evaluated by the <code>run -eoi</code> command.</p> |

```
module top;
    reg reset;
    initial begin
        reset = 0; // no delay/event clause. This runs at time '0'
        #0 reset = 1; // delay clause. This is re-scheduled
        @(posedge reset) #10 reset = 0; // event clause. This is re-
        scheduled.
    end
    ...
endmodule
```

Note that the `#0` syntax is special in that the delay is until the end of the current time slice, it is still a re-schedule action that would remove the initial block from the cycle list, thus removing it from further consideration for the `run -eoi` command.

| | |
|--------------------------|---|
| <code>-next</code> | <p>Runs one behavioral statement, stepping over any subprogram calls.</p> <p>In some cases, a <code>run -next</code> command produces results that are similar to a <code>run -step</code> command when the current execution point is a VHDL non-zero <code>WAIT</code> statement. For example, if the current execution point is a statement such as <code>wait for 10 ns;</code>, another process may be scheduled to run at the current simulation time while the current process is suspended because of the <code>WAIT</code> statement. A <code>run -next</code> command executes the next behavioral statement, and the simulation stops in the scheduled process. If you want to run to the next executable line in the source code after the <code>WAIT</code> statement, you can set a line breakpoint on that line and enter a <code>run</code> command.</p> |
| <code>-phase</code> | <p>Runs to the beginning of the next phase of the simulation cycle. A simulation cycle consists of two phases: signal evaluation and process execution.</p> |
| <code>-process</code> | <p>Runs until the beginning of the next scheduled process or to the beginning of the next delta cycle, whichever comes first.</p> <p>In VHDL, a process is a process statement. In Verilog it is an <code>always</code> block, an <code>initial</code> block, or some other behavior that can be scheduled to run.</p> |
| <code>-rand_solve</code> | <p>Executes the current SystemVerilog <code>randomize()</code> call and display the status (1 for success or 0 for failure).</p> <p>The SystemVerilog built-in <code>randomize()</code> function returns the value 1 for success or 0 for failure. A failure occurs because there are conflicts in the collection of constraints to solve or because a variable is over-constrained. In this case, the simulator generates a warning telling you that the <code>randomize()</code> call failed.</p> <p>To help you debug these randomization failures, you can set a breakpoint in <code>randomize()</code> method calls using the <code>stop -randomize</code> command. Then, when the simulator is stopped in a <code>randomize()</code> call, you can use other commands to debug the failures. For example, you can:</p> <ul style="list-style-type: none">• Enable/disable specific constraints with a <code>deposit -constraint_mode</code> command.• Enable/disable specific random variables with a <code>deposit -rand_mode</code> command.• Add a new constraint to the class <code>randomize</code> call you are debugging with the <code>constraint</code> command. <p>After executing these, or other, Tcl commands, use the <code>run -rand_solve</code> command to execute the current randomization call again, using the currently enabled/disabled constraints and variables and the current state variable values.</p> <p>No other <code>run</code> command options can be included on the command line when using the <code>-rand_solve</code> option.</p> <p>New random values are generated for each <code>run -rand_solve</code> command. After re-executing the <code>randomize()</code> call, the simulator stops and returns the Tcl prompt. If the solver succeeds, it copies the newly generated random values to the <code>rand</code> and <code>randc</code> variables. If the solver fails, the copy is not done.</p> <p>Use the <code>run</code> command to continue out of the <code>randomize()</code> call..</p> |
| <code>-return</code> | <p>Runs until the current subprogram (task, function, procedure) returns.</p> |
| <code>-step</code> | <p>Runs one behavioral statement, stepping into subprogram calls.</p> |

| | |
|---|---|
| <code>-sync</code> | <p>Runs to the next point at which the digital engine synchronizes with the analog engine.</p> |
| <code>[-timepoint]</code> <code>[time_spec] [-</code> <code>absolute -</code> <code>relative]</code> | <p>Runs until the specified time is reached. The time specification can be absolute or relative. Relative is the default.</p> <p>In addition to time units such as fs, ps, ns, us, and so on, you can use <code>deltas</code> as the unit. For example:</p> <pre>xcelium> run 10 deltas</pre> <p>This is the same as <code>run -delta 10</code>.</p> <p>If you include a time specification and a breakpoint or interrupt stops simulation before the specified time is reached, the time specification is thrown away. For example, in the following sequence of commands, the last <code>run</code> command does not stop the simulation at 500 ns.</p> <pre>xcelium> stop -object x Created stop 1 xcelium> run 500 ns Stop 1 {x = 0} at 10 ns xcelium> run</pre> <p>Using <code>run -timepoint</code> without a <code>time_spec</code> argument runs the simulation until the next scheduled event.</p> |
| <code>-vda</code> | <p>Runs until all SystemVerilog Variable Declaration Assignments (VDA) are complete.</p> <p>This option runs until the following conditions have been met:</p> <ul style="list-style-type: none">• No Variable Declaration Assignments are present in the design• The simulation is not at time 0• A previous <code>run -vda</code> command has already been issued, which would imply that no VDAs were found <p>This command allows you to perform all VDAs, but stop before initial/always statements or continuous assignments are evaluated. This applies to static variable declaration assignments, not automatic declaration assignments.</p> <pre>xcelium> run -vda ... xcelium> run -vda xmsim: *W,RNNVDA: No SystemVerilog Variable Declaration Assignments found.</pre> |

run Command Examples

The following command runs the simulation until an interrupt occurs or until simulation completes:

```
xcelium> run
```

The following command advances the simulation to 500 ns absolute time. The `-timepoint` option is not required:

```
xcelium> run -timepoint 500 ns -absolute
```

The following command advances the simulation 500 ns relative time. With a time specification, `-relative` is the default:

```
xcelium> run 500 ns
```

The following two commands are equivalent. They both run the simulation for 5 delta cycles:

```
xcelium> run -delta 5
xcelium> run 5 deltas
```

The following command runs one behavioral statement, stepping into any subprogram calls:

```
xcelium> run -step
```

The following command runs until the current subprogram returns. The subprogram can be a task, function, or procedure:

```
xcelium> run -return
```

The following command runs one behavioral statement, stepping over any subprogram calls:

```
xcelium> run -next
```

The following example illustrates the difference between `run -next` and `run -adjacent`. The Verilog code used for this example is as follows:

```
module top;
    int i;

    initial begin
        // Wait for event on i
        @i;                                // Line 6
        $display($stime, " spot 1");        // 7
        // Wait 5 time units                // 8
        #5;                                // 9
        $display($stime, " spot 2");        // 10
        // Wait for another event on i      // 11
        @i;                                // 12
        $display($stime, " spot 3");        // 13
    end                                    // 14
                                        // 15

    initial begin                          // 16
        #10 i = i + 1;                     // 17
        #10 i = i + 1;                     // 18
    end
endmodule
```

A `run -next` command stops at the next executable line of code to be processed by the simulator. This line of code could be anywhere in the design hierarchy. On the other hand, a `run -adjacent` command stops at the next line of executable code within the current process or thread.

In this example, a sequence of `run -next` commands is issued. With each command, the simulator executes the next line of executable code, which could be in the first or the second `initial` block. The simulation is then reset and a sequence of `run -adjacent` commands is issued. The simulator executes the next line of executable code, but remains in the same process, the first `initial` block, until the process is exited.

```
xcelium> stop -line 6
Created stop 1
xcelium> run
0 FS + 0 (stop 1: ./test.sv:6)
./test.sv:6      @i;                                // Line 6
xcelium> run -next
./test.sv:17     #10 i = i + 1;                      // 17
xcelium> run -next
Stepped to 10 NS + 0
xcelium> run -next
./test.sv:17     #10 i = i + 1;                      // 17
xcelium> run -next
./test.sv:18     #10 i = i + 1;                      // 18
xcelium> run -next
10 NS + 0 (stop 1: ./test.sv:6)
./test.sv:6      @i;                                // Line 6
xcelium> run -next
./test.sv:7      $display($stime, " spot 1");        // 7
xcelium> run -next
10 spot 1
./test.sv:9      #5;                                // 9
xcelium> run -next
Stepped to 15 NS + 0
xcelium> run -next
./test.sv:9      #5;                                // 9
xcelium> run -next
./test.sv:10     $display($stime, " spot 2");        // 10
xcelium> run -next
15 spot 2
./test.sv:12     @i;                                // 12
xcelium> run -next
Stepped to 20 NS + 0
xcelium> run -next
./test.sv:18     #10 i = i + 1;                      // 18
xcelium> run -next
./test.sv:12     @i;                                // 12
xcelium> run -next
./test.sv:13     $display($stime, " spot 3");        // 13
xcelium> run -next
20 spot 3
```

```
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> ;#
xcelium> ;# Reset the simulation. Line breakpoint at line 6 remains set.
xcelium> reset
Loaded snapshot worklib.top:sv
xcelium> run
0 FS + 0 (stop 1: ./test.sv:6)
./test.sv:6      @i;                                // Line 6
xcelium> run -adjacent
10 NS + 0 (stop 1: ./test.sv:6)
Simulation time has advanced to 10 NS + 0
./test.sv:6      @i;                                // Line 6
xcelium> run -adjacent
./test.sv:7      $display($stime, " spot 1");        // 7
xcelium> run -adjacent
10 spot 1
./test.sv:9      #5;                                // 9
xcelium> run -adjacent
Simulation time has advanced to 15 NS + 0
./test.sv:9      #5;                                // 9
xcelium> run -adjacent
./test.sv:10     $display($stime, " spot 2");        // 10
xcelium> run -adjacent
15 spot 2
./test.sv:12     @i;                                // 12
xcelium> run -adjacent
Simulation time has advanced to 20 NS + 0
./test.sv:12     @i;                                // 12
xcelium> run -adjacent
./test.sv:13     $display($stime, " spot 3");        // 13
xcelium> run -adjacent
20 spot 3
xmsim: *N,ADJTHER: Thread exited under 'run -adjacent'; continuing as 'run -next'.
xmsim: *W,RNQUIE: Simulation is complete.
xcelium>
```

Debugging SystemVerilog Randomization Constraint Failures

The following example shows you how to set a breakpoint in `randomize()` calls and then use other commands, including the `run -rand_solve` command, to debug randomization failures.

The source code for the example is as follows:

```
// File: test.v
module top;

    integer i4, i5;

    class c1;
        rand integer r1;
        rand integer r2;
        rand integer r3;
        /* Conflicting constraints */
        constraint con1 { r1 == 111; }
        constraint con2 { r1 == 222; }
        constraint con3 { r2 == 888; }
    endclass

    c1 ch1 = new;
    integer res;

    initial begin
        ch1.r1 = 111;
        ch1.r2 = 777;
        ch1.r3 = 666;
        i4 = 444;
        i5 = 555;

        res = ch1.randomize();
        $display("ch1.r1 = %d", ch1.r1);
        $display("ch1.r2 = %d", ch1.r2);

        res = randomize(i4) with { i4 == i5; };

        $display("i4 = %d", i4);

    end
endmodule
```

```
% xmvlog -nocopyright -sv test.v
% xmelab -nocopyright -access +rwc top
% xmsim -nocopyright -tcl top

#; Set a breakpoint for all randomize() calls
xcelium> stop -create -randomize -always
Created stop 1
xcelium> run
./test.v:26 res = ch1.randomize();
#; Disable the constraint named con1
xcelium> deposit -constraint_mode con1 = 0
```

```

#; Execute the current randomize() call again
xcelium> run -rand_solve
xmsim: *N,DBGSLV: The randomization solver returned this status: 1 (success).
#; Display the values of variables r1 and r2
xcelium> value ch1.r1
222
xcelium> value ch1.r2
888
#; Enable the constraint named con1. Disable the constraint named con2.
xcelium> deposit -constraint_mode con1 = 1
xcelium> deposit -constraint_mode con2 = 0
xcelium> run -rand_solve
xmsim: *N,DBGSLV: The randomization solver returned this status: 1 (success).
xcelium> value ch1.r1
111
xcelium> value ch1.r2
888
#; Disable r1, con1, and con2
xcelium> deposit -rand_mode r1 = 0
xcelium> deposit -constraint_mode con1 = 0
xcelium> deposit -constraint_mode con2 = 0
xcelium> run -rand_solve
xmsim: *N,DBGSLV: The randomization solver returned this status: 1 (success).
xcelium> value ch1.r1
111
xcelium> value ch1.r2
888
xcelium> run
ch1.r1 =          111
ch1.r2 =          888
./test.v:30      res = randomize(i4) with { i4 == i5; };
xcelium> value i4
555
xcelium> deposit i5 = 999
xcelium> run -rand_solve
xmsim: *N,DBGSLV: The randomization solver returned this status: 1 (success).
xcelium> value i4
999
xcelium> run
i4 =              999
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

Stepping Through Lines of Code

You can use the `run` command to examine the order in which the simulator executes the statements in your model by stepping through the simulation line by line. Single-stepping through lines of code is an especially useful technique for debugging problems such as infinite loops. You can set a breakpoint so that the simulator stops where the problem occurs, and then step through the code line by line to see what is happening.

From the Tcl interface (the `xcelium>` command prompt or a Tcl input file):

- Use `run -step` to simulate to the next executable line of code in any scope. This command runs one statement, stepping into subprogram calls.
- Use `run -next` to run one statement, stepping over any subprogram calls.


You cannot single-step one *line* at a time or set line breakpoints in a particular design unit unless you have compiled that unit with the `-linedebug` option. If you have compiled the unit without this option, the `run -step` or `run -next` commands run the simulation until the next point where it can stop. If execution is passed to a unit that was compiled with `-linedebug`, full single stepping is resumed.

save

Use the Tcl `save` command to create a snapshot of the current simulation state. You can then use the `restart` command to load the saved snapshot and resume simulation.

The current simulation state that is saved in the snapshot includes the simulation time and all object values, scheduled events, annotated delays, the contents of the memory allocated for access type values, and file pointers. It does not include aspects of the debugging environment such as breakpoints, probes, Tcl variables, and GUI configuration. PLI/VPI callbacks and handles are saved under certain circumstances. Please refer to the PLI/VPI manuals for details.

You cannot save a snapshot if the simulator is in the process of executing sequential HDL code. If the simulation is in a state that cannot be saved, you must use the `run -clean` command to run the simulation until the currently running sequential behavior (if any) suspends itself at a delay, event control, or a VHDL `wait` statement.

 The Tcl `save` command cannot save the current simulation state into a snapshot if the specified design contains one of the following models:

- An ARM Design Simulation Model (DSM)
- An AMP Model

Your operating system may impose a two gigabyte limit on the size of a file. If a library database exceeds this limit, you cannot add objects to the database. If you save many snapshot checkpoints to unique views in a single library, this file size limit could be exceeded.

If you reach the two gigabyte limit, you can:

- Use `save -overwrite` to overwrite an existing snapshot. For example:

```
xcelium> save -simulation -overwrite snap1
```

- Save snapshots to a separate library. For example:

```
% mkdir xcelium.d/snaplib
% xmsim -f xmsim.args
xcelium> run 1000 ns
xcelium> save -simulation snaplib.snap1
xcelium> run 1000 ns
xcelium> save -simulation snaplib.snap2
```

- Remove snapshots using the `ncrm` utility. For example:

```
% ncrm -snapshot worklib.snap1
```

save Command Syntax

```
save -simulation <snapshot_name>
    [-checkpoint]
    [-overwrite]
    [-path path]
    [-postprocess]
    [-snwithlogs]
    -environment [filename]
    -commands [filename]
```

save Command Options

This section describes the options that you can use with the Tcl `save` command.

| | |
|--|--|
| <code>-simulation snapshot_name</code> | <p>Creates a snapshot of the current simulation state. This option is the default.</p> <p>You must specify a snapshot name with the <code>save</code> command. The snapshot name can be specified using the <code>[lib.]cell[:view]</code> notation, or, if you want the snapshot to be a new view of the currently loaded cell, you can specify just the view name preceded by a colon. For example, if you are simulating <code>worklib.top:rtl</code>,</p> <ul style="list-style-type: none"> <code>save ckpt1</code> Saves <code>worklib.ckpt1:rtl</code> <code>save top:ckpt1</code> Saves <code>worklib.top:ckpt1</code> <code>save otherlib.top</code> Saves <code>otherlib.top:rtl</code> <code>save :ckpt1</code> Saves <code>worklib.top:ckpt1</code> <p>The snapshot name must be a simple name containing only letters, numbers, and underscores.</p> <p>By default, the saved snapshot is saved either in the library you specify or in the work library (if you do not specify a library). You can use the <code>-path</code> option to specify a different location for the saved snapshot.</p> |
| <code>-checkpoint</code> | Saves a checkpoint for the analog circuit. |
| <code>-overwrite</code> | Overwrites an existing snapshot. |
| <code>-path path</code> | <p>Saves the simulation snapshot at the specified location.</p> <p>By default, the saved snapshot is saved either in the library you specify or in the work library (if you do not specify a library). The <code>-path</code> option lets you specify a different location for the saved snapshot. The content of the library where the snapshot needs to be saved is first copied to the location you specify, and then the new snapshot is written to that library.</p> <p>The directory in which the snapshot is to be saved must exist.</p> <p>The following command saves a snapshot called <code>top:ckpt1</code> in library <code>worklib</code> at the location specified with the <code>-path</code> option:</p> <pre>xcelium> save -path /path/to/new/snapshot/dir worklib.top:ckpt1</pre> <p>Use the <code>restart -path</code> command to restart a simulation with a snapshot that was saved in a location specified with <code>save -path</code>.</p> |
| <code>-postprocess</code> | Writes out a non-simulatable snapshot of a design containing SystemC when specified with the <code>save -simulation</code> command. The option has no effect on a design that does not contain SystemC. If used with a <code>save</code> command without provided or implied <code>-simulation</code> , the option is ignored with a warning. |
| <code>-snwithlogs</code> | <p>Saves in the <code>esv</code> file the contents of the main Specman log file and all message loggers files. Use this option when you want to be able to append this log to the log file for subsequent simulations.</p> <p>For more information, see Creating a Single Log File for Multiple Simulations in <i>Running Specman with Xcelium Simulator</i>.</p> |

`-environment`
`[filename]`

Creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. The *filename* argument is optional. If no filename is specified, the script is written to standard output.

The state of the Tcl debug environment is not part of the simulation that is saved in a snapshot. To save the debug environment, you must issue a separate `save -environment` command. (Note that the `save -commands` command is the same as `save -environment`.) This command creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. You can then restore the environment by executing this script with the Tcl `source` command, or you can use the `-input` option when you invoke the simulator.

For example:

```
xcelium> save :ckpt1
xcelium> save -environment ckpt1.tcl
xcelium> restart :ckpt1
xcelium> source ckpt1.tcl
(or: % xmsim -tcl cell:ckpt1 -input ckpt1.tcl)
```

These scripts are meant to be sourced into an empty environment (that is, an environment with no breakpoints, no probes, and no databases). If you invoke the simulator, set some breakpoints and probes, and then source a script that contains commands to set breakpoints and probes, the simulator generates error(s) indicating that some commands in the script could not be executed. These errors are due to name conflicts. For example, you may have set a breakpoint that received the default name "1", and the command in the script is trying to create a breakpoint with the same name. You can, of course, give your breakpoints unique names to avoid this problem. You can also edit the scripts to make them work the way you would like them to work.

`-commands [filename]` Similar to the `save -environment`.

save Command Examples

The following command saves the simulation state in `lib.cell:ckpt1`, where *lib* is the name of the current work library, and *cell* is the cell name of the currently loaded snapshot:

```
xcelium> save -simulation :ckpt1
```

The following command saves the simulation state in `lib.top:ckpt1`:

```
xcelium> save top:ckpt1
```

The following command saves the simulation state in `lib.ckpt1:view_name`, where *view_name* is the view name that is currently being simulated:

```
xcelium> save ckpt1
```

The following example illustrates how to use the `save`, `restart`, and `reset` commands:

```
% xmsim -nocopyright -input run.vc hardrive
Loading snapshot worklib.hardrive:module ..... Done

xcelium> database -open waves -default -shm ;# Open default SHM
```



```
                ;# dbase called waves.
```

Created default SHM database waves

```
xcelium> probe -create -all -database waves    ;# Probe all signals
                ;# in current scope.
```

Created probe 1

```
;# Create an object breakpoint and a time breakpoint at absolute time 200.
```

```
xcelium> stop -create -object clr
```

Created stop 1

```
xcelium> stop -create -time -absolute 200 ns
```

Created stop 2

```
xcelium> run
```

0 FS + 0 (stop 1: hardrive.clr = 1)

```
./hardrive.v:12    clr = 1;
```

```
xcelium> run
```

200 NS + 0 (stop 2)

```
xcelium> save :ckpt1                ;# Save the simulation state at 200 ns.
```

Saved snapshot worklib.hardrive:ckpt1

```
xcelium> stop -create -time -relative 300 ns    ;# Create a breakpoint to stop
                ;# every 300 ns.
```

Created stop 3

```
xcelium> run
```

500 NS + 0 (stop 3)

```
xcelium> save :ckpt2                ;# Save another snapshot at time 500 ns.
```

Saved snapshot worklib.hardrive:ckpt2

```
xcelium> run
```

800 NS + 0 (stop 3)

```
xcelium> restart :ckpt1            ;# Use the restart command to load
                ;# worklib.hardrive.ckpt1.
```

Loaded snapshot worklib.hardrive:ckpt1

```
xcelium> time
```

```
200 NS                ;# Simulation is at 200 ns. Two breakpoints are still set
                ;# (the breakpoint set for absolute time 200 was deleted
                ;# automatically when it triggered). The SHM database has
                ;# been closed and all probes deleted.
```

```
xcelium> stop -show
```

```
1      Enabled Object hardrive.clr
```

```
3      Enabled Time 500 NS (every 300 NS)
```

```
xcelium> database -show
```

```
waves   Enabled          (file: waves.shm) (SHM) (default)
```

```
xcelium> run
```

500 NS + 0 (stop 3)

```
xcelium> reset                ;# Use the reset command to reset the model
                ;# to its original state at time zero.
                ;# Notice that both breakpoints are still set.
```

Loaded snapshot worklib.hardrive:module

```
xcelium> time
```

```
0 FS
xcelium> stop -show
1      Enabled      Object hardrive.clr
3      Enabled      Time 200 NS (every 300 NS)
xcelium>
```

The following sequence of commands shows how you can use the `-path` option to save a snapshot to a specified location:

```
% mkdir /path/to/new/snapshot/dir
% xmsim -nocopyright -tcl worklib.hardrive
xcelium> stop -create -time -absolute 200 ns
Created stop 1
xcelium> run
200 NS + 0 (stop 1)
;# The following command creates a new snapshot at the location specified
;# by the -path option.
xcelium> save -path /path/to/new/snapshot/dir worklib.hardrive:ckpt1
Saved snapshot worklib.hardrive:ckpt1
xcelium> run 200ns
Ran until 400 NS + 0
xcelium> restart -path /path/to/new/snapshot/dir worklib.hardrive:ckpt1
Loaded snapshot worklib.hardrive:ckpt1
xcelium> time
200 NS
```

The following example illustrates how to use the `save -environment` command:

```
% xmsim -nocopyright -tcl hardrive
Loading snapshot worklib.hardrive:module ..... Done
;# Set a line breakpoint, an object breakpoint, and create a probe.
;# The probe command creates a default SHM database.
xcelium> stop -create -line 32
Created stop 1
xcelium> stop -create -object hardrive.clk
Created stop 2
xcelium> probe -create -shm hardrive.data
Created default SHM database xcelium.shm
Created probe 1
xcelium> run
0 FS + 0 (stop 2: hardrive.clk = 0)
./hardrive.v:13      clk = 0;
xcelium> run
50 NS + 0 (stop 2: hardrive.clk = 1)
./hardrive.v:16 always #50 clk = ~clk;
xcelium> save -environment env1.env      ;# Save debug settings in a file
                                         ;# called env1.env.
```

```
xcelium> more env1.env                                ;# The file env1.env contains
                                                    ;# commands to recreate debug settings.

set assert_report_level {note}
set assert_stop_level {error}
set autoscope {yes}
set display_unit {auto}
set tcl_prompt1 {puts -nonewline "xcelium> "}
set tcl_prompt2 {puts -nonewline "> "}
set time_unit {module}
set vlog_format {%h}
set assert_1164_warnings {yes}
stop -create -name 1 -line 32 harddrive
stop -create -name 2 -object harddrive.clk
database -open -shm -into xcelium.shm xcelium.shm -default
probe -create -name 1 -database xcelium.shm harddrive.data
scope -set harddrive
xcelium> exit                                          ;# Exit and then reinvoke the simulator.
% xmsim -nocopyright -tcl harddrive
Loading snapshot worklib.harddrive:module ..... Done
xcelium> stop -show
No stops set
xcelium> source env1.env                            ;# Source the script env1.env.
xcelium>
;# Show the status of breakpoints, probes, and databases.
xcelium> stop -show
1      Enabled      Line: ./harddrive.v:32 (scope: harddrive)
2      Enabled      Object harddrive.clk
xcelium> probe -show
1      Enabled      harddrive.data (database: xcelium.shm) -shm
xcelium> database -show
xcelium.shm      Enabled      (file: xcelium.shm) (SHM) (default)
```

Related Topics

- [Saving SystemC Simulations](#)
- [Restarting SystemC Simulations](#)

scope

The Tcl `scope` command enables you:

- Set the current debug scope (`-set`).
- Describe items declared within a scope (`-describe`).

- Display the drivers of objects declared within a scope (`-drivers`).
- Print the source code, or part of the source code, for a scope (`-list`).
- Display scope information (`-show`).

The `scope` command can be applied to SystemC scopes; however, it is restricted to objects derived from `sc_module` and SystemC processes.

scope Command Syntax

```
scope [<scope_name>] [-set] [-fullpath]
    -derived
    -running
    -super
    -up
scope
    -aicms [<scope_name>] [-all] [-recurse]
    -describe [<scope_name>] [-names] [-sort {name | kind | declaration}]
    -disciplines [<scope_name>] [-all] [-recurse] [-sort {name | kind | declaration}]
    -drivers [<scope_name>]
    -history
    -list [line | start_line end_line] [scope_name]
    -sc_processes [-all] [-recurse] [-verbose]
    -show
    -tops
```

scope Command Options

This section describes the options that you can use with the Tcl `scope` command.

`-set` [*scope_name*] Sets the current debug scope to the specified scope. If no scope or other option is given, the name of the current scope is printed.

The `-set` modifier is optional.

You can use the following options with `-set`:

- **-derived:** Sets the debug scope to be the derived class of the current class instance scope.

SystemVerilog allows classes to extend classes. In a complex design using extensive object-oriented techniques, this class inheritance can sometimes make it difficult to interpret data when debugging class objects.

Use the `scope -derived` command to set the current debug scope to the derived class of the current class instance scope. The `scope -super` command lets you set the current debug scope to the super or base class of the current class instance scope.

- **-fullpath**: Treats the given path as the full hierarchical path.
- **-running**: Sets the debug scope to the currently running process.
- **-super**: Sets the debug scope to be the super or base class of the current class instance scope.

SystemVerilog allows classes to extend classes. In a complex design using extensive object-oriented techniques, this class inheritance can sometimes make it difficult to interpret data when debugging class objects.

Use the `scope -super` command to set the current debug scope to the super or base class of the current class instance scope. The `scope -derived` command lets you set the current debug scope to the derived class of the current class instance scope.

- **-up**: Sets the debug scope to one level up the hierarchy from the current scope.

```
-aicms [-all] [-recurse]
[scope_name]
```

Lists auto-inserted connect modules (AICMs) inserted within the specified scope, or within the current debug scope if no scope is specified.

The `-recurse` option descends recursively through the design hierarchy, starting with the specified scope (or the current debug scope if no scope is specified), listing all AICM instances.

The `-all` option lists the AICM instances in all top-level scopes. If used with `-recurse`, the `-all` option recursively lists all AICM instances in the entire design.

```
-describe [-names]
[-sort {name | kind
| declaration}]
[scope_name]
```

Describes all objects declared within the specified scope. If no scope is specified, objects in the current debug scope are described.

The kind of access that has been enabled for simulation objects is shown in the output. For example, the string `(-WC)` is included in the output for objects that have read access but no write or connectivity access. See [Access to Simulation Objects](#) for details on specifying access to simulation objects.

For objects without read access, the output of `scope -describe` does not include the object's value.

Use the `-names` option if you want to display only the names of each declared item in the scope.

Use the `-sort` option if you want to specify a sort order. There are three possible arguments to the `-sort` option:

- **name**: Sorts alphabetically by name.
- **kind**: Sorts by declaration type (reg, wire, instance, process, and so on).
- **declaration**: Sorts by the order in which objects are declared in the source code.

| | |
|--|---|
| <pre>-disciplines [-all] [-recurse] [-sort {name kind declaration}] [scope_name]</pre> | <p>Lists all resolved net disciplines within the given scope, or within the current debug scope if no scope is given.</p> <ul style="list-style-type: none"> • The <code>-recurse</code> option descends recursively through the design hierarchy, starting with the specified scope (or the current debug scope if no scope is specified), listing all resolved net disciplines. • The <code>-all</code> option lists all resolved net disciplines in all top-level scopes. If used with <code>-recurse</code>, it recursively lists all resolved net disciplines in the entire design. • The <code>-sort</code> option can be used to sort the nets alphabetically by net name, by discipline (electrical, logic, and so on), or by the order they are declared in the source code. The default is to sort by discipline. |
| <pre>-drivers [scope_name]</pre> | <p>Shows the drivers of each object declared within the specified scope. If no scope is specified, the drivers of objects in the current debug scope are displayed.</p> <p>The output of <code>scope -drivers</code> includes only the objects that have read access. However, even if an object has read access, its drivers may have been collapsed, combined, or optimized away, and the output of the command might indicate that the object has no drivers. See Access to Simulation Objects for details on specifying access to simulation objects.</p> |
| <pre>-fullpath</pre> | <p>Treats the specified path as the full hierarchical path to the scope.</p> |
| <pre>-history</pre> | <p>Lists all scopes in the order in which they were visited.</p> <p>As you change scopes during a debug session, each scope that you visit is captured in a list. Use the <code>scope -history</code> command to display this list.</p> <p>In the scope history list, the current debug scope is marked with an asterisk (*). You can use the <code>scope -back</code> command to change the debug scope to the scope that is shown immediately before the current debug scope. Use the <code>scope -forward</code> command to change the debug scope to the scope that is shown immediately after the current debug scope.</p> <p>During the course of a simulation run, dynamic scopes (class instances and methods) may become invalid because the dynamic object no longer exists. Scopes that are no longer valid are removed from the scope history list.</p> |
| <pre>-list [line start_line end_line] [scope_name]</pre> | <p>Prints lines of source code for the specified scope, or for the current debug scope if no scope is specified.</p> <p>You can follow the <code>-list</code> modifier with:</p> <ul style="list-style-type: none"> • No range of lines to print all lines for the scope. • One line number to display that line of the source text. • Two line numbers to display the text between those two line numbers. You can use a dash (-) for either the <code>start_line</code> or the <code>end_line</code>. |

| | |
|--|--|
| <code>-sc_processes [-all] [-recurse] [-verbose] [scope_name]</code> | <p>Lists SystemC processes in the specified scope, or in the current debug scope if no scope is specified.</p> <ul style="list-style-type: none"> • The <code>-verbose</code> option reports all properties of the processes. • The <code>-recurse</code> option descends recursively through the design hierarchy, starting with the specified scope (or the current debug scope if no scope is specified), and listing all of the processes. • The <code>-all</code> option lists the processes in all top-level scopes. If used with <code>-recurse</code>, it recursively lists all SystemC processes in the entire design. |
| <code>-show</code> | <p>Shows scope information, including the current debug scope, instances within the debug scope, and top-level modules in the currently loaded model.</p> <p>If the current debug scope is a subprogram on a VHDL call stack, the output of the <code>scope -show</code> command or of the <code>scope</code> command (with no options or arguments) shows the current scope as <code>:process_scope [nest_level]</code>. For example, if the current scope is a subprogram called <code>function1</code>, which is at nest-level 1, the output of these commands shows the current scope as:</p> <pre>:process1[1]</pre> |
| <code>-tops</code> | <p>Displays a list of the names of the top-level modules in the design, including VHDL packages. For example, the following command shows the output for a VHDL model with two packages and the one top-level scope (:) :</p> <pre>xcelium>> scope -tops @std.STANDARD @ieee.std_logic_1164 :</pre> <p>The following command shows the output for a Verilog model with two top-level modules:</p> <pre>xcelium> scope -tops top initialize</pre> <p>The <code>-tops</code> modifier is useful when you want to execute a Tcl command that takes a list of scopes as arguments, and you want to execute the command on all top-level scopes. For example, you can use the following command to probe all objects in the entire design:</p> <pre>xcelium> probe -shm -all [scope -tops] -depth all</pre> |

scope Command Examples

The following example prints the name of the current scope:

```
xcelium> scope
```

The following example sets the debug scope to scope `u1`. The `-set` modifier is not required:

```
xcelium> scope -set u1
```

The following example moves the debug scope up one level in the hierarchy:

```
xcelium> scope -up
```

In the following VHDL example, the `stack -show` command displays the current call stack. The process

(`process1`) is displayed as nest-level 0, the base of the stack. The subprogram `function1` is `:process1[1]`, and the subprogram `function2` is `:process1[2]`.

- The first `scope` command displays the current debug scope.
- The second `scope` command sets the debug scope to `:process1`.
- The third `scope` command sets the debug scope to the subprogram `:process1[2]` (that is, to `function2`).

```
xcelium> stop -subprogram function1
Created stop 1
xcelium> run
0 FS + 0 (stop 1: Subprogram :function1)
./test.vhd:36 tmp4_local := function2 (tmp4);
xcelium> run -step
./test.vhd:29 tmp5_local := tmp5 + 1;
xcelium> stack -show                # Display the current call stack
2: Scope: :process1[2] Subprogram:@work.e(a):function2
   File: /usr1/belanger/xm/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 29

1: Scope: :process1[1] Subprogram:@work.e(a):function1
   File: /usr1/belanger/xm/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 36

0: Scope: :process1
   File: /usr1/belanger/xm/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 52
xcelium> scope -show                # Display the current debug scope
Directory of scopes at current scope level:

Current scope is (:process1[2])
Highest level modules:
  Top level VHDL design unit:
    entity (e:a)
  VHDL Package:
    STANDARD
    ATTRIBUTES
    std_logic_1164
    TEXTIO
xcelium> scope -set :process1        # Set scope to :process1
xcelium> scope -set :process1[2]    # Set scope to :process1[2]
                                     # i.e., function2
```

In the previous example, you can also use the `stack` command to set the debug scope to `:process1[2]` (that is, `function2`).


```
xcelium> scope -set :process1
xcelium> stack -set 2
```

The following SystemVerilog example is used to illustrate the `scope -super` and `scope -derived` commands. This example also illustrates the `-back`, `-forward`, and `-history` options, which are language-independent:

```
package pack;
  class A;
    int aVal;
  endclass
endpackage

module top;
  class B extends pack::A;
    task bTask();
      int value;
      $display( value );
    endtask
  endclass

  B b = new;
endmodule
```

```
xcelium> run 1 ns
Ran until 1 NS + 0
xcelium> scope
top
xcelium> # Scope into class object
xcelium> scope top.b
xcelium> # Scope into the task
xcelium> scope bTask
xcelium> scope
top.B@1_1.bTask
xcelium> # Attempt to scope to base class
xcelium> scope -super
xmsim: *E,SCPSP2: Current debug scope is not a class instance : top.B::bTask.
xcelium> # scope one level up
xcelium> scope -up
xcelium> scope
top.B@1_1
xcelium> # Scope to base class
xcelium> scope -super
xcelium> scope
pack::A@1_1
xcelium> # Scope back to the derived class
```

```
xcelium> scope -derived
xcelium> scope
top.B@1_1
xcelium> # View list of scopes in order in which they were visited
xcelium> scope -history
    1) top
    2) top.B@1_1
    3) bTask
    4) top.B@1_1
    5) pack::A@1_1
*   6) top.B@1_1
    Where: * => current debug scope
xcelium> # Scope back to previous scope in the list
xcelium> scope -back
xcelium> scope
pack::A@1_1
xcelium> scope -back
xcelium> scope
top.B@1_1
xcelium> # Scope forward
xcelium> scope -forward
xcelium> scope
pack::A@1_1
xcelium>
```

The following command displays a list and a description of all objects declared in the current debug scope (a Verilog module):

```
xcelium> scope -describe
clr.....variable logic = 1'hx
clk.....variable logic = 1'hx
data.....variable logic [3:0] = 4'hx
q.....net logic [3:0]
    q[3] (wire/tri) = StX
    q[2] (wire/tri) = StX
    q[1] (wire/tri) = StX
    q[0] (wire/tri) = StX
end_first_pass...named event
hl.....instance of module hardreg
```

The following command displays a list and a description of all objects declared in the current debug scope (a VHDL architecture):

```
xcelium> scope -describe
top.....component instantiation
load_nickels.....process statement
load_dimes.....process statement
```

```
load_cans.....process statement
load_action.....process statement
gen_clk.....process statement
gen_reset.....process statement
gen_nickels.....process statement
gen_dimes.....process statement
gen_quarters.....process statement
$PROCESS_000.....process statement
$PROCESS_001.....process statement
stoppit.....signal : BOOLEAN = TRUE
t_NICKEL_OUT.....signal : std_logic = '0'
t_EMPTY.....signal : std_logic = '1'
t_EXACT_CHANGE...signal : std_logic = '0'
t_TWO_DIME_OUT...signal : std_logic = 'Z'
...
...
t_NICKELS.....signal : std_logic_vector(7 downto 0) = "11111111"
t_RESET.....signal : std_logic = '0'
```

The following command lists the names of all objects declared in the current debug scope. No description is included:

```
xcelium> scope -describe -names
clr clk data q end_first_pass h1
```

The following example displays a list and a description of all objects declared in the current debug scope. Objects are listed in alphabetical order:

```
xcelium> scope -describe -sort name
```

The following command displays a list and a description of all objects declared in the current debug scope. Objects are sorted by type of declaration:

```
xcelium> scope -describe -sort kind
```

The following command displays a list and a description of all objects declared in scope `u1`. Objects are listed in the order in which they were declared in the source code:

```
xcelium> scope -describe -sort declaration u1
```

The following example shows a Verilog module that contains nested generate-loops. Each for-generate instance is a scope. You can set the debug scope to a for-generate instance or describe a for-generate instance with `scope -describe`. See [Generate Constructs](#) for details on generated instantiations:

```

module top;
  generate
    genvar i;
    for (i = 5; i < 9; i = i + 1) begin : g1
      genvar j;
      for (j = i; j >= 1; j = j - 1) begin : g2
        reg [0:i] r;
        initial begin
          r = i;
          $display("%b", r);
        end
      end
    end
  endgenerate
endmodule

```

```

xcelium> scope -describe top
i.....genvar
g1.....for-generate statement [5 to 8]
xcelium> scope g1[6]
xcelium> scope -describe
i.....generate parameter = 6
j.....genvar
g2.....for-generate statement [6 to 1, step -1]
xcelium> scope -describe g2[5]
j.....generate parameter = 5
r.....variable logic [0:6] = 7'hxx

```

The following command shows the drivers for all objects declared in scope `h1`:

```

xcelium> scope -drivers h1
clk.....input net (wire/tri) logic = St1
      St1 <- (hardrive.h1) input port 2, bit 0 (./hardrive.v:8)
clrb.....input net (wire/tri) logic = St1
      St1 <- (hardrive.h1) input port 3, bit 0 (./hardrive.v:8)
d.....input net logic [3:0]
  d[3] (wire/tri) = St1
      St1 <- (hardrive.h1) input port 1, bit 3 (./hardrive.v:8)
  d[2] (wire/tri) = St0
      St0 <- (hardrive.h1) input port 1, bit 2 (./hardrive.v:8)
  d[1] (wire/tri) = St0
      St0 <- (hardrive.h1) input port 1, bit 1 (./hardrive.v:8)
  d[0] (wire/tri) = St1
      St1 <- (hardrive.h1) input port 1, bit 0 (./hardrive.v:8)
q.....output net logic [3:0]
  q[3] (wire/tri) = St1

```

```
St1 <- (hardrive.h1.f4) nd7 (q, e, qb)
q[2] (wire/tri) = St0
St0 <- (hardrive.h1.f3) nd7 (q, e, qb)
q[1] (wire/tri) = St0
St0 <- (hardrive.h1.f2) nd7 (q, e, qb)
q[0] (wire/tri) = St0
St0 <- (hardrive.h1.f1) nd7 (q, e, qb)
```

The following example lists the source for the current debug scope:

```
xcelium> scope -list
```

The following example lists the source for scope `u1`:

```
xcelium> scope -list u1
```

The following example displays line 12 of the source for the current debug scope:

```
xcelium> scope -list 12
```

The following example lists lines 10 through 15 of the source for the current debug scope:

```
xcelium> scope -list 10 15
```

The following command lists lines from the top of the module through line 10 of the source for the current debug scope:

```
xcelium> scope -list - 10
```

The following command lists lines of source for the current debug scope, beginning with line 30:

```
xcelium> scope -list 30 -
```

The following command shows the output of the `scope -describe` command when you run in regression mode (no read, write, or connectivity access to simulation objects) and some objects do not have read or write access:

```
xcelium> scope -describe h1
clk.....input net logic (-RWC)
clrb.....input net logic (-RWC)
d.....input net logic [3:0]
  d[3] (-RWC)
  d[2] (-RWC)
  d[1] (-RWC)
  d[0] (-RWC)
q.....output net logic [3:0]
  q[3] (-RWC)
  q[2] (-RWC)
  q[1] (-RWC)
  q[0] (-RWC)
f1.....instance of module flop
f2.....instance of module flop
```

```
f3.....instance of module flop  
f4.....instance of module flop
```

Related Topics

- [Hierarchy Separators in Tcl Commands](#)
- [Tcl Commands for Debugging Processes](#)
- [Tcl-Based Debugging](#)

scverbosity

The Tcl `scverbosity` command sets SystemC message verbosity level interactively.

scverbosity Command Syntax

```
scverbosity -get |-set [-recursive] <verbosity_level> [<instance-name>]
```

scverbosity Command Options

This section describes the options that you can use with the Tcl `scverbosity` command.

| | |
|--|--|
| <code>-get</code> | Displays the current maximum verbosity level of the SystemC reporting mechanism. |
| <code>-set [-recursive]</code> <code><verbosity_level> [<instance-name>]</code> | Sets the maximum verbosity level of the SystemC reporting mechanism to the value specified in <i>verbosity_level</i> . If a module's hierarchical pathname is not specified, the setting applies to the maximum global verbosity. If the <code>-recursive</code> option is used, it also applies to the maximum verbosity of the entire module hierarchy. If a module's pathname is specified, the maximum verbosity setting applies to that module, and also to the hierarchy rooted at that module if the <code>-recursive</code> option is used. |

scverbosity Command Examples

To output the global maximum verbosity setting of the SystemC reporting mechanism if a module's hierarchical path name is not specified:

```
xcelium> scverbosity -get  
SystemC Global Reporting Verbosity is SC_LOW (100)
```

otherwise, it outputs the maximum verbosity setting of the specified module:

```
xcelium> sc_verbosity -get top.mod1  
SystemC Reporting Verbosity of "top.mod1" is SC_HIGH (300)
```

To assign the maximum verbosity.

- If a module's hierarchical pathname is not specified, the setting applies to the maximum global verbosity.

```
xcelium> scverbosity -set 100  
Setting SystemC Global Reporting Verbosity to 150
```

- If the `-recursive` option is used, it also applies to the maximum verbosity of the entire module hierarchy.

```
xcelium> scverbosity -set -recursive 150  
Setting SystemC Global and Hierarchical Reporting Verbosity to 150
```

- If a module's pathname is specified, the maximum verbosity setting applies to that module, and also to the hierarchy rooted at that module if the `-recursive` option is used.

```
xcelium> scverbosity -set 150 "top.mod1"  
Setting SystemC Reporting Verbosity of "top.mod1" to 150  
  
xcelium> scverbosity -set -recursive 150 "top.mod1"  
Setting SystemC Hierarchical Reporting Verbosity of "top.mod1" to 150
```

- Any of the `sc_verbosity` enumeration strings, corresponding to the predefined verbosity levels, as specified in [Enumeration of Verbosity Levels](#). For example:

```
xmsim -scverbosity SC_DEBUG sc_main
```

- A positive integer value. For example:

```
xmsim -scverbosity 500 sc_main
```

- A negative integer value prefixed by the character `D` or `d`, representing a decimal number. (Without this prefix, the simulator interprets the negative number as another command option, and reports an error.) For example:

```
xmsim -scverbosity D-100 sc_main  
xmsim -scverbosity d-100 sc_main
```

Related Topic

- [Controlling Verbosity of the SystemC Reporting Mechanism](#)

simtfile

The Tcl `simtfile` command is used to add or modify delays for ports using the specified timing file.

This command reads in a timing file at run time, which enables you to add or change port delays using the

`-simport` timing specification without re-elaborating the design. You must first mark (or define) the ports at elaboration time. If a delay is specified on a port during simulation that was not defined during elaboration, then the software issues an error.

simtfile Command Syntax

```
simtfile <filename>
```

simtfile Command Options

None.

simtfile Command Examples

```
xcelium> simtfile sim.tfile
```

This command specifies a file `sim.tfile` with modified port delays to run during simulation.

Related Topics

- [Modifying Delays at Ports](#)
- [Timing Control in Selected Portions of a Design](#)

simvision

The Tcl `simvision` command allows you to:

- Invoke SimVision and connect to the current simulation (`simvision` command with no options).
- Send a script containing SimVision commands to SimVision (`simvision -input`). If you are not currently in SimVision, this command invokes the GUI and runs the commands in the script.
- Send a SimVision command (or multiple commands) to SimVision (`simvision -submit`). If you are not currently in SimVision, this command invokes the GUI and runs the command(s).

simvision Command Syntax

```
simvision
  -input script_file
  -layout name
  -simvisargs <string>
  [-submit] simvision_command
```

simvision Command Options

This section describes the options that you can use with the Tcl `simvision` command.

| | |
|---|--|
| <code>-input script_file</code> | <p>Runs the SimVision commands in the specified script.</p> <ul style="list-style-type: none">• If you are not currently in the SimVision environment, a <code>simvision -input</code> command invokes SimVision and then runs the commands in the script.• If you are in the SimVision environment, entering a <code>simvision -input</code> command at the <code>xcelium></code> prompt in the Console window is the same as using the <i>File - Source Command Script</i> command to source a script. |
| <code>-layout name</code> | <p>Launches SimVision with a built-in layout.</p> <p>The <code>-layout cdebug</code> option arranges the windows with the <i>CDebug</i> layout, and launches the GUI if needed. You can issue the command any time at the prompt or in an initial Tcl input script. If SimVision is already connected, the windows are re-arranged to the built-in <i>CDebug</i> layout.</p> <p>Using the <code>-layout</code> option to the <code>simvision</code> command is equivalent to using the following command:</p> <pre>xcelium> simvision -submit window layout choose name</pre> |
| <code>-simvisargs <string></code> | <p>Invokes SimVision with command-line arguments.</p> |
| <code>-submit simvision_command</code> | <p>Runs the specified SimVision command.</p> <p>If you are not currently in the SimVision environment, a <code>simvision -submit</code> command invokes SimVision and then runs the command.</p> <p>You can specify multiple commands by enclosing the commands in curly braces.</p> <p>The <code>-submit</code> option is not required.</p> |

simvision Command Examples

The following command invokes SimVision and connects to the current simulation:

```
xcelium> simvision
```

The following command runs the SimVision commands in the file called `simvision.sv`:

```
xcelium> simvision -input simvision.sv
```

The following command runs a SimVision command to create a new waveform window. The `submit` option can be omitted:

```
xcelium> simvision -submit waveform new
```

The following command runs two SimVision `waveform` commands. The first command creates a new waveform window called `MyWaves`. The second command adds the signals `board.count` and `board.af` to the new waveform window:

```
xcelium> simvision {  
> waveform new -name "MyWaves"  
> waveform add -signals {board.count board.af}  
> }
```

Related Topic

- [Using SimVision Tcl Commands](#)

sn

The Tcl `sn` command is used to pass commands from the simulator to Specman.

sn Command Syntax

```
sn [specman_command[\; specman_command ...]]
```

sn Command Options

None.

sn Command Examples

Use the `sn` command to pass a single command to Specman by including the Specman command. For example:

```
xcelium> sn load test_1
```

You can also issue multiple commands on the same command line by separating each `sn` command with a semicolon. For example:

```
xcelium> sn load test_1; sn test
```

This is the same as:

```
xcelium> sn load test_1
xcelium> sn test
```

Xcelium also supports issuing multiple commands from *xmsim* to Specman with one `sn` command. In this case, the Specman commands are separated with a semicolon, and the semicolon must be escaped. For example:

```
xcelium> sn config gen -seed=125645\; test
```

Related Topics

- [Specman Command Reference](#)

source

The Tcl `source` command lets you execute Tcl commands contained in a script file.

When *xmsim* has processed all the commands in the source file, or if you interrupt processing with CTRL/C, input reverts to the terminal.

You can also execute Tcl commands in a script file by using the `input` command or by using the `-input` command-line option when you invoke the simulator. However, the behavior of the `source` command differs from the behavior of the `input` command or the `-input` option in the following ways:

- With the `source` command, execution of the commands in the script stops if a command generates an error. With the `input` command, the contents of the file are read in place of standard input at the next Tcl prompt, as if you had typed the commands at the command-line prompt. This means that the execution of commands in the script **do not stop due to an error**.
- The `input` command echoes commands to the screen as they are executed, along with any command output or error messages. On the other hand, the `source` command displays the output of only the last command in the file. Output from the model (for example, the output of `$display`, `$monitor`, or `$strobe` tasks, or the output of stop points) is printed on the screen.

source Command Syntax

```
source script_file
```

You can specify only one *script_file*.

source Command Options

None.

source Command Examples

The following example uses the command file named `commands.inp` to show how the `source` command displays output from only the last command in the file. The `commands.inp` file contains the following commands:

```
stop -create -line 27
run
value data
run 50
value data
run 50
value data
run
```

```
%> xmsim -nocopyright -tcl hardrive
```

```
xcelium> source commands.inp
```

```
0 FS + 0 (stop 1) 27: repeat (2)           ;# Output of the stop command
at time 50 clr =1 data= 0 q= x              ;# Output of $strobe task in model
at time 150 clr =1 data= 1 q= 0
at time 250 clr =1 data= 2 q= 1
...
at time 3150 clr =0 data=14 q= 0
at time 3250 clr =0 data=15 q= 0
at time 3350 clr =0 data=15 q= 0
Simulation complete via $finish(1) at time 3400 NS + 0 ;# Output of last
                                                    ;# command (run)

xcelium>           ;# Control reverts to the terminal after the simulator
                  ;# executes the last command
```

Related Topic

- [-input](#)

stack

The Tcl `stack` command allows you to debug Verilog descriptions that involve multiple task and function calls, and VHDL descriptions that involve multiple subprogram calls. Using this command, you can:

- View the current call stack (`-show`).
- For VHDL, set the current stack frame (`-set`) so that you can describe objects that are local to a subprogram, view object values, and set object values.

The `stack -set` command is not supported for Verilog call stacks.

stack Command Syntax

```
stack  
  [-set] [{stack_level_spec | -down | -up}]  
  [-show] [-levels stack_level_count]
```

stack Command Options

This section describes the options that you can use with the Tcl `stack` command.

```
-set [{  
  stack_level_spec |  
  -down | -up}]
```

The `stack -set` command with no argument displays the call stack for the current debug scope. This is the same as `stack -show`.

You can use the following options:

- ***stack_level_spec***: Set the current debug stack to the specified stack level. This applies only to the current debug scope, which must be a process scope.

The *stack_level_spec* argument specifies the call stack depth. This can be:

- An absolute value, which sets the context to the specified stack frame. For example:
`stack -set 2`
- A relative value, which moves the stack context up or down relative to the current stack frame. For example:
`stack -set +2`
`stack -set -1`

- **`-down`**: Sets the debug stack one level down from the current level.

`stack -set -down` is the same as `stack -set -1`.

- **`-up`**: Sets the debug stack one level up from the current level.

`stack -set -up` is the same as `stack -set +1`.

```
-show [-levels  
stack_level_count]
```

Displays the call stack for the current debug scope.

By default, a `stack -show` command displays the call stack used by the process to reach the current execution point. It stops at the construct that started the process (for example, a Verilog `initial` block or `fork` process, or a VHDL process). Include the `-levels` option to limit the depth of the call stack that is displayed.

Note: For Verilog, the `stack -show` command works only for the currently executing process, and only if the debug scope is set to the scope (usually a task or function) in which it is executing.

stack Command Examples

The section provides some examples of debugging Verilog descriptions and VHDL descriptions using the `stack` command.

- [VHDL Examples](#)
- [Verilog Examples](#)

VHDL Example

The following VHDL source code is used for the example.

```
LIBRARY IEEE;
USE IEEE.Std_logic_1164.all;
USE STD.TEXTIO.ALL;
entity e is
end;

architecture a of e is

    procedure procedure1 (tmp1 : INOUT bit) is
    begin
        tmp1 := not tmp1;
        return;
    end procedure1;

    procedure procedure2 (tmp2 : INOUT bit) is
    begin
        if (tmp2 = '0')
        then
            tmp2 := not tmp2;
            return;
        else
            procedure1(tmp2);
        end if;
    end procedure2;
```

```
function function2 (tmp5 : IN integer) return integer is
    variable tmp5_local : integer := tmp5;
begin
    tmp5_local := tmp5 + 1;
    return tmp5_local;
end function2;

function function1 (tmp4 : IN integer) return integer is
    variable tmp4_local : integer := tmp4;
begin
    tmp4_local := function2 (tmp4);
    return tmp4_local;
end function1;

begin
    process1 : process
        variable var1, var2, var3 : bit := '0';
        variable var4, var5, var6, var7, var8 : integer := 0;
    begin
        var1 := '1';
        procedure1 (var1);
        var2 := '1';
        procedure2 (var2);
        procedure2 (var3);
        var5 := 1;
        var6 := function1 (var5);
        var7 := function2 (var6);
        wait;
    end process;
end;
```

In the following example, the `stack -show` command displays the current call stack. The process (process1) is displayed as nest-level 0, the base of the stack.

After setting the scope to process1, a `stack -set` command sets the current debug scope to nest-level 1 (function1). The `stack -set -up` command sets the debug scope to nest-level 2 (function2).

```
xcelium> stop -subprogram function1
Created stop 1
xcelium> run
0 FS + 0 (stop 1: Subprogram :function1)
./test.vhd:36 tmp4_local := function2 (tmp4);
xcelium> run -step
./test.vhd:29 tmp5_local := tmp5 + 1;
xcelium>
xcelium> stack -show                                # Display the current call stack
```

```
2: Scope: :process1[2] Subprogram:@work.e(a):function2
  File: ./test.vhd
  Line: 29

1: Scope: :process1[1] Subprogram:@work.e(a):function1
  File: ./test.vhd
  Line: 36

0: Scope: :process1
  File: ./test.vhd
  Line: 52
xcelium> stack -show -levels 2           # Display only two levels
2: Scope: :process1[2] Subprogram:@work.e(a):function2
  File: ./test.vhd
  Line: 29

1: Scope: :process1[1] Subprogram:@work.e(a):function1
  File: ./test.vhd
  Line: 36
xcelium>
xcelium> scope -show                     # Display the current scope
Directory of scopes at current scope level:

Current scope is (:process1[2])
Highest level modules:
  Top level VHDL design unit:
    entity (e:a)
  VHDL Package:
    STANDARD
    ATTRIBUTES
    std_logic_1164
    TEXTIO
xcelium> scope -set :process1             # Set scope to process1
xcelium> stack -set 1                     # Set scope to :process1[1]
                                           #(i.e., function1)
xcelium> scope -show
Directory of scopes at current scope level:

Current scope is (:process1[1])
...
...
xcelium> stack -show                     # Show stack for function1
1: Scope: :process1[1] Subprogram:@worklib.e(a):function1
  File: ./test.vhd
  Line: 36
```



```
0: Scope: :process1
  File: ./test.vhd
  Line: 52
xcelium> stack -set -up                # Set scope up one level, to :process1[2]
                                           # This is the same as stack -set +1

xcelium> scope -show
Directory of scopes at current scope level:

Current scope is (:process1[2])
...
...
xcelium> value tmp5                    # Display the value of :process1[2]:tmp5
1
xcelium> value :process1[1]:tmp4      # Display the value of :process1[1]:tmp4
1
```

Verilog Example

The following Verilog source code is used in the example.

```
module top;
  task error;
    begin
      $display("Error at");
      $stop;
    end
  endtask

  task t;
    input integer i;
    if (i == 0)
      error;
  endtask

  initial
    begin
      t(1);
      t(0);
    end
endmodule
```

In the following example, the `stack -show` command displays the current call stack. The `initial` block, the base of the stack, is displayed as nest-level 2.

```
xcelium> run
Error at
Simulation stopped via $stop(1) at time 0 FS + 0
./test.v:7      $stop;
xcelium> stack -show
0: task top.error at ./test.v:7
1: task top.t at ./test.v:14
2: initial block in top at ./test.v:20
xcelium>
xcelium> stack -show -levels 2
0: task top.error at ./test.v:7
1: task top.t at ./test.v:14
xcelium>
xcelium> scope -show
Directory of scopes at current scope level:

Current scope is (top.error)
Highest level modules:
top
```

```
xcelium> scope top.t           # Set scope to top.t
xcelium> stack -show           # stack -show will not work. Works only for the
                               # currently executing process.

xmsim: *E,STKRVL: Command works only in the running scope in verilog.
xcelium> scope -set top
xcelium> stack -set 1          # The stack -set command is not implemented for Verilog
xmsim: *E,BADSNL: Given stack level does not exist in the current debug stack frame.
```

Cadence has also implemented a `$stacktrace` system task. You can use this task to print the call stack as part of your own error message output from Verilog.

The `$stacktrace` task prints the same stack trace information that the Tcl `stack` command would print if the command were executed from the point of the system task in the Verilog code. For example:

```
module top;
  task error;
  begin
    $display("Error at");
    $stacktrace;
  end
endtask

task t;
  input integer i;
  if (i == 0)
    error;
endtask

initial
  begin
    t(1);
    t(0);
  end
endmodule
```

```
% xmvlog -nocopyright test.v
% xmelab -nocopyright worklib.top
% xmsim -nocopyright worklib.top
xcelium> run
Error at
Verilog Stack Trace:
0: task top.error at ./test.v:6
1: task top.t at ./test.v:13
2: initial block in top at ./test.v:19

xmsim: *W,RNQUIE: Simulation is complete.
xcelium>
```

status

The Tcl `status` command allows you to display memory and CPU usage statistics and shows the current simulation time.

status Command Syntax

```
status
```

status Command Options

None.

status Command Examples

The following example shows the type of statistics displayed by the `status` command:

```
xcelium> status
```

```
Memory Usage - Current physical: 15.6M, Current virtual: 52.1M  
CPU Usage - 0.0s system + 0.0s user = 0.1s total (8.1s, 0.7% cpu)  
Simulation Time - 0 FS + 0
```

Memory Usage Report Prior to Xcelium 17.04

Prior to Xcelium 17.04, the format of memory usage report looked different, and is shown below. The description given below explains the correlation between the current format with the previous one.

```
Memory Usage - 37.9M program + 14.3M data = 52.1M total  
CPU Usage - 0.0s system + 0.0s user = 0.0s total (11.7s, 0.3% cpu)  
Simulation Time - 110 NS + 0
```

Here, '52.1M total' (prior) is similar to 'Current virtual' memory usage.

stop

The Tcl `stop` command creates or operates on a breakpoint.

The following are the common tasks you can perform with the `stop` command:

- [Creating a Breakpoint](#)
- [Specifying Conditions in stop Command Using Tcl Expressions](#)

stop Command Syntax

```
stop [-create]
    -assert [scope_name]
        [{-all | -depth {levels | all | to_cells} | -vhdl }]
    -at_resfunc <net>
        -cv_limit <num>
    -condition {tcl_expression}
    -delta delta_cycle_number [-relative | -absolute]
        [-start delta_cycle_number]
        [-modulo delta_cycle_number]
        [-timestep]
    -iso_rule rule_name [rule_name ...] [-iso_enable | -iso_disable | -iso_corrupt]
    -label "stop_message"
    -line line_number [scope_name]
        [-all]
        [-file filename]
        [-unit unit_name]
    -object object_names
    -pdname power_domain_name [power_domain_name ...]
        [-isolation [-iso_disable | -iso_enable]]
        [-pd_off]
        [-pd_on]
        [-pd_standby]
        [-pd_trans]
        [-retention [-sr_restore | -sr_save | -sr_corrupt]]
    -process process_name
    -pst table_name [table_name...]
    -pwr_mode_transition mode_transition_name [mode_transition_name ...]
    -randomize [-always] [-object_name | -call_id [call_id_number] [-notify]
    -sr_rule rule_name [rule_name ...] [-sr_save | -sr_restore | -sr_corrupt]
    -subprogram subprogram_name
    -supply supply_name
        -net [-supply_full_on | -supply_partial_on | -supply_undetermined | -voltage]
    -time time_spec [-relative | -absolute [-delbreak 0]]
        [-start time_spec]
        [-modulo time_spec]
    -tlmcpu
        [-access <address>]
        [-description <quoted string>]
        [-pc <address>]
```

```
[-physical]
[-read <address>]
[-size <bytes>]
[-write <address>]
[-vsp]
[-continue]
[-delbreak count]
[-esw]
[-execFile file_name]
[-execute command [-noexecout]]
[-if { tcl_expression }]
[-name break_name]
[-silent]
[-skip count]
-delete {break_name | pattern} ...
-disable {break_name | pattern} ...
-enable {break_name | pattern} ...
-show [{break_name | pattern} ...]
```

stop Command Options

This section describes the options that you can use with the Tcl `stop` command.

`-create`

Creates a breakpoint. This modifier is optional. If used, it must be followed by an option that specifies the breakpoint type.

```
-assert [scope_name]
[{-all | -depth {levels|all|to_cells} |
-vhdl}]
```

Stops at failures for assertions.

This option lets you define a single breakpoint that is shared by multiple assertions in the design. By default, you can stop at failures for all assertions in a specified scope (or the current debug scope if no scope is specified. For example:

```
xcelium> stop -assert
xcelium> stop -assert top
xcelium> stop -assert top.u1
```

This option also supports the following sub-options:

- **-all**: Stops at failures for all assertions in the design hierarchy.

```
xcelium> stop -assert -all
```
- **-depth**: Specifies how many scope levels to descend when searching for assertions. Valid arguments include:
 - **levels**: Descends the specified number of scopes. For example, `-depth 1` means include only the given scope, `-depth 2` means include the given scope and its subscopes, and so on. The default is 1.

```
xcelium> stop -assert -depth 3
xcelium> stop -assert top.u1 -depth 2
```

- **all**: Includes all scopes in the hierarchy below the given scope.

```
xcelium> stop -assert top.u1 -depth all
```

- **to_cells**: Includes all scopes in the hierarchy below the specified scope(s), but stop at cells (Verilog modules with ``celldefine` or VITAL entities with VITAL Level0 attribute).

- **-vhdl**: Stops at failures for VHDL assertions in the given scope.

```
xcelium> stop -assert -vhdl
```

By using the `-assert` option, you can avoid having to define a whole set of breakpoints on the assertions using separate `stop -object` commands. The `-assert` option can also be used with other `stop` command options, such as `-execute` and `-continue`. For example, the following command sets one breakpoint on all assertions in the design hierarchy with a common script to execute when the breakpoint triggers.

```
xcelium> stop -assert -all -execute {input
stopscript.tcl}
```

```
-at_resfunc <net>
    -cv_limit <num>
```

Sets a breakpoint when the resolution function value changes for the target userdefined nettype (UDN). The break occurs in the resolution function for any RNM net within the specified scope.

Optionally, you can set the breakpoint on the number of convergence iterations using by the `cv_limit` option. The `cv_limit` value must be a number larger than the expected number of delta cycle iterations required to resolve a time point.

```
-condition {tcl_expression}
```

Sets a breakpoint that triggers when an object referenced in the `tcl_expression` changes value (wires, signals, registers, and variables) or is written to (memories) AND the expression evaluates to true (non-zero, non-x, non-z).

The simulator does not support stop points on individual bits of registers. If a bit-select of a register appears in the expression, the simulator stops and evaluates the expression when any bit of that register changes value. The same holds true for compressed wires.

See [Tcl Expressions as Arguments](#) for details on the format of conditional expressions.

Objects included in a `-condition` expression must have read access. An error is printed if the object does not have read access. See [Access to Simulation Objects](#).

```
-delta delta_cycle_num
[-absolute] [-relative]
[-start delta_cycle_num ]
[-modulo delta_cycle_num ]
[-timestep]
```

Sets a breakpoint that triggers when the simulation delta cycle count reaches the specified delta cycle.

The delta cycle specification can be absolute or relative (the default) based on which:

- If absolute, the breakpoint is automatically deleted after the delta cycle is reached and the breakpoint triggers.
- If relative, the delta cycle specification is an interval, and the breakpoint stops the simulation every *n*th delta cycle.

Use `-start` to specify the absolute delta cycle at which a repetitive breakpoint is to begin firing. If this cycle is before the current cycle, the first stop occurs at the next cycle at which it would have occurred had the stop been set at the cycle specified with `-start`.

The `-modulo` option is similar to `-start`. Use `-modulo` to specify the absolute delta cycle of the first stop cycle for a repeating delta cycle stop. This differs from `-start` only when the given cycle is more than one repeat interval in the future. In this case, the first stop occurs at a delta cycle less than or equal to one interval in the future, such that a stop eventually occurs at the given cycle. For example, if you set a delta breakpoint to stop the simulation every 10 delta cycles, and specify `-modulo 15`, the simulation stops at delta cycle 5, 15, 25, and so on.

The `-timestep` option provides a way to detect infinite loops (due to infinite delta cycles) in the design. This option halts the simulation if the specified number of delta cycles is created at any

given simulation time. The simulation halts after the first timestep delta cycle is reached, and the simulation cannot be advanced. For example:

```
xcelium> stop -delta 1000 -timestep
Created stop 1
xcelium> stop -show
1          Enabled          Delta 1000 (after 1000
deltas at timestep)
xcelium> run
10 NS + 999 (stop 1: delta cycle 1000)
xcelium> run
10 NS + 999 (stop 1: delta cycle 1000)
```

If you want to proceed with the simulation to debug further, you can add the `-delbreak 1` option to the command, as follows:

```
xcelium> stop -delta 1000 -timestep -delbreak 1
```

You can also execute a `stop -delete` command.

```
xcelium> stop -delete breakpoint_name
```

If you want to exit the simulation once you have hit the delta limit, include the `-execute` option to exit the simulation. For example:

```
xcelium> stop -delta 1000 -timestep -execute
exit
```

Because the `-start` and `-modulo` options apply only to repetitive (`-relative`) time stops, you cannot use `-start` or `-modulo` with the `-timestep` option.

When you execute a `save -environment` command to save your debug environment, this option is written to the script to restore your delta breakpoint pattern

Stops when the specified isolation rule becomes enabled or disabled.

You can specify multiple isolation rule names. The `-iso_rule` option has two sub-options:

- `-iso_disable`: Stop only when the isolation rule becomes disabled.
- `-iso_enable`: Stop only when the isolation rule becomes enabled.

Sets a breakpoint that triggers when the specified line number is about to execute.



To enable the setting of line breakpoints, compile with the `-linedebug` option.

You can use the `stop -line` command to stop at a specified line in one or all instances (module instances or class instances within a given class method).

- **Instance-specific breakpoints:** By default, the breakpoint is

```
-iso_rule rule_name [rule_name ...]
[-iso_enable | -iso_disable]
```

```
-line line_number [scope_name]
[-all] [-file filename]
[-unit unit_name]
```

set on the specified line in the current debug scope.

```
xcelium> stop -line 8
```

To set the breakpoint on a line in a different scope, include the *scope_name* on the command line.

```
xcelium> stop -line 8 top.mid1.bot  
xcelium> stop -line 13 myClassInstance
```

To set a breakpoint on a line in a class method for a specific class instance, the following syntax can be used:

- `stop -line line_number class_handle`
xcelium> stop -line 13 c11
- `stop -line line_number class_instance`
xcelium> stop -line 13 @1
- `stop -line line_number instance_handle`
xcelium> stop -line 13 @1_1
- `stop -`
`line line_number [value class_variable]`
xcelium> stop -line 13 [value c11]

- **Non-instance-specific breakpoints:** To create a breakpoint that is not instance-specific, use the `-all` option. The break occurs on all scopes that are instances of the same module or class.

The `-unit unit_name` option specifies a design unit name for non-instance-specific breakpoints. The stop occurs whenever the line number in the specified design unit is about to execute, no matter where in the design hierarchy that unit appears.

The `-file` option specifies the name of the source file that contains the specified line. This is necessary if the scope has multiple source files.

You can use the `-file` option to set a breakpoint on an executable line of code line in a specified file for all occurrences without specifying a scope or design unit name.

```
xcelium> stop -create -line line_number -  
file filename -all
```

For example, the following command stops the simulation whenever line 48 in the specified file is about to execute.

```
xcelium> stop -create -line 48 -file
./ovm/sv/src/base/ovm_packer.sv -all
```

The following limitations apply when setting a line breakpoint in a specified file:

- A breakpoint cannot be created on a line that is a continuation of a line break. In the example below, you can set a breakpoint on line 6, but not on line 7.

```
xcelium> stop -create -line 7 -file leaf.v -
all
```

```
// File leaf.v
module leaf(input reg b);
    reg c;

    task f();
        c = b &          // Line 6
        b;              // Line 7    // Cannot
set a breakpoint on this line
    endtask

    always @(b)
        f();
endmodule
```

- A breakpoint cannot be created on a line that specifies a `bind` directive.
- A breakpoint cannot be created on a line inside a macro definition. For example:

```
`define PRINT begin : blk \
    $display("%m: b = %d\n", b); \ //
Cannot set a breakpoint on this line
end
```

You can, however, set a breakpoint at the location where the macro is used.

`-object object_name`

Sets a breakpoint that triggers when the specified object changes value (wires, signals, registers, and variables) or is written to (memories). You can also set a breakpoint on an assertion.

By default, vector Verilog wires and VHDL signals are compressed if the model does not require operations on individual bits of the vector. For VHDL, you can set an object breakpoint on a subelement of a compressed vector signal. For Verilog, however, you must elaborate the design with the `-expand` option (`xmelab -expand`) in order to set a breakpoint on a subelement of a compressed vector wire.

You cannot set a breakpoint on a VHDL subprogram object.

The `stop -object` command can be used to monitor changes to SystemVerilog dynamic objects such as class instances, queues, dynamic arrays, associative arrays, and strings.

- For class instances, you can set a breakpoint that triggers when a class instance is created or deleted or when data members are written to. The `stop -object` command takes the following syntax:

```
stop -object classObject
```

where *classObject* takes the following forms:

- Class handle

```
stop -object c11
```
- Instance handle

```
stop -object @5_1
stop -object [value c11]
```
- Class instance

```
xcelium> stop -object @5
```
- Class property

```
stop -object C::staticVar
stop -object @1_1.dynamicVar
```

Breakpoints set using the class instance handle (for example, `stop -object @1_1` or `stop -object @1_1.member`) do not live beyond the life of the specified class object; in other words, breakpoints are garbage collected once a class object is no longer valid. When the instance is no longer valid, the simulator stops and produces a message noting that the stop is no longer valid.

For queues, dynamic arrays, associative arrays, and strings, you can set a breakpoint that triggers when the object is created, deleted, or resized, or when data elements are written to.

You can use wildcard characters in the argument to a `stop -object` command.

- The asterisk (`*`) matches any number of characters.
- The question mark (`?`) matches any one character.

You cannot use wildcard characters inside escaped names.

The object specified as the argument must have read access for the breakpoint to be created. An error is printed if the object does not have read access. See [Access to Simulation Objects](#) for details on specifying access to simulation objects.

You can apply the `stop -object` command to the following SystemC objects: `sc_signal`, `sc_clock`, `sc_in`, `sc_out`, `sc_inout`, `ncsc_viewable`, `sc_fifo`, `sc_fifo_in`, `sc_fifo_out`, `sc_mutex`, `sc_semaphore`, `tlm_fifo`, and `tlm_req_rsp_channel`. For the templated objects in this

list, `stop -object` is supported when the object is templated by a C++ data type or by a SystemC built-in data type, or by a user-defined data type that implements the `<<` operator.

For `sc_port` and `sc_export` objects, `stop -object` is supported if the `sc_object` bound to the port or export supports `stop -object`. The debug scope remains unchanged when the `stop` command is triggered.

For details on using the `stop` command on the SystemC objects `sc_fifo`, `sc_mutex`, `sc_semaphore`, `tlm_fifo`, and `tlm_req_rsp_channel` refer to [Debugging SystemC Models](#).

If an arbitrary SystemC object declares support for value and value change callback (the member methods `ncsc_supports_value()` and `ncsc_supports_value_change_callback()`), you can apply the `stop -object` command to that object. See [Specifying Supported Commands](#).

When you use the `stop -object` command, the stop occurs whenever the value changes. That is, writing the same value to the object does not cause the simulation to stop. If you are simulating your design with `-scSyncEveryDelta` turned on, the `stop -object` command returns control to the Tcl prompt after the delta cycle, in which a value is written to the object. If `-scSyncEveryDelta` is off, the command returns control to the Tcl prompt after all of the delta cycles for SystemC are executed for the current time. The current debug scope does not change when control returns to the Tcl prompt because of an executed `stop` command for a SystemC object.

```
-pdname power_domain_name
[power_domain_name ...
[-isolation
[-iso_disable | -iso_enable]]
[-pd_off] [-pd_on]
[-pd_standby] [-pd_trans]
[-retention [-sr_restore | -sr_save]]
```

Sets a breakpoint that triggers when the specified power domain changes status. If no options are specified, the simulation stops when the power domain is powered down or powered up. You can specify multiple power domain names.

The `-pdname` option has several sub-options that let you create power domain breakpoints that trigger under specific conditions:

- **-pd_off**: Stops when the power domain turns off.
- **-pd_on**: Stops when the power domain turns on. This option stops the simulation when the specified power domain has transitioned to the ON state or to the UNINITIALIZED state. The state of the power domain is UNINITIALIZED when the simulation is being controlled by active state conditions, and when the power domain is on but there are no active state conditions enabled to specify the nominal condition to which the domain should transition.
- **-pd_standby**: Stops when the power domain enters standby mode.
- **-pd_trans**: Stops when the power domain transitions. The breakpoint triggers when the specified power domain starts transitioning from one nominal condition to a different nominal condition.
- **-isolation**: Stops when any isolation rule associated with the power domain is enabled or disabled.
- **-iso_disable**: Stops when any isolation rule associated with the power domain is disabled.
- **-iso_enable**: Stops when any isolation rule associated with the power domain is enabled.
- **-retention**: Stops when any state retention rule associated with the power domain saves or restores its variables.
- **-sr_restore**: Stops when any state retention rule associated with the power domain restores its variables.
- **-sr_save**: Stops when any state retention rule associated with the power domain saves its variables.

```
-process process_name
```

Sets a breakpoint that triggers when the specified VHDL named process starts executing or when it resumes executing after a wait statement.



To enable the setting of line breakpoints, compile with the `-linedebug` option.

```
-pst table_name [table_name...]
```

Sets a breakpoint to stop the simulation when a power state in the specified power state table (PST) changes its value.

This option takes one or more `table_name` values as input. The `table_name` specification is either a full or relative path to the power state table in the current scope.

```
-pwr_mode_transition transition_name  
[transition_name ...]
```

Stops when the specified power mode transition starts and ends.

```
-randomize [object_name] [-always]  
[-call_id call_id_number]  
[-notify]
```

Sets a breakpoint in SystemVerilog `randomize()` function calls.

The SystemVerilog built-in `randomize()` function returns the value 1 for success or 0 for failure. A failure occurs because there are conflicts in the collection of constraints to solve or because a variable is over-constrained.

The `stop -create -randomize` command lets you set a breakpoint in `randomize()` method calls. You can then use other Tcl commands, such as `deposit -constraint_mode`, `deposit -rand_mode`, `constraint`, and `run -rand_solve`, to debug the randomization failures. See [run Command Examples](#) for an example of using these commands.

`stop -randomize` commands are supported for calls to class and scope `randomize` methods.

The `-randomize` option has several sub-options:

- `-always`: By default, simulation stops at the end of `randomize()` calls when the call is about to return 0, or failure. If you include the `-always` option, simulation stops for all `randomize()` calls, regardless of the return status of the call.
You can include an `object_name` argument to stop the simulation in specific `randomize()` calls. The argument can be a class name or a module name. The simulator stops on a failure in any call of the `randomize()` method in the specified module or with the specified class name. If `-always` is specified, the simulator stops in all calls to the `randomize()` method.
- `-call_id <call_id_number>`: To stop the simulation at a particular randomization call number. You must specify the call id number with this option.
- `-notify`: To receive a notification when a `randomize` breakpoint is executed.

The `stop -randomize` command can be used in the following ways:

- When the `stop -randomize` command is used with the `-call_id` option, the simulation stops at a specified `randomize` call in case of a failure. When `stop -randomize -always` command is used with the `-call_id` option, the simulation always stops at the specified `randomize` call irrespective of the return status of the call.
- When the `stop -randomize` is used with the option, you are notified only when a call fails. When the `stop -randomize`

`-always` command is used with the `-notify` option, you are notified for all randomize calls.

- When the `stop -randomize` command is used with both `-call_id` and `-notify` options, you are notified when the randomization fails at the specified randomize call and simulation also stops. When both `-call_id` and `-notify` are used with the `-randomize -always` option, you are always notified at a particular randomize call, and simulation also stops.

```
-sr_rule rule_name [ rule_name ...]  
[-sr_save | -sr_restore]
```

Stops when the specified state retention rule saves or restores its variables.

You can specify multiple state retention rule names.

The `-sr_rule` option has two suboptions:

- `-sr_restore`—Stop only when the state retention rule restores its variables.
- `-sr_save`—Stop only when the state retention rule saves its variables.

Sets a breakpoint that triggers when the specified VHDL subprogram or Verilog task or function is called.



To enable the setting of subprogram breakpoints, compile with the `-linedebug` option.

```
-supply supply_name -net  
[-supply_full_on |  
-supply_partial_on |  
-supply_undetermined |  
-voltage]
```

Sets a breakpoint that stops the simulation when the named supply changes its state.

To create a breakpoint for a supply net, you must specify the `supply_name` and include the `-net` modifier.

The Tcl `stop -supply supply_name -net` specification supports the following sub-options:

- **`-supply_full_on`**: Stops when the supply net state changes to FULL_ON.
- **`-supply_partial_on`**: Stops when the supply net state changes to PARTIAL_ON.
- **`-supply_undetermined`**: Stops when the supply net state changes to UNDETERMINED.
- **`-supply_off`**: Stops when the supply net state changes to OFF.
- **`-voltage`**: Stops when the voltage of the specified supply net changes its value.


```
-time time_spec  
[-absolute] [-relative]  
[-start time_spec ]  
[-modulo time_spec]
```

Sets a breakpoint that triggers at the specified time. The time can be absolute or relative (the default). Relative time breakpoints are periodic, stopping, for example, every 10 ns. Absolute time breakpoints are, by default, automatically deleted after they trigger. Use the `-delbreak 0` option to prevent an absolute time breakpoint from being deleted after it triggers.

Use `-start` to specify the absolute simulation time at which a relative time breakpoint is to begin firing. If this time is before the current simulation time, the first stop occurs at the next future time at which it would have occurred had the stop been set at the time specified with `-start`.

The `-modulo` option is similar to `-start`. Use `-modulo` to specify the absolute simulation time of the first stop time for a repeating stop. This differs from `-start` only when the given time is more than one repeat interval in the future. In this case, the first stop occurs at a time less than or equal to one interval in the future such that a stop eventually occurs at the given time. For example, if you set a time breakpoint to stop the simulation every 100 ns, and specify `-modulo 250`, the simulation stops at time 50, 150, 250, and so on.

When you execute a `save -environment` command to save your debug environment, this option is written to the script to restore your time breakpoint pattern.

```
-tlmcpu
  [-access <address>]
  [-description <quoted string>]          [-pc
<address>]  [-physical]
  [-read <address>] [-size <bytes>]
  [-write <address>]
```

Stops when a given CPU access given memory address.

The Tcl `stop -tlmcpu` supports the following sub-options:

- **-access <address>**: Stops when any of the TLM CPUs access (reads or writes) given memory address.
- **-description <quoted string>**: Associates a description with the stop command.
- **-pc <address>**: Stops when TLM CPU reaches given memory address.
- **-physical**: Specifies that a physical address should be used when possible.
- **-read <address>**: Stops when any of the TLM CPUs read given memory address.
- **-size <bytes>**: Specifies the associated size to stop when the TLM CPU accesses (reads or writes) given memory address.
- **-write <address>**: Stops when any of the TLM CPUs write given memory address (default).

```
-vsp
```

Stops after a VSP object access given memory address.

```
-name break_name
```

Specifies a name for the breakpoint. This name can then be used to delete, disable, or enable the breakpoint. If you do not use `-name`, breakpoints are numbered sequentially.

```
-continue
```

Resumes the simulation after executing the breakpoint. The simulator does not go into interactive mode.

```
-delbreak count
```

Deletes the breakpoint after it has triggered *count* number of times.

By default, an absolute time breakpoint is deleted after it triggers. Set the *count* argument to 0 to prevent the absolute time breakpoint from being deleted. For example:

```
xcelium> stop -time -absolute 2ns -delbreak 0
```

```
-esw
```

Stop after ESW Core object access given memory address

```
-execFile file_name
```

Executes the specified file, housing a list of Tcl commands, when the breakpoint is triggered.

This command allows you to run more than one command bundled together in one file. Each command should be listed on its own separate line. Consider the following example:

```
stop -continue -condition (#a="1") -execFile
mycommands.tcl
```

Whenever the design gets the value `a == "1"`, Tcl executes the commands written to the file, `mycommands.tcl`.

`-execute command`
`[-noexecout]`

Executes the specified Tcl command when the breakpoint is triggered.

If the command that you want to execute requires an argument, enclose the command and its argument in curly braces.

You also can specify that you want to execute a list of commands. Separate the commands with a semicolon. Tcl, however, displays only the output of the last command.

Use the `-noexecout` option to suppress output from the Tcl command.

`-if {tcl_expression}`

Sets a condition on the breakpoint. The breakpoint triggers only if the given Tcl boolean expression evaluates to true (non-zero, non-x, non-z). This option can be used with any breakpoint type. See [Tcl Expressions as Arguments](#) for more information on the format of the *tcl_expression* argument.

Objects included in an `-if` expression must have read access. An error is printed if the object does not have read access. See [Access to Simulation Objects](#) for details on specifying access to simulation objects.

`-label "stop_message"`

Specifies a label for the breakpoint message. The breakpoint issues the *stop_message*.

For example:

```
xcelium> stop -line 19 -file testbench.sv -label
"setup phase"
Created stop 1
xcelium> run
0 FS + 0 setup phase
```

The `-name` option can be used with the `-label` option, but it is ignored during the generation of the breakpoint message. For example:

```
xcelium> stop -line 19 -file testbench.v -name
mybreakpoint
Created stop mybreakpoint
xcelium> run
0 FS + 0 (stop mybreakpoint: ./testbench.v:19)
./testbench.v:19          clk_tb = 1'b0;
```

```
xcelium> stop -line 19 -file testbench.v -name
mybreakpoint -label "mylabel"
Created stop mybreakpoint
xcelium> run
0 FS + 0 mylabel
./testbench.v:19          clk_tb = 1'b0;
```

`-silent`

Suppresses the display of the message that is printed when a breakpoint triggers.

| | |
|--|--|
| <code>-skip count</code> | <p>Tells the simulator to ignore the breakpoint for the first <i>count</i> times that it triggers.</p> <p>You can use <code>-skip</code> to set a breakpoint on the <i>n</i>th occurrence of an event; in particular, you can use it to get inside <code>for</code> loops.</p> |
| <code>-delete {break_name pattern} ...</code> | <p>Deletes the breakpoint(s) specified by the argument. The argument can be a break name, a list of break names, a combination of literal break names and patterns, and wildcard patterns.</p> |
| <code>-disable {break_name pattern} ...</code> | <p>Disables the breakpoint(s) specified by the argument without deleting it. The argument can be a break name, a list of break names, a combination of literal break names and patterns, and wildcard patterns.</p> |
| <code>-enable {break_name pattern} ...</code> | <p>Enables the previously disabled breakpoint(s) specified by the argument. The argument can be a break name, a list of break names, a combination of literal break names and patterns, and wildcard patterns.</p> |
| <code>-show [{break_name pattern} ...]</code> | <p>Show the status of the breakpoint(s) specified by the argument. If no breakpoint is specified, all breakpoints are shown. The argument can be a break name, a list of break names, a combination of literal break names and patterns, and wildcard patterns.</p> |

stop Command Examples

The following `-stop` command examples are included:

- [Object Breakpoints](#)
- [Monitoring Changes to Class Objects](#)
- [Monitoring Changes to Queues, Arrays, and Strings](#)
- [Line Breakpoints](#)
- [Setting Line Breakpoints on Class Instances](#)
- [Time Breakpoints](#)
- [Delta Breakpoints](#)
- [Condition Breakpoints](#)
- [Process Breakpoints](#)
- [Subprogram Breakpoints](#)
- [Power Domain Breakpoints](#)
- [Power Mode Transition Breakpoints](#)

- [Examples of Other stop Command Modifiers](#)

Object Breakpoints

The following command creates a breakpoint that stops simulation when `sum` or `cin` changes value:

```
xcelium> stop -object sum cin
Created stop 1
```

The following command creates a breakpoint named `mybreak` that stops simulation when `sum` changes value:

```
xcelium> stop -object sum -name mybreak
Created stop mybreak
```

The following command creates a breakpoint that triggers when `sum` changes value. The breakpoint is ignored the first three times it triggers:

```
xcelium> stop -object sum -skip 3
```

The following command creates a breakpoint on a Verilog vector wire called `dimes` :

```
xcelium> stop -object test_drink.dimes
```

The Verilog vector wire `dimes` is a compressed vector. You cannot set an object breakpoint on a subelement of a compressed Verilog vector unless you have elaborated the design with the `-expand` command-line option. The following example shows the error message that is generated if the design has not been elaborated with `-expand` :

```
xcelium> stop -object test_drink.dimes[2]
xmsim: *E,STWSUB: Cannot set stop point on subelement of a compressed wire or signal.
```

You can set an object breakpoint on a subelement of a compressed VHDL vector without using the `-expand` option when you elaborate the design. The following command creates a breakpoint that stops simulation when `:vec1(29)` changes value:

```
xcelium> stop -object :vec1(29)
```

The following command creates breakpoints on all objects in the current scope:

```
xcelium> stop -object *
Created stop 1
```

The following command creates breakpoints on all objects in the current scope that have a name that starts with `bus` :

```
xcelium> stop -object bus*
```

The following command creates breakpoints on all objects in the scope `top` that have two letters and that have names that start with the letter `c` :

```
xcelium> stop -object top.c?
```

The following command creates a breakpoint that stops simulation when `clr` changes value. The `value data` command is executed when the breakpoint triggers. Because the `value` command requires an argument, it must be enclosed in curly braces:

```
xcelium> stop -object clr -execute {value data}
```

The following command creates a breakpoint that triggers when `clr` changes value. The `value data` command is executed when the breakpoint triggers. The `-continue` option prevents the simulator from entering interactive mode every time the stop triggers:

```
xcelium> stop -object clr -execute {value data} -continue
```

The following command creates an object breakpoint that triggers when `data` changes value. The `-delbreak` option specifies that the breakpoint is deleted after it triggers three times:

```
xcelium> stop -object data -continue -delbreak 3
```

The following command creates a breakpoint that triggers when `clk` changes value, but only if `clk` is high. See [Tcl Expressions as Arguments](#) for details on the syntax of the argument to the `-if` option:

```
xcelium> stop -object clk -if {#clk == 1} -continue
```

The following command creates a breakpoint that triggers when `data[1]` has the value 1 and the time becomes greater than 3 ns :

```
xcelium> stop -object data -if {#data[1] == 1 && [time ns -nounit] > 3}
```

The following command shows the error message that is displayed if you run in regression mode (no read, write, or connectivity access to simulation objects) and then try to set an object breakpoint on an object that does not have read access:

```
xcelium> stop -object r
xmsim: *E,OBJACC: Object must have read access: top.r.

xcelium> stop -object w
xmsim: *E,OBJACC: Object must have read access: top.w.
```

Monitoring Changes to Class Objects

The following example shows how you can use `stop -object` to monitor changes to class objects:

```

module top;
  class C;
    static int staticVar;
    int dynamicVar;
    // Writes to a static variable
    task setStatic(int v);
      staticVar = v;
    endtask
    // Writes to a dynamic variable
    task setDynamic(int v);
      dynamicVar = v;
    endtask
    // Writes to a local variable (within a task scope)
    task setLocal(int v);
      int localVar;
      localVar = v;
    endtask
  endclass:C
  C cl1, cl2;
  initial begin
    cl1 = new;
    cl2 = new;
    #1 cl1.setStatic(417);
    #1 cl1.setDynamic(93);
    #1 cl1.setLocal(54);
    // Writes to static variable thru cl2
    #1 cl2.setStatic( -14 );
  end
endmodule

```

```

;# Set breakpoint on a class handle
xcelium> stop -object cl1
Created stop 1
xcelium> ;# The following command generates an error because the object has
xcelium> ;# not been allocated yet
xcelium> stop -object @1_1
xmsim: *E,HPIINV: Heap Index Invalid: there is no object allocated at this index: @1_1.
xcelium> ;# Run until we stop due to cl1 change
xcelium> run
0 FS + 0 (stop 1: top.cl1 = @1_1)
./test.sv:27      cl1 = new;
xcelium> ;# Set breakpoint on the class instance handle after the object is allocated.
xcelium> stop -object @1_1
Created stop 2
xcelium> ;# Set breakpoints on individual class properties
xcelium> stop -object C::staticVar

```

```
Created stop 3
xcelium> stop -object @1_1.dynamicVar
Created stop 4
xcelium> run
1 NS + 0 (stop 2: top.C@1_1)
1 NS + 0 (stop 3: top.C::staticVar = 417)
./test.sv:9      staticVar = v;
xcelium> run
2 NS + 0 (stop 2: top.C@1_1)
2 NS + 0 (stop 4: top.C@1_1.dynamicVar = 93)
./test.sv:14     dynamicVar = v;
xcelium> run
4 NS + 0 (stop 2: top.C@1_1)
4 NS + 0 (stop 3: top.C::staticVar = -14)
./test.sv:9      staticVar = v;
xcelium> run
xmsim: *W,RNQUIE: Simulation is complete.
```

Monitoring Changes to Queues, Arrays, and Strings

The following example shows how you can monitor changes to queues, dynamic arrays, associative arrays, and strings. A `stop -object` command monitors the object creation or deletion, array resize operations, and data element modifications:

```
module top;
  class C;
    int i;
  endclass
  int q[$];
  int aa[int];
  int da[];
  string s;
  C c1 = new;
  initial begin
    #1 s = "foo";
    #1 da = new[4];
    #1 aa[15] = 4;
    #1 aa[4] = 10;
    #1 q[0] = 1;
    #1 da.delete;
  end
endmodule
```

```
xcelium> ; # Set a breakpoint on the queue, assoc. array, dynamic array, string
xcelium> stop -object s
Created stop 1
```



```
xcelium> stop -object da
Created stop 2
xcelium> stop -object aa
Created stop 3
xcelium> stop -object q
Created stop 4
xcelium> run ;# Write to string.
1 NS + 0 (stop 1: top.s = foo)
./test.sv:15 #1 s = "foo";
xcelium> run ;# Run. Stop when dynamic array is created.
2 NS + 0 (stop 2: top.da = (0,0,0,0))
./test.sv:16 #1 da = new[4];
xcelium> run ;# Write to location [15] of the associative array.
3 NS + 0 (stop 3: top.aa = 1)
./test.sv:17 #1 aa[15] = 4;
xcelium> run ;# Write to location [4] of the associative array.
4 NS + 0 (stop 3: top.aa = 2)
./test.sv:18 #1 aa[4] = 10;
xcelium> value -keys aa ;# Get a list of indices (keys) for the associative array.
4 15
xcelium> run ;# Write to location [0] of the queue.
5 NS + 0 (stop 4: top.q = 1)
./test.sv:19 #1 q[0] = 1;
xcelium> run ;# Delete the dynamic array. Stop when it is deleted.
6 NS + 0 (stop 2: top.da)
./test.sv:20 #1 da.delete;
xcelium> run
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

Line Breakpoints

The following command creates a breakpoint that stops simulation when line number 10 in the current debug scope is about to execute:

```
xcelium> stop -line 10
```

The following command creates a breakpoint that stops simulation when line number 13 in scope `mid1.bot1` is about to execute:

```
xcelium> stop -line 13 mid1.bot1
```

In the following command, the `-all` option specifies that the stop is non-instance-specific. The breakpoint occurs on all scopes that are instances of the same module. For example, if there are two instances of module `m16`, as follows, the breakpoint triggers when line 13 in either `counter1` or `counter2` is about to execute:

```
module board;
  <declarations>
    m16 counter1 (...);
    m16 counter2 (...);
  <code>
endmodule
```

```
xcelium> stop -line 13 counter1 -all
```

The following command is equivalent to the command shown in the previous example. Both commands create non-instance-specific breakpoints:

```
xcelium> stop -line 13 -unit m16
```

In the following example, the `-file` option specifies which of the source files that make up the given scope (or the debug scope if none is given) contains the specified line. This is necessary if the scope has multiple source files:

```
xcelium> stop -line 13 counter -file foo.v
```

The following command creates a breakpoint at a line inside a function `print`, which is defined inside a class `my_class` inside a package. The definition of the package is in file `package.sv`. The simulator stops whenever line 5 in file `package.sv` is about to execute:

```
// File package.sv
package p;
    class my_class;
        int a;
        function void print();
            $display("a = %d\n", a);    // Line 5
        endfunction
    endclass
endpackage

// File top.sv
`timescale 1 ns/1 ps
module top;
    leaf l();
endmodule

// File leaf.sv
`timescale 1 ns/1 ps
module leaf;
import p::*;
my_class obj;
initial
begin
    obj = new;
    #1 obj.a = 1;
    #2 obj.a = 0;
end
always @(obj.a)
    obj.print();
endmodule
```

```
% xrun -nocopyright -linedebug package.sv top.sv leaf.sv
```

```
xrun: *W,BADPRF: The -linedebug option may have an adverse performance impact.
```

```
file: package.sv
```

```
    package worklib.p:sv
```

```
        errors: 0, warnings: 0
```

```
...
```

```
    Elaborating the design hierarchy:
```

```
...
```

```
...
```

```
    Writing initial simulation snapshot: worklib.top:sv
```

```
Loading snapshot worklib.top:sv ..... Done
```

```
xcelium> stop -line 5 -file package.sv -all
```

```
Created stop 1
```

```
xcelium> stop -show
```

```
1      Enabled      Line: ./package.sv:5 (unit: worklib.p:sv)
xcelium>
```

Setting Line Breakpoints on Class Instances

The following example illustrates how you can set a breakpoint on a line of source code in a class method. The example uses the following Verilog code:

```
module top;
  class C;
    int value;
    task show;
      $display("class data follows");    // Line 6
      $display("value=%d", value);      // Line 7
      $display("complement=%d", ~value); // Line 8
      $display("That's all");           // Line 9
    endtask
    function new (int v);
      value = v;
    endfunction
  endclass
  C cl1, cl2;
  initial begin
    cl1 = new(10);
    cl2 = new(20);
    $stop;
    // Stop 1 applies to all instances at line 9
    // Stop 2 is specific to cl1 at line 7
    // Stop 3 is specific to cl2 at line 8
    // This call will stop for breaks 1 and 2
    cl1.show();
    // This call will stop for breaks 1 and 3
    cl2.show();
  end
endmodule
```

The example is run in single-step invocation mode with *xrun* using the following command:

```
% xrun -q -linedebug -access +rw -tcl test.sv
xrun: *W,BADPRF: The -linedebug option may have an adverse performance impact.
      Top level design units:
          top
xcelium> source /proj/install/tools/xm/files/xmsimrc
xcelium> run
Simulation stopped via $stop(1) at time 0 FS + 0
./test.sv:22      $stop;
xcelium> stop -line 9 -all      ;# This stop applies to all instances
```

```
Created stop 1
xcelium> stop -line 7 c11           ;# This stop applies only to c11
Created stop 2                      ;# Can also use:
                                   ;# stop -line 7 @1
                                   ;# stop -line 7 [value c11]

xcelium> stop -line 8 c12           ;# This stop applies only to c12
Created stop 3
xcelium> run
class data follows
0 FS + 0 (stop 2: ./test.sv:7)
./test.sv:7      $display("value=%d", value);    // Line 7
xcelium> run
value=           10
complement=      -11
0 FS + 0 (stop 1: ./test.sv:9)
./test.sv:9      $display("That's all");         // Line 9
xcelium> run
That's all
class data follows
value=           20
0 FS + 0 (stop 3: ./test.sv:8)
./test.sv:8      $display("complement=%d", ~value); // Line 8
xcelium> run
complement=      -21
0 FS + 0 (stop 1: ./test.sv:9)
./test.sv:9      $display("That's all");         // Line 9
xcelium> run
That's all
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

Time Breakpoints

The following command creates a breakpoint that stops simulation at absolute time 200 ns. The breakpoint is automatically deleted after it triggers. If you want to prevent the breakpoint from being deleted after it triggers, include the `-delbreak 0` option.

```
xcelium> stop -time 200 ns -absolute
```

The following command creates a repetitive breakpoint that stops the simulation every 200 ns and then executes the `value` command. The `-relative` option is the default for time breakpoints.

```
xcelium> stop -time 200 ns -relative -execute {value data}
```

The following command creates a repetitive breakpoint that stops the simulation every 200 ns. The `-start` option specifies the absolute time at which the breakpoint starts. For example, if the current simulation time is 300 ns, the breakpoint stops the simulation at time 600, 800, 1000, and so on.

```
xcelium> stop -time 200 ns -start 600 ns
```

In the following example, assume that the current simulation time is 300 ns. The absolute time specified with `-start` is before the current simulation time. The first stop occurs at the next future time at which it would have occurred had the stop been set at the time specified with `-start`. In this example, the first stop occurs at time 450 ns.

```
xcelium> stop -time 200 ns -start 250 ns
```

The following example shows how the `-modulo` option is used to save a breakpoint pattern. Suppose that you simulate to time 300 ns and then set a repetitive breakpoint with the following command:

```
xcelium> stop -time 200 ns -start 350 ns
```

This command stops the simulation at time 350, 550, 750, and so on. If you then execute a `save -environment` command to save your debug environment, the following line is written to the script:

```
xcelium> stop -create -name 1 -time 200 NS -relative -modulo 950 NS
```

If you then exit and re-enter the simulation and source the script containing this command, the breakpoint pattern is re-established. In this example, if you reinvoke the simulation and start at time 0, the breakpoint triggers the first time at time 150. It then triggers at 350, 550, 750, and so on.

The following command includes the `-if` option to set a breakpoint at time 100 ns (relative) if `data[1]` has the value 1.

```
xcelium> stop -time 100 ns -if {#data[1] == 1}
```

Delta Breakpoints

The following command creates a breakpoint that stops the simulation when it reaches 20 delta cycles. The breakpoint is automatically deleted after it triggers.

```
xcelium> stop -delta 20 -absolute
```

The following command creates a repetitive breakpoint that stops the simulation every 10 delta cycles. The `-start` option specifies the absolute delta cycle at which the breakpoint starts. For example, if the current delta cycle count is 0, the breakpoint stops the simulation when the delta cycle count is 30, 40, 50, and so on.

```
xcelium> stop -delta 10 -start 30
```

Condition Breakpoints

In a condition breakpoint, the argument to the `-condition` option is a Tcl expression. See [Tcl Expressions as Arguments](#) for more information on writing these expressions.

The following command sets a condition breakpoint that stops the simulation when `count`, the output of a 32-bit counter, has the value 100, decimal. The signal `count` is available from the top level of the hierarchy.


```
xcelium> stop -condition {[value %e count(0)] == `1'}
```


Note the use of `%e` in the VHDL example. The Tcl expression evaluator must know what enumeration type to use in the comparison expression, so at least one of the enumeration literals in the expression must be in the fully qualified format, which includes a path to the type declaration that defines the literal.

The `value` command used with the `%e` format specifier returns the current value of a VHDL object in the fully qualified format. This value is substituted into the command line before the expression is passed to the Tcl expression evaluator.

The following command uses the `/` character as the hierarchy separator. The absolute path begins with `/` followed by the top-level Verilog module or the top-level VHDL entity.

```
xcelium> stop -condition {#/d/top/count == 100}
```

The following commands use the `radix#number` syntax to specify the value.

 The `tcl_relaxed_literal` variable must be set to 1 to enable this functionality.

Decimal radix (for example `10#25`) is not supported for VHDL.

```
set tcl_relaxed_literal 1
xcelium> stop -condition {#/top/count == 2#11}
xcelium> stop -condition {#/top/count == 8#11}
xcelium> stop -condition {#/top/count == 10#11} ;# Decimal radix not supported
                                                for VHDL
xcelium> stop -condition {#/top/count == 16#11}
```

In the following command, the `-if` option is used to conditionalize the condition breakpoint. This breakpoint stops the simulation at the next positive edge of the clock if `en1` or `en2` is 1:

Verilog:

```
xcelium> stop -condition {#clock == 1} -if {#en1 || #en2}
```

VHDL:

```
xcelium> stop -condition {#clk_n == '1'} -if {#enable=='1' || #reset_n=='1'}
```

The following command stops the simulation at 5 ns (absolute time). After that, `clock` changes depending on the condition in the `if` expression, and this happens repeatedly every 5 ns. The `-continue` option is used to prevent the simulation from stopping every time the breakpoint triggers. VHDL requires use of the single quotation marks:

```
xcelium> stop -time 5 ns -start 5 ns -execute {if {#clk == '0'} {force clk '1'}
      else {force clk '0'}} -continue
```

The following command stops the simulation when the value of the specified signal has the value `x`. Notice that in the Tcl expression, the case-equality operator (`===`) is used. For this operator, bits that are

unknown are included in the comparison, and the result of the expression is always 1 (true) or 0 (false):

```
xcelium> stop -create -condition {#top.load === 1'bx}
```

In the following command, the logical comparison operator (= or ==) is used. These operators return the unknown value (x) if either operand is unknown. In a conditional expression, an unknown result is treated as false. Therefore, the following command *does not* stop the simulation when the signal has the value x:

```
xcelium> stop -create -condition {#top.load == 1'bx}
```

The following command compares the value of an unpacked array inside `stop -condition`:

```
stop -condition {value -packed unpacked_array == 4'b1010}
```

Process Breakpoints

The following command sets a breakpoint that stops the simulation whenever the process called `:load_action` is executed:

```
xcelium> stop -process :load_action
```

Subprogram Breakpoints

The following command sets a breakpoint that stops the simulation when the VHDL subprogram `procedure1` is called. You can also set a subprogram breakpoint on a Verilog task or function:

```
xcelium> stop -subprogram procedure1
Created stop 1
xcelium> run
0 FS + 0 (stop 1: Subprogram :procedure1)
./scope_test.vhd:11 tmp1 := not tmp1;
```

The following command sets a breakpoint that stops the simulation when the VHDL subprogram `function2` is called:

```
xcelium> stop -subprogram function2
Created stop 1
xcelium> run
0 FS + 0 (stop 1: Subprogram :function2)
./scope_test.vhd:29 tmp5_local := tmp5 + 1;
```

Power Domain Breakpoints

The following partial CPF file is used for the examples in this section. The CPF file defines:

- Power domain `PDau`, which has an associated state retention rule (`PDau_sr`) and an associated isolation rule (`PDau_iso`)

- Power domain `PDrf`, which has an associated state retention rule (`PDrf_sr`) and an associated isolation rule (`PDrf_iso`)

```
set_design core

create_power_domain -name PDau \
                    -instances alu_inst/au1 \
                    -shutoff_condition {pcu_inst/pau[2]}

create_power_domain -name PDrf \
                    -instances rf_inst \
                    -shutoff_condition {pcu_inst/prf[2]}

create_state_retention_rule -name PDrf_sr \
                            -instances {rf_inst/rf[0] rf_inst/rf[1] rf_inst/rf[2]
                                         rf_inst/rf[3] rf_inst/rf[4] rf_inst/rf[5]
                                         rf_inst/rf[6] rf_inst/rf[7]} \
                            -restore_edge {!pcu_inst/prf[1]}

create_state_retention_rule -name PDau_sr \
                            -instances {alu_inst/au1/z* } \
                            -restore_edge {!pcu_inst/pau[1]}

create_isolation_rule -name PDau_iso \
                     -from PDau \
                     -pins alu_inst/au1/z* \
                     -isolation_condition {pcu_inst/pau[0]} \
                     -isolation_output high

create_isolation_rule -name PDrf_iso \
                     -from PDrf \
                     -pins rf_inst/z* \
                     -isolation_condition {pcu_inst/prf[0]} \
                     -isolation_output high
```

The following command sets a breakpoint on power domain `PDau`. Simulation stops when the power switch enable signal is asserted and the power domain is powered down, or when it is deserted and power is restored to the domain:

```
xcelium> scope TESTBENCH.inst
xcelium> stop -pdname PDau
Created stop 1
xcelium> stop -show
1      Enabled      Power Domain TESTBENCH.inst.PDau
xcelium> run
15800 NS + 6 (stop 1: Power Domain TESTBENCH.inst.PDau is OFF)
TESTBENCH.inst.PDau ./nano.cpf line 16
```

```
xcelium> value pcu_inst.pau[2]
1'h1
xcelium> run
23800 NS + 6 (stop 1: Power Domain TESTBENCH.inst.PDau is ON)
TESTBENCH.inst.PDau ./nano.cpf line 16
xcelium> value pcu_inst.pau[2]
1'h0
xcelium>
```

The following command sets a breakpoint on power domains `PDau` and `PDrf`:

```
xcelium> stop -pdname PDau PDrf
```

The following command creates a breakpoint on power domain `PDau`. The `-pd_off` option specifies that simulation stops when the domain powers down:

```
xcelium> stop -pdname -pd_off PDau
Created stop 1
xcelium> run
15800 NS + 6 (stop 1: Power Domain TESTBENCH.inst.PDau is OFF)
TESTBENCH.inst.PDau ./nano.cpf line 16
xcelium> run
60600 NS + 6 (stop 1: Power Domain TESTBENCH.inst.PDau is OFF)
TESTBENCH.inst.PDau ./nano.cpf line 16
```

The following command creates a breakpoint on power domain `PDrf`. The `-isolation -iso_enable` option specifies that simulation stops when the isolation rule associated with the domain `PDrf` (`PDrf_iso`) is enabled:

```
xcelium> stop -pdname PDrf -isolation -iso_enable
Created stop 1
xcelium> stop -show
1          Enabled          Power Domain Isolation TESTBENCH.inst.PDrf_iso (enable)
xcelium> run
99800 NS + 6 (stop 1: Isolation Rule TESTBENCH.inst.PDrf_iso: is ENABLED)
TESTBENCH.inst.PDrf_iso ./nano.cpf line 72
```

The following `stop` command creates a breakpoint on power domain `PDau`. The `-retention` option specifies that simulation stops when the state retention rule associated with the domain `PDau` (`PDau_sr`) either saves or restores its variables. A subsequent `stop` command includes the `-retention -sr_save` option to specify that simulation stops only when the state retention rule saves its variables:

```
xcelium> scope inst
xcelium> stop -name PDau_ret -pdname PDau -retention
Created stop PDau_ret
xcelium> run
14200 NS + 6 (stop PDau_ret: State Retention Rule TESTBENCH.inst.PDau_sr)
TESTBENCH.inst.PDau_sr ./nano.cpf line 44
xcelium> value pcu_inst.pau[1]
```

```
1'h1
xcelium> run
26200 NS + 6 (stop PDau_ret: State Retention Rule TESTBENCH.inst.PDau_sr)
    TESTBENCH.inst.PDau_sr ./nano.cpf line 44
xcelium> value pcu_inst.pau[1]
1'h0
xcelium> stop -disable PDau_ret
xcelium> stop -name PDau_ret_save -pdname PDau -retention -sr_save
Created stop PDau_ret_save
xcelium> run
59 US + 6 (stop PDau_ret_save: State Retention Rule TESTBENCH.inst.PDau_sr)
    TESTBENCH.inst.PDau_sr ./nano.cpf line 44
xcelium> value pcu_inst.pau[1]
1'h1
```

The following command creates a breakpoint on the isolation rule called PDau_iso. Simulation stops when isolation is either enabled or disabled:

```
xcelium> stop -iso_rule PDau_iso
Created stop 1
xcelium> stop -show
1          Enabled          Isolation Rule TESTBENCH.inst.PDau_iso
xcelium> run
13400 NS + 6 (stop 1: Isolation Rule TESTBENCH.inst.PDau_iso is ENABLED)
    TESTBENCH.inst.PDau_iso ./nano.cpf line 54
xcelium> run
28600 NS + 6 (stop 1: Isolation Rule TESTBENCH.inst.PDau_iso is DISABLED)
    TESTBENCH.inst.PDau_iso ./nano.cpf line 54
```

The following command creates a breakpoint on two state retention rules: PDrf_sr and PDau_sr. The -sr_save option is included to specify that simulation should stop when the rule saves its variables:

```
xcelium> stop -sr_rule PDrf_sr PDau_sr -sr_save
Created stop 1
xcelium> stop -show
1          Enabled          State Retention Rule TESTBENCH.inst.PDrf_sr
                                TESTBENCH.inst.PDau_sr (save)
xcelium> run
100600 NS + 6 (stop 1: State Retention Rule TESTBENCH.inst.PDrf_sr, TESTBENCH.inst.PDau_sr)
    TESTBENCH.inst.PDrf_sr (saved) ./nano.cpf line 38
```

Power Mode Transition Breakpoints

In this example, the CPF file defines the power mode transitions with the following create_mode_transition command:

```
create_mode_transition -name mt1 \  
                      -from M3 -to M1 \  
                      -start_condition pmc.mte[1]  
  
create_mode_transition -name mt6 \  
                      -from M3 -to M5 \  
                      -start_condition pmc.mte[6]
```

Power modes M1, M3, and M5 are defined with the following `create_power_mode` commands:

```
create_power_mode -name M1 \  
                 -domain_conditions {pdT@NC_08 pdA@NC_08 pdB@NC_OFF}  
  
create_power_mode -name M3 -default \  
                 -domain_conditions {pdT@NC_12 pdA@NC_12 pdB@NC_12}  
create_power_mode -name M5 \  
                 -domain_conditions {pdT@NC_12 pdA@NC_12 pdB@NC_STANDBY}
```

The following `stop` command creates a breakpoint that stops the simulation when power mode transition `mt1` starts and ends:

```
xcelium> stop -pwr_mode_transition mt1  
Created stop 1  
xcelium> stop -show  
1      Enabled      Power Mode Transition top.mt1  
xcelium> run  
0 clk=0 data=000000 ndata=111111  
...  
95 clk=1 data=001010 ndata=110101  
  
At simulation time 99 NS: Mode transition mt1 [M3->M1] has started.  
At simulation time 99 NS: Power domain pdT has transitioned to nominal condition NC_08  
At simulation time 99 NS: Power domain pdA has started transitioning to nominal condition  
NC_08  
99 NS + 1 (stop 1: Power Mode Transition top.mt1)  
top.mt1 ./dut.cpf line 62
```

The following command stops the simulation when power domain `pdA` transitions from one power mode to another mode:

```
xcelium> stop -pdname pdA -pd_trans  
Created stop 1  
xcelium> run  
...  
At simulation time 99 NS: Power domain pdA has started transitioning to nominal condition  
NC_08  
99 NS + 1 (stop 1: Power Domain top.pdA is in TRANSITION)
```

```
top.pdA ./dut.cpf line 27
```

The following command stops the simulation when power domain `pdB` enters standby mode:

```
xcelium> stop -pdname pdB -pd_standby
Created stop 1
xcelium> run
...
...
At simulation time 814 NS: Mode transition mt6 [M3->M5] has started.
At simulation time 814 NS: Power domain pdB has transitioned to nominal condition
NC_STANDBY
At simulation time 814 NS: Mode transition mt6 [M3->M5] has completed.
814 NS + 1 (stop 1: Power Domain top.pdB is in STANDBY)
top.pdB ./dut.cpf line 32
xcelium>
```

Examples of Other stop Command Modifiers

The following command sequence illustrates the `-show` modifier. The first command creates a source line breakpoint called `break1`; the second creates an object breakpoint called `break2`. The third command shows the status of the two breakpoints:

```
xcelium> stop -line 12 -name break1
Created stop break1
xcelium> stop -object data -name break2
Created stop break2
xcelium> stop -show
break1 Enabled      Line: ./shortdrive.v:12 (scope: top)
break2 Enabled      Object top.data
```

In the following command sequence, breakpoint `break1` is first disabled with the `-disable` modifier and then enabled with the `-enable` modifier:

```
xcelium> stop -show
break1 Enabled Line: ./shortdrive.v:12 (scope: top)
break2 Enabled Object top.data
xcelium> stop -disable break1
xcelium> stop -show
break1 Disabled Line: ./shortdrive.v:12 (scope: top)
break2 Enabled Object top.data
xcelium> stop -enable break1
```

The following command deletes breakpoint `break1`.

```
xcelium> stop -delete break1
```

To disable, enable, or delete the two breakpoints `break1` and `break2`, any of the following commands could

be used:

```
xcelium> stop -delete *1 *2
xcelium> stop -delete break?
xcelium> stop -delete br*
```

The following command displays information on any breakpoint beginning with `v` or `b`:

```
xcelium> stop -show {[vb]*}
```

Creating a Breakpoint

Use the `-create` option, followed by an option that specifies the breakpoint type, with the `stop` command to create various kinds of breakpoints.

Syntax

```
stop [-create]
    -assert [scope_name]
        [{-all | -depth {levels | all | to_cells} | -vhdl }]
    -at_resfunc <net>
        -cv_limit <num>
    -condition {tcl_expression}
    -delta delta_cycle_number [-relative | -absolute]
        [-start delta_cycle_number]
        [-modulo delta_cycle_number]
        [-timestep]
    -iso_rule rule_name [rule_name ...] [-iso_enable | -iso_disable | -iso_corrupt]
    -label "stop_message"
    -line line_number [scope_name]
        [-all]
        [-file filename]
        [-unit unit_name]
    -object object_names
    -pdname power_domain_name [power_domain_name ...]
        [-isolation [-iso_disable | -iso_enable]]
        [-pd_off]
        [-pd_on]
        [-pd_standby]
        [-pd_trans]
        [-retention [-sr_restore | -sr_save | -sr_corrupt]]
    -process process_name
    -pst table_name [table_name...]
    -pwr_mode_transition mode_transition_name [mode_transition_name ...]
    -randomize [-always] [-object_name | -call_id [call_id_number] [-notify]
    -sr_rule rule_name [rule_name ...] [-sr_save | -sr_restore | -sr_corrupt]
```

```
-subprogram subprogram_name
-supply supply_name
    -net [-supply_full_on | -supply_partial_on | -supply_undetermined | -voltage]
-time time_spec [-relative | -absolute [-delbreak 0]]
    [-start time_spec]
    [-modulo time_spec]
-tlmcpu
    [-access <address>]
    [-description <quoted string>]
    [-pc <address>]
    [-physical]
    [-read <address>]
    [-size <bytes>]
    [-write <address>]
[-vsp]
[-continue]
[-delbreak count]
[-esw]
[-execFile file_name]
[-execute command [-noexecout]]
[-if { tcl_expression }]
[-name break_name]
[-silent]
[-skip count]
```

Options

-create

Creates a breakpoint. This modifier is optional. If used, it must be followed by an option that specifies the breakpoint type.


```
-assert [scope_name]  
[{-all | -depth {levels|all|to_cells} |  
-vhdl}]
```

Stops at failures for assertions.

This option lets you define a single breakpoint that is shared by multiple assertions in the design. By default, you can stop at failures for all assertions in a specified scope (or the current debug scope if no scope is specified. For example:

```
xcelium> stop -assert  
xcelium> stop -assert top  
xcelium> stop -assert top.u1
```

This option also supports the following sub-options:

- **-all**: Stops at failures for all assertions in the design hierarchy.

```
xcelium> stop -assert -all
```
- **-depth**: Specifies how many scope levels to descend when searching for assertions. Valid arguments include:
 - **levels**: Descends the specified number of scopes. For example, `-depth 1` means include only the given scope, `-depth 2` means include the given scope and its subscopes, and so on. The default is 1.

```
xcelium> stop -assert -depth 3  
xcelium> stop -assert top.u1 -depth 2
```

- **all**: Includes all scopes in the hierarchy below the given scope.

```
xcelium> stop -assert top.u1 -depth all
```

- **to_cells**: Includes all scopes in the hierarchy below the specified scope(s), but stop at cells (Verilog modules with ``celldefine` or VITAL entities with VITAL Level0 attribute).

- **-vhdl**: Stops at failures for VHDL assertions in the given scope.

```
xcelium> stop -assert -vhdl
```

By using the `-assert` option, you can avoid having to define a whole set of breakpoints on the assertions using separate `stop -object` commands. The `-assert` option can also be used with other `stop` command options, such as `-execute` and `-continue`. For example, the following command sets one breakpoint on all assertions in the design hierarchy with a common script to execute when the breakpoint triggers.

```
xcelium> stop -assert -all -execute {input  
stopscript.tcl}
```

```
-at_resfunc <net>
    -cv_limit <num>
```

Sets a breakpoint when the resolution function value changes for the target userdefined nettype (UDN). The break occurs in the resolution function for any RNM net within the specified scope.

Optionally, you can set the breakpoint on the number of convergence iterations using by the `cv_limit` option. The `cv_limit` value must be a number larger than the expected number of delta cycle iterations required to resolve a time point.

```
-condition {tcl_expression}
```

Sets a breakpoint that triggers when an object referenced in the `tcl_expression` changes value (wires, signals, registers, and variables) or is written to (memories) AND the expression evaluates to true (non-zero, non-x, non-z).

The simulator does not support stop points on individual bits of registers. If a bit-select of a register appears in the expression, the simulator stops and evaluates the expression when any bit of that register changes value. The same holds true for compressed wires.

See [Tcl Expressions as Arguments](#) for details on the format of conditional expressions.

Objects included in a `-condition` expression must have read access. An error is printed if the object does not have read access. See [Access to Simulation Objects](#).

```
-delta delta_cycle_num
[-absolute] [-relative]
[-start delta_cycle_num ]
[-modulo delta_cycle_num ]
[-timestep]
```

Sets a breakpoint that triggers when the simulation delta cycle count reaches the specified delta cycle.

The delta cycle specification can be absolute or relative (the default) based on which:

- If absolute, the breakpoint is automatically deleted after the delta cycle is reached and the breakpoint triggers.
- If relative, the delta cycle specification is an interval, and the breakpoint stops the simulation every *n*th delta cycle.

Use `-start` to specify the absolute delta cycle at which a repetitive breakpoint is to begin firing. If this cycle is before the current cycle, the first stop occurs at the next cycle at which it would have occurred had the stop been set at the cycle specified with `-start`.

The `-modulo` option is similar to `-start`. Use `-modulo` to specify the absolute delta cycle of the first stop cycle for a repeating delta cycle stop. This differs from `-start` only when the given cycle is more than one repeat interval in the future. In this case, the first stop occurs at a delta cycle less than or equal to one interval in the future, such that a stop eventually occurs at the given cycle. For example, if you set a delta breakpoint to stop the simulation every 10 delta cycles, and specify `-modulo 15`, the simulation stops at delta cycle 5, 15, 25, and so on.

The `-timestep` option provides a way to detect infinite loops (due to infinite delta cycles) in the design. This option halts the simulation if the specified number of delta cycles is created at any given simulation time. The simulation halts after the first timestep

delta cycle is reached, and the simulation cannot be advanced. For example:

```
xcelium> stop -delta 1000 -timestep
Created stop 1
xcelium> stop -show
1          Enabled          Delta 1000 (after 1000
deltas at timestep)
xcelium> run
10 NS + 999 (stop 1: delta cycle 1000)
xcelium> run
10 NS + 999 (stop 1: delta cycle 1000)
```

If you want to proceed with the simulation to debug further, you can add the `-delbreak 1` option to the command, as follows:

```
xcelium> stop -delta 1000 -timestep -delbreak 1
```

You can also execute a `stop -delete` command.

```
xcelium> stop -delete breakpoint_name
```

If you want to exit the simulation once you have hit the delta limit, include the `-execute` option to exit the simulation. For example:

```
xcelium> stop -delta 1000 -timestep -execute
exit
```

Because the `-start` and `-modulo` options apply only to repetitive (`-relative`) time stops, you cannot use `-start` or `-modulo` with the `-timestep` option.

When you execute a `save -environment` command to save your debug environment, this option is written to the script to restore your delta breakpoint pattern.

Stops when the specified isolation rule becomes enabled or disabled.

You can specify multiple isolation rule names. The `-iso_rule` option has two sub-options:

- `-iso_disable`: Stop only when the isolation rule becomes disabled.
- `-iso_enable`: Stop only when the isolation rule becomes enabled.

Sets a breakpoint that triggers when the specified line number is about to execute.



To enable the setting of line breakpoints, compile with the `-linedebug` option.

You can use the `stop -line` command to stop at a specified line in one or all instances (module instances or class instances within a given class method).

- **Instance-specific breakpoints:** By default, the breakpoint is set on the specified line in the current debug scope.

```
-iso_rule rule_name [rule_name ...]
[-iso_enable | -iso_disable]
```

```
-line line_number [scope_name]
[-all] [-file filename]
[-unit unit_name]
```

```
xcelium> stop -line 8
```

To set the breakpoint on a line in a different scope, include the *scope_name* on the command line.

```
xcelium> stop -line 8 top.mid1.bot  
xcelium> stop -line 13 myClassInstance
```

To set a breakpoint on a line in a class method for a specific class instance, the following syntax can be used:

- `stop -line line_number class_handle`
xcelium> stop -line 13 c11
- `stop -line line_number class_instance`
xcelium> stop -line 13 @1
- `stop -line line_number instance_handle`
xcelium> stop -line 13 @1_1
- `stop -`
`line line_number [value class_variable]`
xcelium> stop -line 13 [value c11]

- **Non-instance-specific breakpoints:** To create a breakpoint that is not instance-specific, use the `-all` option. The break occurs on all scopes that are instances of the same module or class.

The `-unit unit_name` option specifies a design unit name for non-instance-specific breakpoints. The stop occurs whenever the line number in the specified design unit is about to execute, no matter where in the design hierarchy that unit appears.

The `-file` option specifies the name of the source file that contains the specified line. This is necessary if the scope has multiple source files.

You can use the `-file` option to set a breakpoint on an executable line of code line in a specified file for all occurrences without specifying a scope or design unit name.

```
xcelium> stop -create -line line_number -  
file filename -all
```

For example, the following command stops the simulation whenever line 48 in the specified file is about to execute.

```
xcelium> stop -create -line 48 -file
./ovm/sv/src/base/ovm_packer.sv -all
```

The following limitations apply when setting a line breakpoint in a specified file:

- A breakpoint cannot be created on a line that is a continuation of a line break. In the example below, you can set a breakpoint on line 6, but not on line 7.

```
xcelium> stop -create -line 7 -file leaf.v -
all
```

```
// File leaf.v
module leaf(input reg b);
    reg c;

    task f();
        c = b &          // Line 6
        b;              // Line 7    // Cannot
    set a breakpoint on this line
    endtask

    always @(b)
        f();
endmodule
```

- A breakpoint cannot be created on a line that specifies a `bind` directive.
- A breakpoint cannot be created on a line inside a macro definition. For example:

```
`define PRINT begin : blk \
    $display("%m: b = %d\n", b); \ //
Cannot set a breakpoint on this line
end
```

You can, however, set a breakpoint at the location where the macro is used.

`-object object_name`

Sets a breakpoint that triggers when the specified object changes value (wires, signals, registers, and variables) or is written to (memories). You can also set a breakpoint on an assertion.

By default, vector Verilog wires and VHDL signals are compressed if the model does not require operations on individual bits of the vector. For VHDL, you can set an object breakpoint on a subelement of a compressed vector signal. For Verilog, however, you must elaborate the design with the `-expand` option (`xmelab -expand`) in order to set a breakpoint on a subelement of a compressed vector wire.

You cannot set a breakpoint on a VHDL subprogram object.

The `stop -object` command can be used to monitor changes to SystemVerilog dynamic objects such as class instances, queues, dynamic arrays, associative arrays, and strings.

- For class instances, you can set a breakpoint that triggers when a class instance is created or deleted or when data members are written to. The `stop -object` command takes the following syntax:

```
stop -object classObject
```

where *classObject* takes the following forms:

- Class handle

```
stop -object c11
```
- Instance handle

```
stop -object @5_1
stop -object [value c11]
```
- Class instance

```
xcelium> stop -object @5
```
- Class property

```
stop -object C::staticVar
stop -object @1_1.dynamicVar
```

Breakpoints set using the class instance handle (for example, `stop -object @1_1` or `stop -object @1_1.member`) do not live beyond the life of the specified class object; in other words, breakpoints are garbage collected once a class object is no longer valid. When the instance is no longer valid, the simulator stops and produces a message noting that the stop is no longer valid.

For queues, dynamic arrays, associative arrays, and strings, you can set a breakpoint that triggers when the object is created, deleted, or resized, or when data elements are written to.

You can use wildcard characters in the argument to a `stop -object` command.

- The asterisk (`*`) matches any number of characters.
- The question mark (`?`) matches any one character.

You cannot use wildcard characters inside escaped names.

The object specified as the argument must have read access for the breakpoint to be created. An error is printed if the object does not have read access. See [Access to Simulation Objects](#) for details on specifying access to simulation objects.

You can apply the `stop -object` command to the following SystemC objects: `sc_signal`, `sc_clock`, `sc_in`, `sc_out`, `sc_inout`, `ncsc_viewable`, `sc_fifo`, `sc_fifo_in`, `sc_fifo_out`, `sc_mutex`, `sc_semaphore`, `tlm_fifo`, and `tlm_req_rsp_channel`. For the templated objects in this list, `stop -object` is supported when the object is templated by a C++ data type or by a SystemC built-in data type, or by a user-

defined data type that implements the `<<` operator.

For `sc_port` and `sc_export` objects, `stop -object` is supported if the `sc_object` bound to the port or export supports `stop -object`. The debug scope remains unchanged when the `stop` command is triggered.

For details on using the `stop` command on the SystemC objects `sc_fifo`, `sc_mutex`, `sc_semaphore`, `tlm_fifo`, and `tlm_req_rsp_channel` refer to [Debugging SystemC Models](#).

If an arbitrary SystemC object declares support for value and value change callback (the member methods `ncsc_supports_value()` and `ncsc_supports_value_change_callback()`), you can apply the `stop -object` command to that object. See [Specifying Supported Commands](#).

When you use the `stop -object` command, the stop occurs whenever the value changes. That is, writing the same value to the object does not cause the simulation to stop. If you are simulating your design with `-scSyncEveryDelta` turned on, the `stop -object` command returns control to the Tcl prompt after the delta cycle, in which a value is written to the object. If `-scSyncEveryDelta` is off, the command returns control to the Tcl prompt after all of the delta cycles for SystemC are executed for the current time. The current debug scope does not change when control returns to the Tcl prompt because of an executed `stop` command for a SystemC object.

```
-pdname power_domain_name
[power_domain_name ...
[-isolation
[-iso_disable | -iso_enable]]
[-pd_off] [-pd_on]
[-pd_standby] [-pd_trans]
[-retention [-sr_restore | -sr_save]]
```

Sets a breakpoint that triggers when the specified power domain changes status. If no options are specified, the simulation stops when the power domain is powered down or powered up. You can specify multiple power domain names.

The `-pdname` option has several sub-options that let you create power domain breakpoints that trigger under specific conditions:

- **-pd_off**: Stops when the power domain turns off.
- **-pd_on**: Stops when the power domain turns on. This option stops the simulation when the specified power domain has transitioned to the ON state or to the UNINITIALIZED state. The state of the power domain is UNINITIALIZED when the simulation is being controlled by active state conditions, and when the power domain is on but there are no active state conditions enabled to specify the nominal condition to which the domain should transition.
- **-pd_standby**: Stops when the power domain enters standby mode.
- **-pd_trans**: Stops when the power domain transitions. The breakpoint triggers when the specified power domain starts transitioning from one nominal condition to a different nominal condition.
- **-isolation**: Stops when any isolation rule associated with the power domain is enabled or disabled.
- **-iso_disable**: Stops when any isolation rule associated with the power domain is disabled.
- **-iso_enable**: Stops when any isolation rule associated with the power domain is enabled.
- **-retention**: Stops when any state retention rule associated with the power domain saves or restores its variables.
- **-sr_restore**: Stops when any state retention rule associated with the power domain restores its variables.
- **-sr_save**: Stops when any state retention rule associated with the power domain saves its variables.

```
-process process_name
```

Sets a breakpoint that triggers when the specified VHDL named process starts executing or when it resumes executing after a wait statement.



To enable the setting of line breakpoints, compile with the `-linedebug` option.

```
-pst table_name [table_name...]
```

Sets a breakpoint to stop the simulation when a power state in the specified power state table (PST) changes its value.

This option takes one or more `table_name` values as input. The `table_name` specification is either a full or relative path to the power state table in the current scope.


```
-pwr_mode_transition transition_name  
[transition_name ...]
```

Stops when the specified power mode transition starts and ends.

```
-randomize [object_name] [-always]  
[-call_id call_id_number]  
[-notify]
```

Sets a breakpoint in SystemVerilog `randomize()` function calls.

The SystemVerilog built-in `randomize()` function returns the value 1 for success or 0 for failure. A failure occurs because there are conflicts in the collection of constraints to solve or because a variable is over-constrained.

The `stop -create -randomize` command lets you set a breakpoint in `randomize()` method calls. You can then use other Tcl commands, such as `deposit -constraint_mode`, `deposit -rand_mode`, `constraint`, and `run -rand_solve`, to debug the randomization failures. See [run Command Examples](#) for an example of using these commands.

`stop -randomize` commands are supported for calls to class and scope `randomize` methods.

The `-randomize` option has several sub-options:

- `-always`: By default, simulation stops at the end of `randomize()` calls when the call is about to return 0, or failure. If you include the `-always` option, simulation stops for all `randomize()` calls, regardless of the return status of the call.
You can include an `object_name` argument to stop the simulation in specific `randomize()` calls. The argument can be a class name or a module name. The simulator stops on a failure in any call of the `randomize()` method in the specified module or with the specified class name. If `-always` is specified, the simulator stops in all calls to the `randomize()` method.
- `-call_id <call_id_number>`: To stop the simulation at a particular randomization call number. You must specify the call id number with this option.
- `-notify`: To receive a notification when a `randomize` breakpoint is executed.

The `stop -randomize` command can be used in the following ways:

- When the `stop -randomize` command is used with the `-call_id` option, the simulation stops at a specified `randomize` call in case of a failure. When `stop -randomize -always` command is used with the `-call_id` option, the simulation always stops at the specified `randomize` call irrespective of the return status of the call.
- When the `stop -randomize` is used with the option, you are notified only when a call fails. When the `stop -randomize`

`-always` command is used with the `-notify` option, you are notified for all randomize calls.

- When the `stop -randomize` command is used with both `-call_id` and `-notify` options, you are notified when the randomization fails at the specified randomize call and simulation also stops. When both `-call_id` and `-notify` are used with the `-randomize -always` option, you are always notified at a particular randomize call, and simulation also stops.

```
-sr_rule rule_name [ rule_name ...]  
[-sr_save | -sr_restore]
```

Stops when the specified state retention rule saves or restores its variables.

You can specify multiple state retention rule names.

The `-sr_rule` option has two suboptions:

- `-sr_restore`—Stop only when the state retention rule restores its variables.
- `-sr_save`—Stop only when the state retention rule saves its variables.

```
-subprogram subprogram_name
```

Sets a breakpoint that triggers when the specified VHDL subprogram or Verilog task or function is called.



To enable the setting of subprogram breakpoints, compile with the `-linedebug` option.

```
-supply supply_name -net  
[-supply_full_on |  
-supply_partial_on |  
-supply_undetermined |  
-voltage]
```

Sets a breakpoint that stops the simulation when the named supply changes its state.

To create a breakpoint for a supply net, you must specify the `supply_name` and include the `-net` modifier.

The Tcl `stop -supply supply_name -net` specification supports the following sub-options:

- **`-supply_full_on`**: Stops when the supply net state changes to FULL_ON.
- **`-supply_partial_on`**: Stops when the supply net state changes to PARTIAL_ON.
- **`-supply_undetermined`**: Stops when the supply net state changes to UNDETERMINED.
- **`-supply_off`**: Stops when the supply net state changes to OFF.
- **`-voltage`**: Stops when the voltage of the specified supply net changes its value.

```
-time time_spec
[-absolute] [-relative]
[-start time_spec ]
[-modulo time_spec]
```

Sets a breakpoint that triggers at the specified time. The time can be absolute or relative (the default). Relative time breakpoints are periodic, stopping, for example, every 10 ns. Absolute time breakpoints are, by default, automatically deleted after they trigger. Use the `-delbreak 0` option to prevent an absolute time breakpoint from being deleted after it triggers.

Use `-start` to specify the absolute simulation time at which a relative time breakpoint is to begin firing. If this time is before the current simulation time, the first stop occurs at the next future time at which it would have occurred had the stop been set at the time specified with `-start`.

The `-modulo` option is similar to `-start`. Use `-modulo` to specify the absolute simulation time of the first stop time for a repeating stop. This differs from `-start` only when the given time is more than one repeat interval in the future. In this case, the first stop occurs at a time less than or equal to one interval in the future such that a stop eventually occurs at the given time. For example, if you set a time breakpoint to stop the simulation every 100 ns, and specify `-modulo 250`, the simulation stops at time 50, 150, 250, and so on.

When you execute a `save -environment` command to save your debug environment, this option is written to the script to restore your time breakpoint pattern.

```
-tlmcpu
[-access <address>]
[-description <quoted string>]          [-pc
<address>] [-physical]
[-read <address>] [-size <bytes>]
[-write <address>]
```

Stops when a given CPU access given memory address.

The Tcl `stop -tlmcpu` supports the following sub-options:

- **-access <address>**: Stops when any of the TLM CPUs access (reads or writes) given memory address.
- **-description <quoted string>**: Associates a description with the stop command.
- **-pc <address>**: Stops when TLM CPU reaches given memory address.
- **-physical**: Specifies that a physical address should be used when possible.
- **-read <address>**: Stops when any of the TLM CPUs read given memory address.
- **-size <bytes>**: Specifies the associated size to stop when the TLM CPU accesses (reads or writes) given memory address.
- **-write <address>**: Stops when any of the TLM CPUs write given memory address (default).

```
-vsp
```

Stops after a VSP object access given memory address.

| | |
|---|---|
| <code>-name <i>break_name</i></code> | <p>Specifies a name for the breakpoint. This name can then be used to delete, disable, or enable the breakpoint. If you do not use <code>-name</code>, breakpoints are numbered sequentially.</p> |
| <code>-continue</code> | <p>Resumes the simulation after executing the breakpoint. The simulator does not go into interactive mode.</p> |
| <code>-delbreak <i>count</i></code> | <p>Deletes the breakpoint after it has triggered <i>count</i> number of times.</p> <p>By default, an absolute time breakpoint is deleted after it triggers. Set the <i>count</i> argument to 0 to prevent the absolute time breakpoint from being deleted. For example:</p> <pre>xcelium> stop -time -absolute 2ns -delbreak 0</pre> |
| <code>-esw</code> | <p>Stop after ESW Core object access given memory address</p> |
| <code>-execFile <i>file_name</i></code> | <p>Executes the specified file, housing a list of Tcl commands, when the breakpoint is triggered.</p> <p>This command allows you to run more than one command bundled together in one file. Each command should be listed on its own separate line. Consider the following example:</p> <pre>stop -continue -condition (#a="1") -execFile mycommands.tcl</pre> <p>Whenever the design gets the value <code>a == "1"</code>, Tcl executes the commands written to the file, <code>mycommands.tcl</code>.</p> |
| <code>-execute <i>command</i></code> <code>[-noexecout]</code> | <p>Executes the specified Tcl command when the breakpoint is triggered.</p> <p>If the command that you want to execute requires an argument, enclose the command and its argument in curly braces.</p> <p>You also can specify that you want to execute a list of commands. Separate the commands with a semicolon. Tcl, however, displays only the output of the last command.</p> <p>Use the <code>-noexecout</code> option to suppress output from the Tcl command.</p> |
| <code>-if {<i>tcl_expression</i>}</code> | <p>Sets a condition on the breakpoint. The breakpoint triggers only if the given Tcl boolean expression evaluates to true (non-zero, non-x, non-z). This option can be used with any breakpoint type. See Tcl Expressions as Arguments for more information on the format of the <i>tcl_expression</i> argument.</p> <p>Objects included in an <code>-if</code> expression must have read access. An error is printed if the object does not have read access. See Access to Simulation Objects for details on specifying access to simulation objects.</p> |

`-label "stop_message"`

Specifies a label for the breakpoint message. The breakpoint issues the *stop_message*.

For example:

```
xcelium> stop -line 19 -file testbench.sv -label
"setup phase"
Created stop 1
xcelium> run
0 FS + 0 setup phase
```

The `-name` option can be used with the `-label` option, but it is ignored during the generation of the breakpoint message. For example:

```
xcelium> stop -line 19 -file testbench.v -name
mybreakpoint
Created stop mybreakpoint
xcelium> run
0 FS + 0 (stop mybreakpoint: ./testbench.v:19)
./testbench.v:19          clk_tb = 1'b0;

xcelium> stop -line 19 -file testbench.v -name
mybreakpoint -label "mylabel"
Created stop mybreakpoint
xcelium> run
0 FS + 0 mylabel
./testbench.v:19          clk_tb = 1'b0;
```

`-silent`

Suppresses the display of the message that is printed when a breakpoint triggers.

`-skip count`

Tells the simulator to ignore the breakpoint for the first *count* times that it triggers.

You can use `-skip` to set a breakpoint on the *n*th occurrence of an event; in particular, you can use it to get inside `for` loops.

Examples

The following command creates a breakpoint that stops simulation when `sum` changes value. The `-create` modifier is not required. Because the `-name` option is not included to specify a breakpoint name, *xmsim* assigns a sequential number as the name. This breakpoint is called 1.

```
xcelium> stop -create -object sum
Created stop 1
```

Related Topics

- [Condition Breakpoints](#)
- [Line Breakpoints](#)
- [Object Breakpoints](#)
- [Time Breakpoints](#)
- [Subprogram Breakpoints](#)
- [Specifying Conditions in stop Command Using Tcl Expressions](#)

Specifying Conditions in stop Command Using Tcl Expressions

You can use the `stop` command to specify certain conditions by adding one of two options. Both options require a Tcl expression argument.

- `-condition`: This option specifies that you are creating a condition breakpoint, as opposed to some other kind of breakpoint, such as a time or object breakpoint. A condition breakpoint triggers when any object named in the Tcl expression has an event that would trigger an object breakpoint and the expression evaluates to non-zero, non-x, or non-z.
- `-if`: This option can be used with any breakpoint type, including condition breakpoints. The Tcl expression argument is evaluated, and the stop triggers if the expression evaluates to non-zero, non-x, or non-z.


The simulator does not support breakpoints on individual bits of registers. If a bit-select of a register appears in the expression, the simulator stops and evaluates the expression when any bit of that register changes value. The same holds true for compressed wires.

You cannot set a breakpoint on an object in a VHDL subprogram.

Objects included in a conditional expression must have read access. An error is printed if the object does not have read access. For details on specifying access to simulation objects, see [Access to Simulation Objects](#).

There are two general rules to keep in mind when writing the Tcl expression:

- You must enclose the expression in braces to suppress immediate substitution of values.
`{tcl_expression}`

 If you are setting the breakpoint using the SimVision analysis environment, these braces are included on the Set Breakpoint form.

- In the following example, the value of `sig[1]` would be substituted with its current value (`1'b0`, for example) if there were no braces. No object would be named in the expression by the time

the stop command routine sees it, resulting in an error.

```
stop -condition #sig[1] == 1      ;# Error. Expression is not in curly braces.
stop -condition {#sig[1] == 1}    ;# Correct syntax
```

- You must use either an explicit `value` command or the shorthand notation for getting the value of an object (using the pound sign (#) character) to get the object's value into the expression parser because the parser does not understand names. For example, the following command generates an error message:

```
stop -time 100 ns -if {sig == 1}
```

In the following examples, an explicit value command or the # notation is used to get the object's value:

Verilog:

```
stop -time 100 ns -if {[value sig] == 4'b0001}    ;# Explicit value command
stop -time 100 ns -if {#sig == 4'b0001}           ;# Using # notation
```

VHDL:

```
stop -time 100 ns -if {[value sig] == `1'}        ;# Explicit value command
stop -time 100 ns -if {#sig == `1'}               ;# Using # notation
```

Format specifiers can be used with either the `value` command or the # sign. If you use the # sign, place the format specifier after the # sign. For example,

Verilog:

```
stop -condition {[value %d out] == 12}
stop -condition {#%dout = 12}
```

VHDL:

```
stop -condition {[value %d out] == 12}
stop -condition {#%dout == 12}
```

In a Tcl expression, the single-equality operator (`=`), which is used for assignment in Verilog, is a logical comparison operator. In a Tcl expression, `=` is the same as the Verilog logical comparison operator (`==`). The following two expressions are the same:

```
{ #top.load = 1'b1 }
{ #top.load == 1'b1 }
```

These operators return the unknown value (x) if either operand is unknown. In a conditional expression, an unknown result is treated as false. For example, in the following `stop` command, the expression returns an unknown result, and is, therefore, false. This conditional breakpoint will not trigger when the signal `top.load` has the value x.

```
stop -create -condition {#top.load == 1'bx}
```

To set a breakpoint that will stop when the value is `x`, use the case equality operator (`===`). The result of the expression is always 1 (true) or 0 (false). For example:

```
xcelium> stop -create -condition {#top.load === 1'bx}
```

Hierarchy Separators

If you specify the path to the object, you can use the period (`.`), the colon (`:`), or the slash (`/`) interchangeably as the hierarchy separator. See [Hierarchy Separators in Tcl Commands](#) for details. Here are some examples:

- **Absolute Paths:** For absolute paths, you can use the standard Verilog or VHDL notation:

- If the top level is Verilog, the path starts with the name of the top-level Verilog module.

```
stop -condition {#vlog_top.counter.value == ...}
stop -condition {#vlog_top:counter.value == ...}
stop -condition {#vlog_top/counter/value == ...}
```

- If the top level is VHDL, the path starts with a colon.

```
stop -condition {#:counter.value == ...}
stop -condition {#:counter.value == ...}
stop -condition {#:counter/value == ...}
```

You can also use a language-neutral syntax in which an absolute path begins with a `/`.

- If the top level is Verilog, include the name of the top-level Verilog module after the slash.

```
stop -condition {#/vlog_top/counter/value == ...}
```

- If the top level is VHDL, include the name of the top-level VHDL entity after the slash.

```
stop -condition {#/vhdl_top/counter/value == ...}
```

- **Relative Paths:** The following examples assume that the current scope is the top-level of the design:

```
stop -condition {#counter.value == ...}
stop -condition {#counter:value == ...}
stop -condition {#counter/value == ...}
```

Specifying a Value using Hierarchy Separators

In the examples shown in this section, all examples use the shorthand `#` notation for getting the value of the object, and specify an absolute path using the `/` hierarchy separator.

Verilog

For Verilog, you can use the standard Verilog notation for specifying the value:


```
stop -condition {#/vlog_top/count == 4'b1011}  
stop -condition {#/vlog_top/count == 4'hb}  
stop -condition {#/vlog_top/count == 11}  
stop -condition {#/vlog_top/count[0] == 1}
```

VHDL

For VHDL, you must enclose single-bit entities in single quotation marks and vectors in quotation marks. For example:

```
stop -condition {#/vhdl_top/clock == `1'  
stop -condition {#/vhdl_top/count == "01010101"}
```

For both Verilog and VHDL, you can use the following language-independent syntax for specifying the value:

radix#number

The `tcl_relaxed_literal` variable must be set to 1 to enable this functionality.

For Verilog, valid radices are 2, 8, 10, and 16. For VHDL, valid radices are 2, 8, and 16.

For example:

```
set tcl_relaxed_literal 1  
stop -condition {#/top/count == 2#11}  
stop -condition {#/top/count == 8#11}  
stop -condition {#/top/count == 10#11}    ;# Decimal radix not supported for VHDL  
stop -condition {#/top/count == 16#11}  
  
stop -condition {#%b/top/count == 2#11}  
stop -condition {#%o/top/count == 8#11}  
stop -condition {#%d/top/count == 10#11}  ;# Decimal radix not supported for VHDL  
stop -condition {#%x/top/count == 16#11}
```

The *radix#number* syntax cannot be used in an expression. For example, the following is not valid:

```
{#/top/count == 2#11 * 2}
```

Related Topic

- [Creating a Breakpoint](#)

Detecting Infinite Loops

Use the `-timestep` option with the `stop` command to set a delta breakpoint and detect infinite loops (due to infinite delta cycles) in a design.

Syntax

```
stop -delta delta_cycle_number -timestep
```

For example:

```
xcelium> stop -delta 1000 -timestep
```

Without the `-timestep` option, a `stop -delta` command sets a breakpoint that triggers when the simulation delta cycle count reaches the specified delta cycle. For example, the following command stops the simulation at every third delta cycle:

```
xcelium> stop -delta 3
Created stop 1
xcelium> run
1 NS + 2 (stop 1: delta cycle 3)
xcelium> run
5 NS + 0 (stop 1: delta cycle 6)
xcelium> run
5 NS + 3 (stop 1: delta cycle 9)
xcelium> run
9 NS + 0 (stop 1: delta cycle 12)
xcelium>
```

The `-timestep` option halts the simulation if the specified number of delta cycles is created at any given simulation time. The simulation halts after the first timestep delta cycle is reached. For example,

```
xcelium> stop -delta 1000 -timestep

Created stop 1
xcelium> stop -show
1      Enabled      Delta 1000 (after 1000 deltas at timestep)
xcelium> run
10 NS + 999 (stop 1: delta cycle 1000)
```

If the breakpoint triggers, the simulation will not advance. For example:

```
xcelium> stop -delta 1000 -timestep
Created stop 1
xcelium> run
10 NS + 999 (stop 1: delta cycle 1000)
xcelium> run
10 NS + 999 (stop 1: delta cycle 1000)
```

If you want to proceed with the simulation to debug further, you can:

- Add the `-delbreak 1` option to the command.

```
xcelium> stop -delta 1000 -timestep -delbreak 1
```

- Execute a `stop -delete` command.

```
xcelium> stop -delete breakpoint_name
```

If you want to exit the simulation once you have hit the delta limit, include the `-execute` option to exit the simulation. For example,

```
xcelium> stop -delta 1000 -timestep -execute exit
```

Setting a Source Code Line Breakpoint

You can use the `stop` command to set a source code line breakpoint at a specified line in your HDL file to halt the simulation. This type of breakpoint is usually set when you want to simulate to a certain point and then single-step through lines of code. To enable the line breakpoint feature in Xcelium:

1. First, you must compile your simulation with the `-linedebug` option.
2. Then, to set a line breakpoint, use the Tcl `stop` command with the `-line` option.

Related Topic

- [Deactivating, Enabling, Deleting, and Displaying Breakpoints](#)

Setting an Object Breakpoint

Use the `-object` option with the `stop` command to set an object breakpoint and halt your simulation when a specified object changes value (wires and signals) or when it is written to (registers, memories, variables). This type of breakpoint is usually set when you want the simulation to stop every time the signal changes value or when you want to see the value of signals when some condition is true (for example, on every positive edge of the clock).

The object specified as the argument must have read access for the breakpoint to be created. An error is printed if the object does not have read access.

By default, vector Verilog wires and VHDL signals are compressed if the model does not require operations on individual bits of the vector. For VHDL, you can set an object breakpoint on a subelement of a compressed vector signal. For Verilog, however, you must elaborate the design with the `-expand` option (`xmelab -expand`) in order to set a breakpoint on a subelement of a compressed vector wire.

You cannot set a breakpoint on an object in a VHDL subprogram.

Related Topic

- [Access to Simulation Objects](#)

Setting a Delta Breakpoint

Use the `-delta` option with the `stop` command to halt your simulation when the simulation delta cycle count reaches a specified delta cycle.

Related Topic

- [Deactivating, Enabling, Deleting, and Displaying Breakpoints](#)

Setting a VHDL Process Breakpoint

Use the `-process` option with the `stop` command to halt your simulation when a named VHDL process starts executing or resumes executing after a `wait` statement.

You must compile with the `-linedebug` option to enable the setting of source line and process breakpoints.

Related Topic

- [Deactivating, Enabling, Deleting, and Displaying Breakpoints](#)

Setting a Subprogram Breakpoint

Use the `-subprogram` option with the `stop` command to halt the simulation when a named subprogram (procedure or function, Verilog task or function) starts executing.

You must compile with the `-linedebug` option to enable the setting of source line, process, and subprogram breakpoints.

Related Topic

- [Deactivating, Enabling, Deleting, and Displaying Breakpoints](#)

Deactivating, Enabling, Deleting, and Displaying Breakpoints

After setting breakpoints in your design, you can display information on all breakpoints, deactivate breakpoints, enable previously deactivated breakpoints, and delete breakpoints.

Use the Tcl `stop` command with the `-show`, `-disable`, `-enable`, or `-delete` modifier. The argument to these modifiers can be:

- A break name or a list of break names

- A pattern

The asterisk (*) matches any number of characters

The question mark (?) matches any one character

[*characters*] matches any one of the characters

- Any combination of literal break names and patterns

strobe

The Tcl `strobe` command prints the values of specified VHDL, Verilog, or SystemC objects based on one of the following specifications:

- When a specified condition is true.
- When a specified signal changes value.
- At a specified time interval.

The `strobe` command is a Tcl procedure that uses the `stop` command to set a condition, object, or time breakpoint and then, when the breakpoint triggers, executes a `value` command to print out the values of the specified objects in tabular format. In order to use this command, Simulation objects in the design must have read access. Be aware that you can only set one strobe at a time. Hence, setting a new strobe automatically deletes the previous strobe.

For SystemC, you can apply this command only to objects in the SystemC portion of the design that are derived from `sc_signal`, `sc_port`, `sc_export`, `sc_in`, `sc_inout`, `sc_clock`, or `ncsc_viewable`.

After you specify this command, the simulator prints the values of the specified objects using the default format of the `value` command.

Optionally, you can choose to specify a custom format for individual objects by enclosing the object-format pair in curly braces. For example:

```
xcelium> strobe -time 100 clk {data %b}
```

You can also set the `strobeFmt` variable to specify a global format. For example:

```
xcelium> set strobeFmt %b
```

If a strobe is already set, you must reset the strobe in order for the change to take effect.

The simulator does not print a header with the tabular output if you interrupt or stop the simulation (with `CTRL-C`, an `assert` statement, or by running the simulation for a specified period of time) and then continue the simulation. This is done so that if you send the output to a file with the `-redirect` option, the header does not appear in the output file every time you continue the simulation. However, if you choose to send output to the screen, you might want to redisplay the header. To display the header again, set

the `strobeHeader` variable to 1, as follows:

```
xcelium> set strobeHeader 1
```

The variable `strobeTimeWidth` can be used to control how much space is used to print the simulation time in the strobe output. By default, the value is 15. To change this, set the `strobeTimeWidth` variable before creating the strobe. For example:

```
xcelium> set strobeTimeWidth 25
```

If a strobe is already set, you must reset the strobe in order for this change to take effect.

The `strobe` command defines other Tcl variables that are internal to the operation of the command. The following variables should not be changed:

- `strobeObjects`
- `strobeObjectList`
- `strobeStream`
- `headerList`
- `widthList`
- `valueList`

strobe Command Syntax

```
strobe <strobe_specification> <object_list> [<output_file_specification>]  
      -condition <condition_specification> |  
      -object <object_specification> |  
      -time <time_specification>  
[-redirect <filename> [-append]]  
[-delete]
```

strobe Command Options

This section describes the options that you can use with the Tcl `strobe` command.

| | |
|---|--|
| <code><strobe_specification></code> | <p>Specifies a condition for the strobe. When using this command you must choose one of the three available methods to display signal values for simulation objects. Xcelium supports displaying values when:</p> <ul style="list-style-type: none"> • a specified condition is true (<code>-condition</code>) • the signal of an object monitor changes its value (<code>-object</code>), or • at a specified time interval (<code>-time</code>). |
| <code><object_list></code> | <p>Defines a list of one or more simulation objects using the syntax shown:</p> <pre><object> {object_format} [<object> {object_format}...]</pre> |
| <code><output_file_specification></code> | <p>Saves output to the specified file. By default, the simulator prints the output of the strobe command to the screen.</p> |
| <code>-condition <condition_specification></code> | <p>Displays the values of the specified object(s) defined in the object list when the <i>condition_specification</i> evaluates to true. This argument is the same as the one used in the stop command.</p> <p>For example, the following command displays the values of the specified signals when <i>data</i> has the value 3, decimal:</p> <pre>xcelium> strobe -condition {[value %d top.data] = 3} clk clr data q</pre> |
| <code>-object <object_specification></code> | <p>Displays the values of the specified object(s) defined in the object list when an explicit <i>object_specification</i> changes value.</p> <p>With this option, you can specify only one object to monitor for a change in value. For example, the following command displays the values of <i>y</i> and <i>z</i> when <i>x</i> changes value:</p> <pre>xcelium> strobe -object x y z</pre> <p>The following command displays the value of <i>y</i> in the default format and the value of <i>z</i> in binary when <i>x</i> changes value:</p> <pre>xcelium> strobe -object x y {z %b}</pre> |
| <code>-time <time_specification></code> | <p>Displays the values of the object(s) defined in the object list at the given <i>time_specification</i>.</p> <pre>xcelium> strobe -time 100 ns x y z xcelium> strobe -time 100 ns x {y %b} z</pre> |
| <code>-redirect filename [-append]</code> | <p>Redirects output to the specified file. Include the <code>-append</code> sub-option to append the output to the file specified instead of overwriting it.</p> |
| <code>-delete</code> | <p>Deletes the strobe.</p> |

strobe Command Examples

The following command prints the values of signals `clk`, `clr`, `data`, and `q` every 100 ns.

```
xcelium> strobe -time 100 clk clr data q
Setting up strobe time - '100'
xcelium> run
Time          |clk| |clr| |data| |q|  |
-----
100 NS        |1'h1| |1'h1| |4'h0| |4'hx| |
200 NS        |1'h1| |1'h1| |4'h1| |4'h0| |
300 NS        |1'h1| |1'h1| |4'h2| |4'h1| |
400 NS        |1'h1| |1'h1| |4'h3| |4'h2| |
...
```

The following command prints the values of signals `clk`, `clr`, `data` (in binary), and `q` every 100 ns.

```
xcelium> strobe -time 100ns clk clr {data %b} q
Setting up strobe time - '100ns'
xcelium> run 300 ns
Time          |clk| |clr| |data| |q|  |
-----
100 NS        |1'h1| |1'h1| |4'b0000| |4'hx| |
200 NS        |1'h1| |1'h1| |4'b0001| |4'h0| |
300 NS        |1'h1| |1'h1| |4'b0010| |4'h1| |
Ran until 300 NS + 0
```

In the following command sequence, the simulation is run for 300 ns. When the simulation is resumed, the strobe header is not displayed. The `strobeHeader` variable is then set to display the header when the simulation is resumed.

```
xcelium> strobe -time 100 clk clr data q
Setting up strobe time - '100'
xcelium> run 300
Time          |clk| |clr| |data| |q|  |
-----
100 NS        |1'h1| |1'h1| |4'h0| |4'hx| |
200 NS        |1'h1| |1'h1| |4'h1| |4'h0| |
300 NS        |1'h1| |1'h1| |4'h2| |4'h1| |
Ran until 300 NS + 0
xcelium> run 300
400 NS        |1'h1| |1'h1| |4'h3| |4'h2| |
500 NS        |1'h1| |1'h1| |4'h4| |4'h3| |
600 NS        |1'h1| |1'h1| |4'h5| |4'h4| |
Ran until 600 NS + 0
xcelium> set strobeHeader 1
1
xcelium> run 300
```



```
Time          |clk |clr |data |q    |
-----
700 NS        |1'h1 |1'h1 |4'h6 |4'h5 |
800 NS        |1'h1 |1'h1 |4'h7 |4'h6 |
900 NS        |1'h1 |1'h1 |4'h8 |4'h7 |
Ran until 900 NS + 0
```

To change the format globally, you can set the `strobeFmt` variable, as shown in the following example. You must reset the strobe after setting the `strobeFmt` variable:

```
xcelium> strobe -time 100 clk clr data q
Setting up strobe time - '100'
xcelium> run 300
Time          |clk |clr |data |q    |
-----
100 NS        |1'h1 |1'h1 |4'h0 |4'hx |
200 NS        |1'h1 |1'h1 |4'h1 |4'h0 |
300 NS        |1'h1 |1'h1 |4'h2 |4'h1 |
Ran until 300 NS + 0
xcelium> set strobeFmt %b
%b
xcelium> strobe -time 100 clk clr data q
Setting up strobe time - '100'
xcelium> run 300
Time          |clk |clr |data    |q    |
-----
400 NS        |1'b1 |1'b1 |4'b0011 |4'b0010 |
500 NS        |1'b1 |1'b1 |4'b0100 |4'b0011 |
600 NS        |1'b1 |1'b1 |4'b0101 |4'b0100 |
Ran until 600 NS + 0
```

The following command displays the values of `clk`, `clr`, and `q` when `data` changes value.

```
xcelium> strobe -object data clk clr q
Setting up strobe object - 'data'
xcelium> run 300 ns
Time          |clk |clr |q    |
-----
0 NS          |1'h0 |1'h1 |4'hx |
100 NS        |1'h1 |1'h1 |4'hx |
200 NS        |1'h1 |1'h1 |4'h0 |
Ran until 300 NS + 0
```

The following command displays the values of `data`, `clk`, `clr`, and `q` when `data` changes value.

```
xcelium> strobe -object data data clk clr q
Setting up strobe object - 'data'
xcelium> run 300 ns
```

```

Time           |data |clk  |clr  |q    |
-----
0 NS           |4'h0 |1'h0 |1'h1 |4'hx |
100 NS         |4'h1 |1'h1 |1'h1 |4'hx |
200 NS         |4'h2 |1'h1 |1'h1 |4'h0 |
Ran until 300 NS + 0

```

The following command displays the values of `data` (in binary), `clk`, `clr`, and `q` (in hex) when `data` changes value. Notice that the object-format pair is enclosed in curly braces.

```
xcelium> strobe -object data {data %b} clk clr q
```

```
Setting up strobe object - 'data'
```

```
xcelium> run 300 ns
```

```

Time           |data      |clk  |clr  |q    |
-----
0 NS           |4'b0000 |1'h0 |1'h1 |4'hx |
100 NS         |4'b0001 |1'h1 |1'h1 |4'hx |
200 NS         |4'b0010 |1'h1 |1'h1 |4'h0 |
Ran until 300 NS + 0

```

The following command displays the values of the specified signals when `data` has the value 3, decimal. The signal `data` is available from the top level of the hierarchy.

Verilog:

```
xcelium> strobe -condition {[value %d top.data] = 3} clk clr data q
```

VHDL:

```
xcelium> strobe -condition {[value %d :data] = 3} clk clr data q
```

If you are currently at the top level, you can omit the hierarchical path specification to `data`, and write the two commands shown in the previous example as follows:

```
xcelium> strobe -condition {[value %d data] = 3} clk clr data q
```

Instead of using the `value` command to get the value of `data` into the expression evaluator, you can use `#data`. Include the format specifier after the `#` sign.

```
xcelium> strobe -condition {#%ddata = 3} clk clr data q
```

The following command displays the values of the specified signals when `data` has the value 0010 (binary). For VHDL, you must enclose vectors in quotation marks.

Verilog:

```
xcelium> strobe -condition {#data = 4'b0010} clk clr data q
```

VHDL:

```
xcelium> strobe -condition {#data ="0010"} clk clr data q
```

The following command displays the values of the specified signals when bit 0 of `data` is 1. The

expression is evaluated when any bit of `data` changes value. For VHDL, you must enclose single-bit entities in single quotation marks.

Verilog:

```
xcelium> strobe -condition {#data[0] == 1} clk clr data q
```

VHDL:

```
xcelium> strobe -condition {#data(0) == `1'} clk clr data q
```

The following command is identical to the previous command except that it uses an explicit `value` command to get the value of `data` (bit 0) into the expression parser.

Verilog:

```
xcelium> strobe -condition {[value %b data[0]] == 1'b1} clk clr data q
```

VHDL:

```
xcelium> strobe -condition {[value %b data(0)] == `1'} clk clr data q
```

The following command displays the values of the specified signals if the value of `top.load` has the value `x`. Notice that in the Tcl expression, the case-equality operator (`===`) is used. For this operator, bits that are unknown are included in the comparison, and the result of the expression is always 1 (true) or 0 (false).

```
xcelium> strobe -condition {#top.load === 1'bx} clk clr data q
```

In the following command, the logical comparison operator (`=` or `==`) is used. These operators return the unknown value (`x`) if either operand is unknown. In a conditional expression, an unknown result is treated as false. Therefore, the following command does *not* stop the simulation when the signal has the value `x`.

```
xcelium> strobe -condition {#top.load == 1'bx} clk clr data q
```

Related Topics

- [Access to Simulation Objects](#)
- [stop](#)
- [value](#)

task

The Tcl `task` command schedules Verilog tasks for execution during a simulation run. Verilog tasks are those tasks that are implemented in the Verilog source code with a `task` declaration. You cannot use this command to schedule the execution of built-in or user-defined Verilog system tasks or functions, or VHDL functions or procedures.

Xcelium schedules the specified task(s) to execute at some point during the current simulation time. The specified task(s) do not execute immediately but after you resume the simulation. The simulator may execute other behaviors that are already scheduled for the current simulation time before executing a scheduled task.

Notes:

- When scheduling tasks that may be called elsewhere (from within the Verilog code, from VPI, or from Tcl), remember that, because tasks in Verilog are static, multiple concurrent calls to tasks can interfere with each other, stomping on the values of parameters and registers within the task.
- Parameters to tasks are static registers. You must set the input and inout parameters before the scheduled task runs by using a `deposit` command. You can schedule the task before or after you deposit values to the input and inout parameters. Remember that intervening calls to the same task in the model can interfere with and overwrite the values that you deposited to the task parameters before the scheduled task actually runs.
- You can read the value of output parameters after the task has completed by using the `value` command.

To deposit values to parameters and to read output parameters, the parameters must have read/write access. See [Access to Simulation Objects](#) for details on specifying access to simulation objects.

- Disable statements in the Verilog code affect active tasks that you scheduled with the task command in the same way that they affect tasks that are scheduled by the code in the model.
- Once a task is scheduled, it becomes part of the model's state. If you save a snapshot with a task that is scheduled or that is in progress, the task is still scheduled or in progress when you restart that snapshot. Tasks that are scheduled or that are in progress at the time of the save are the only tasks that are scheduled or in progress after the restart.

task Command Syntax

```
task [-schedule] <task_name> [<task_name>...]
```

task Command Options

This section describes the options that you can use with the Tcl `task` command.

```
-schedule  
<task_name> [<task_name>.  
..]
```

Schedules the specified task(s) for execution at the current simulation time.

Use this option to specify that one or more tasks begin executing after the simulation is resumed, and at some point during the current simulation time. Other behaviors that are already scheduled for the current simulation time may execute before the scheduled task starts.

task Command Examples

The Verilog source code is used in the examples in this section is shown below:

```
module top;  
  reg out;  
  reg i1,i2,i3;  
  reg clock;  
  initial  
    #10000 $display("DONE");  
  always  
    @(posedge clock) print_task1(out,i1,i2,i3);  
  
  task print_task1;  
    output [31:0] out;  
    input [31:0] i1;  
    input [31:0] i2;  
    input [31:0] i3;  
    begin  
      out = i1 + i2 + i3;  
      $display("%t %0d %0d %0d %0d", $time,out,i1,i2,i3);  
    end  
  endtask  
  
  task print_task2;  
    output [31:0] out;  
    input [31:0] i1;  
    input [31:0] i2;  
    input [31:0] i3;  
    begin  
      out = i1 * i2 * i3;  
      $display("%t %0d %0d %0d %0d", $time,out,i1,i2,i3);  
    end  
  endtask  
endmodule
```

The following sequence of commands schedules the task named `print_task1` for execution and sets the input parameter values that are in effect when the task runs. The `run` command continues the simulation so

that the task is executed:

```
xcelium> run 50
xcelium> task -schedule print_task1
xcelium> deposit print_task1.i1 1
xcelium> deposit print_task1.i2 1
xcelium> deposit print_task1.i3 1
xcelium> run 1
      50 3 1 1 1
Ran until 51 NS + 0
```

The following command specifies the full path to the task. The `-schedule` modifier is not required:

```
xcelium> task top.print_task1
```

The following sequence of commands illustrates how to schedule more than one task, specify the task names on the command line, and then deposit values to the input and inout parameters.

```
xcelium> task print_task1 print_task2
xcelium> deposit print_task1.i1 1
xcelium> deposit print_task1.i2 1
xcelium> deposit print_task1.i3 1
xcelium> deposit print_task2.i1 1
xcelium> deposit print_task2.i2 1
xcelium> deposit print_task2.i3 1
xcelium> run 1
      0 3 1 1 1
      0 1 1 1 1
Ran until 1 NS + 0
```

The following command shows how to schedule a task at a specific time. For instance, you could set a time breakpoint, advance the simulation to that time, and then schedule the task.

```
xcelium> stop -time 50 -absolute
xcelium> run
xcelium> task print_task2
xcelium> (deposit values to parameters)
xcelium> run
```

In the following sequence of commands, a line breakpoint is set on the first statement in the task so that the simulation stops when the task is about to execute. Parameter values are then deposited immediately before the execution of the task:

```
xcelium> stop -line 18
xcelium> run
xcelium> deposit i1 0
xcelium> deposit i2 1
xcelium> deposit i3 1
xcelium> run 1
      100 2 0 1 1
```

```
Ran until 101 NS + 0
```

In the following sequence of commands, a line breakpoint is set on the first statement in the task. After depositing values to the input parameters, a `run -return` command is issued. This stops the simulation when the task returns so that you can read output parameters as soon as they are available:

```
xcelium> stop -line 18
Created stop 1
xcelium> run
100 NS + 0 (stop 1: ./test2.v:18)
./test2.v:18 begin
xcelium> deposit i1 1
xcelium> deposit i2 0
xcelium> deposit i3 0
xcelium> run -return
100 1 1 0 0
./test2.v:10 @(posedge clock) print_task1(out,i1,i2,i3);
```

tcheck

The Tcl `tcheck` command controls timing check messages and notifier updates for a specified Verilog instance. The specified Verilog instance can be an instance of a Verilog module instantiated in VHDL. Use the `-depth` option to apply this command to all subscopes beneath the specified instance, or to an explicit subscope level. Use wildcards with this command to enable hierarchical and partial-name matching. For instance, you can:

- Use the asterisk (*) to match one or more characters in any instance within the current scope.
- Use the question mark (?) to match a single character in any instance within the current scope.

tcheck Command Syntax

```
tcheck instance_path
  -depth all | <level>
  -message
  -notifier
  -off
  -on
  -type <timing_check_type>
  -verbose
```

If you do not explicitly specify either `-message` or `-notifier`, then both options are considered implicitly by the Tcl `tcheck` command.

tcheck Command Options

This section describes the options that you can use with the Tcl `tcheck` command.

| | |
|--|--|
| <code>-message</code> | Controls the display of timing check messages. |
| <code>-depth all <level></code> | <p>Controls timing check messages and notifier updates from the depth specified below the given instance.</p> <p>This option requires that you specify one of two possible arguments:</p> <ul style="list-style-type: none">• all: Enables the <code>tcheck</code> command to update all subscopes in the hierarchy below the named instance.• <level>: Enables the <code>tcheck</code> command to update the subscopes to a specified level below the named instance. The specified level value must be greater than 0. If a level value of 1 is specified, the <code>tcheck</code> command controls the specified instance only, meaning that <code>tcheck -depth 1</code> and <code>tcheck</code> (without <code>-depth</code>) are synonymous. If a level value of 0 is specified, Xcelium generates a TCHGTAR error. <p>You can only specify one <code>-depth</code> option per <code>tcheck</code> command. If multiple <code>-depth</code> options are specified, a TCHMARG error results.</p> <p>To print a list of all updated subscope instances, use this option with <code>-verbose</code>.</p> |
| <code>-notifier</code> | Controls the display of timing check notifiers. |
| <code>-off</code> | <p>Turns off timing check messages and notifier updates. This is the default.</p> <p>If you want to turn off check messages for a selective bit/net, you must also specify the <code>xrun/xmelab</code> option, <code>-enable_tcl_tcheck_net</code> during elaboration.</p> |
| <code>-on</code> | <p>Turns on timing check messages and notifier updates.</p> <p>If you want to turn on check messages for a selective bit/net, you must also specify the <code>xrun/xmelab</code> option, <code>-enable_tcl_tcheck_net</code> during elaboration.</p> |
| <code>-type <timing_check_type></code> | <p>Controls the display of timing check messages and notifier updates for the specified <code>timing_check_type</code>.</p> <p>This option supports the following types:</p> <ul style="list-style-type: none">• setup |

- hold
- setuphold
- recovery
- removal
- recrem
- skew
- timeskew
- fullskew
- period
- width
- nochange

With Xcelium, specifying multiple `-type` options to a single `tcheck` command is allowed.

When specifying multiple `tcheck` commands, if there are conflicts in applying the timing check rules, the last `tcheck` command specified takes precedence.

If any timing check is disabled on the `xmelab` command line using the `-tfile` option, that timing check is ignored by the Tcl `tcheck` command.

The following rules apply when using `$setuphold` and `$recrem`:

- When `$setuphold` is in the HDL source file, a `-type setup` or `-type hold` statement controls the `$setup` or `$hold` part of the timing check. However, if `$setup` or `$hold` is in the HDL source file, a `-type setuphold` statement to control `$setuphold` timing checks does not affect `$setup` or `$hold`.
- Similarly, when `$recrem` is in the HDL source file, a `-type recovery` or `-type removal` statement controls the `$recovery` or `$removal` part of the timing check. However, if `$recovery` or `$removal` is in the HDL source file, a `-type recrem` statement to control `$recrem` timing checks does not affect `$recovery` or `$removal`.

`-verbose`

Enables a report of informational messages on instances of wildcard substitution.

tcheck Command Examples

The following examples illustrate the various use cases of the `tcheck` command.

Example 1

The following sequence of commands turns off all timing check messages and notifier updates for instance `top.y1.u2`, runs the simulation for 1000 ns, and then turns the timing check messages and notifier updates back on:

```
xcelium> tcheck -off top.y1.u2
xcelium> run 1000 ns
xcelium> tcheck -on top.y1.u2
```

Example 2

The following command turns off all timing check messages and notifier updates for instance `:U_DFF`, an instance of a Verilog module instantiated in VHDL:

```
xcelium> tcheck -off :U_DFF
```

Example 3

The following example disables all timing check messages for `top.I1`, except for those corresponding to `$setup`. The setup timing checks for instance `top.I1` are all be enabled:

```
xcelium> tcheck top.I1 -off -message
xcelium> tcheck top.I1 -on -type setup
```

Example 4

The following example disables all timing checks for `$setuphold`, but in this case the `$setup` part of the `$setuphold` (as well as any other `$setup` timing checks) is enabled for instance `top.I1`:

```
xcelium> tcheck top.I1 -off -type setuphold
xcelium> tcheck top.I1 -on -type setup
```

Example 5

The following example disables all timing checks starting from instance `TB`, which is the root of the design.

The `-verbose` option enables the simulator to report on all updated subscope instances.

```
xcelium> tcheck TB -depth all -verbose
Scope updated : TB
Scope updated : TB.minst
Scope updated : TB.minst.binst
```

Example 6

The following example disables multiple timing check types using one `tcheck` command. Both `$setup` and `$width` are disabled for instance `top.I1`:

```
xcelium> tcheck top.I1 -off -type setup -type width
```

Example 7

This last example shows how to use the `tcheck` command to selectively control the display of messages and notifier updates for certain timing checks. The first command disables `$setup` timing check message updates for instance `top.I1`. The second command disables `$setup` timing check notifier updates for instance `top.I2`:

```
xcelium> tcheck top.I1 -off -type setup -message
xcelium> tcheck top.I2 -off -type setup -notifier
```

Example 8

Consider the following simple test.

```
test_setup_simple.v:
=====
module BOT(out, in, clk);
input in, clk;
output out;
and an(out, in, clk);
specify
    $setup(in, posedge clk, 15);
endspecify
endmodule

module MID(out, in, clk);
input in, clk;
output out;
BOT binst(out, in, clk);
specify
    $setup(in, posedge clk, 15);
endspecify
endmodule

module TB;
reg in, clk;
```

```
wire out;

MID minst (out, in, clk);

initial begin
    $monitor($stime,,,,in,,,clk,,,out);
    in=1;
    clk=1;
    #10 in = ~in;
    #10 clk = ~clk;
    #10 in = ~in;
    #10 clk = ~clk;
end
endmodule
```

When running this example, Xcelium reports timing violations for scopes `TB.minst` and `TB.minst.binst`.

```
xcelium> run
0 1 1 1
10 0 1 0
20 0 0 0
30 1 0 0

Warning! Timing violation
$setup( in:30 NS, posedge clk:40 NS, 15 : 15 NS );
File: ./test_setup_simple.v, line = 6
Scope: TB.minst.binst
Time: 40 NS

Warning! Timing violation
$setup( in:30 NS, posedge clk:40 NS, 15 : 15 NS );
File: ./test_setup_simple.v, line = 15
Scope: TB.minst
Time: 40 NS

40 1 1 1
xmsim: *W,RNQUIE: Simulation is complete.
```

After specifying the command `tcheck TB.minst` and running the simulation again, Xcelium turns off the timing check warnings and notifier updates for scope `TB.minst` only.

```
xcelium> tcheck TB.minst
xcelium> run
0 1 1 1
10 0 1 0
20 0 0 0
30 1 0 0

Warning! Timing violation
$setup( in:30 NS, posedge clk:40 NS, 15 : 15 NS );
```

```
File: ./test_setup_simple.v, line = 6
```

```
Scope: TB.minst.binst
```

```
Time: 40 NS
```

```
40 1 1 1
```

```
xmsim: *W,RNQUIE: Simulation is complete.
```

If `-depth all` is included when specifying the above `tcheck` command, Xcelium disables timing check warnings and notifier updates for all subscopes below `TB.minst`.

```
xcelium> tcheck TB.minst -depth all
```

```
xcelium> run
```

```
0 1 1 1
```

```
10 0 1 0
```

```
20 0 0 0
```

```
30 1 0 0
```

```
40 1 1 1
```

```
xmsim: *W,RNQUIE: Simulation is complete.
```

Example 9

Consider this test which uses wildcard characters and an instance of arrays in the subscope hierarchy.

```
test.v:
```

```
=====
```

```
module AND(out, in, clk);
```

```
input in, clk;
```

```
output out;
```

```
assign out = in & clk;
```

```
specify
```

```
    $setup(in, posedge clk, 15);
```

```
    $hold(posedge clk, in, 20);
```

```
endspecify
```

```
endmodule
```

```
module MID_1(out, in, clk);
```

```
input in, clk;
```

```
output out;
```

```
AND an(out, in, clk);
```

```
specify
```

```
    $setup(in, posedge clk, 15);
```

```
    $hold(posedge clk, in, 20);
```

```
endspecify
```

```
endmodule
```

```
module MID(out, in, clk);
```

```
input in, clk;
```

```
output out;
```

```
MID_1 m_1[1:0](out, in, clk);
specify
    $setup(in, posedge clk, 15);
    $hold(posedge clk, in, 20);
endspecify
endmodule

module DUT(out, in, in1, clk);
input in, in1, clk;
output out;

MID m11(out1, in, clk);
MID m22(out2, in1, clk);
or (out, out1, out2);
specify
    $setup(in, posedge clk, 15);
    $hold(posedge clk, in, 20);
endspecify
endmodule

module TB;
reg in, in1, clk;
wire out;

DUT d1 (out, in, in1, clk);

initial begin
    $monitor($stime,,,,in,,,in1,,,clk,,,out);
    in=1;
    in1 = 0;
    clk=1;
    #10 in = ~in;
    in1 = ~in1;
    #10 clk = ~clk;
    #10 in = ~in;
    in1 = ~in1;
    #10 clk = ~clk;
    #10 in = ~in;
    in1 = ~in1;
    #10 clk = ~clk;
end
endmodule
```

In this example, after specifying the command `tcheck TB.d?.m1* -verbose -type hold -message` and running the simulation, Xcelium only turns off `$hold` warning messages in scope `TB.d1.m11`.

```
xcelium> tcheck TB.d?.m1* -verbose -type hold -message
Instance TB.d1.m11 matched wildcard pattern TB.d?.m1*
xcelium> run
```

```
0 1 0 1 1
10 0 1 1 1
20 0 1 0 0
30 1 0 0 0
```

```
Warning! Timing violation
$setup( in:30 NS, posedge clk:40 NS, 15 : 15 NS );
File: ./test.v, line = 8
Scope: TB.d1.m11.m_1[1].an
Time: 40 NS
```

```
Warning! Timing violation
$setup( in:30 NS, posedge clk:40 NS, 15 : 15 NS );
File: ./test.v, line = 18
Scope: TB.d1.m11.m_1[1]
Time: 40 NS
```

```
Warning! Timing violation
$setup( in:30 NS, posedge clk:40 NS, 15 : 15 NS );
File: ./test.v, line = 8
Scope: TB.d1.m11.m_1[0].an
Time: 40 NS
```

```
Warning! Timing violation
$setup( in:30 NS, posedge clk:40 NS, 15 : 15 NS );
File: ./test.v, line = 18
Scope: TB.d1.m11.m_1[0]
Time: 40 NS
```

```
...
```

```
40 1 0 1 1
```

```
Warning! Timing violation
$hold( posedge clk:40 NS, in:50 NS, 20 : 20 NS );
File: ./test.v, line = 9
Scope: TB.d1.m11.m_1[1].an
Time: 50 NS
```

```
Warning! Timing violation
$hold( posedge clk:40 NS, in:50 NS, 20 : 20 NS );
File: ./test.v, line = 19
Scope: TB.d1.m11.m_1[1]
Time: 50 NS
```

```
Warning! Timing violation
$hold( posedge clk:40 NS, in:50 NS, 20 : 20 NS );
File: ./test.v, line = 9
Scope: TB.d1.m11.m_1[0].an
Time: 50 NS
```

```
Warning! Timing violation
$hold( posedge clk:40 NS, in:50 NS, 20 : 20 NS );
File: ./test.v, line = 19
Scope: TB.d1.m11.m_1[0]
Time: 50 NS
```

...

```
50 0 1 1 1
60 0 1 0 0
xmsim: *W,RNQUIE: Simulation is complete.
```

If **-depth 2** is included when specifying the above **tcheck** command, Xcelium turns off **\$hold** warning messages for **TB.d1.m11** and subscopes **TB.d1.m11.m_1[1]** and **TB.d1.m11.m_1[0]**.

```
xcelium> tcheck TB.d?.m1* -verbose -type hold -message -depth 2
Instance TB.d1.m11 matched wildcard pattern TB.d?.m1*
Scope updated : TB.d1.m11
Scope updated : TB.d1.m11.m_1[1]
Scope updated : TB.d1.m11.m_1[0]
xcelium> run
0 1 0 1 1
10 0 1 1 1
20 0 1 0 0
30 1 0 0 0
```

```
Warning! Timing violation
$setup( in:30 NS, posedge clk:40 NS, 15 : 15 NS );
File: ./test.v, line = 8
Scope: TB.d1.m11.m_1[1].an
Time: 40 NS
```

```
Warning! Timing violation
$setup( in:30 NS, posedge clk:40 NS, 15 : 15 NS );
File: ./test.v, line = 18
Scope: TB.d1.m11.m_1[1]
Time: 40 NS
```

```
Warning! Timing violation
$setup( in:30 NS, posedge clk:40 NS, 15 : 15 NS );
File: ./test.v, line = 8
Scope: TB.d1.m11.m_1[0].an
Time: 40 NS
```

```
Warning! Timing violation
$setup( in:30 NS, posedge clk:40 NS, 15 : 15 NS );
File: ./test.v, line = 18
Scope: TB.d1.m11.m_1[0]
Time: 40 NS
```



```
Warning! Timing violation
$setup( in:30 NS, posedge clk:40 NS, 15 : 15 NS );
File: ./test.v, line = 28
Scope: TB.d1.m11
Time: 40 NS

...

40 1 0 1 1

Warning! Timing violation
$hold( posedge clk:40 NS, in:50 NS, 20 : 20 NS );
File: ./test.v, line = 9
Scope: TB.d1.m11.m_1[1].an
Time: 50 NS

Warning! Timing violation
$hold( posedge clk:40 NS, in:50 NS, 20 : 20 NS );
File: ./test.v, line = 9
Scope: TB.d1.m11.m_1[0].an
Time: 50 NS

Warning! Timing violation
$hold( posedge clk:40 NS, in:50 NS, 20 : 20 NS );
File: ./test.v, line = 9
Scope: TB.d1.m22.m_1[1].an
Time: 50 NS

Warning! Timing violation
$hold( posedge clk:40 NS, in:50 NS, 20 : 20 NS );
File: ./test.v, line = 19
Scope: TB.d1.m22.m_1[1]
Time: 50 NS

Warning! Timing violation
$hold( posedge clk:40 NS, in:50 NS, 20 : 20 NS );
File: ./test.v, line = 9
Scope: TB.d1.m22.m_1[0].an
Time: 50 NS

...

50 0 1 1 1
60 0 1 0 0
xmsim: *W,RNQUIE: Simulation is complete.
```

Use `-depth all` to turn off all `$hold` warning messages for `TB.d1.m11` and its subsscopes, as shown.

```
xcelium> tcheck TB.d?.m1* -verbose -type hold -message -depth all
```

time

The Tcl `time` command displays the current simulation time, scaled to the specified unit. The unit can be:

- A time unit that you specify.
- `auto`: Uses the largest base unit that makes the numeric part of the time an integer.
- `module`: Uses the timescale of the current debug scope.

The simulation time can be displayed in the following time units:

- `fs`: femtoseconds
- `ps`: picoseconds
- `ns`: nanoseconds
- `us`: microseconds
- `ms`: milliseconds
- `sec/s`: seconds
- `min`: minutes
- `hr`: hours

You can also display the time at 10 or 100 times the base unit. If no unit is given, the value of the `$display_unit` variable is used. This variable is set to `auto` by default.

To perform conversion, comparison, and arithmetic operations on objects of type `time`, include `-operation`.

time Command Syntax

```
time [[10|100] time_unit|auto|module]
    [-delta]
    [-nount]
    [-operation [-addtime <time> <time>] | [-divtime <time> <time>] /
        [-eqtime <time> <time>] | [-gtetime <time> <time>] |
        [-gttime <time> <time>] | [-int2time <intHi32> <intLo32>] |
        [-ltetime <time> <time>] | [-lttime <time> <time>] |
        [-multime <time> <time>] | [-neqtime <time> <time>] |
        [-real2time <real>] | [-scaletime <time> <scale_factor>] |
        [-subtime <time> <time>] | [-time2real <time>]]
```

time Command Options

This section describes the options that you can use with the Tcl `time` command.

| | |
|-------------------------|--|
| <code>-delta</code> | <p>Includes the delta cycle count.</p> <p>At any given simulation time, values of nets are first updated and then behaviors that are sensitive to those nets are executed. This two step process may be repeated any number of times because of zero-delays. The delta cycle count represents the number of times the process is repeated for the given simulation time.</p> |
| <code>-nounit</code> | <p>Does not include the time unit.</p> |
| <code>-operation</code> | <p>Performs conversion, comparison, or arithmetic operations on objects of type <code>time</code>.</p> |

This option supports the following syntax, where you specify the operation that you want to perform by including a sub-option and arguments:

```
-operation <operation_option> <arguments>
```

The sub-options supported by Xcelium, and their corresponding arguments, are listed below:

- **Conversion Operations**

- Convert two 32-bit pieces to one 64-bit integer.
`time -operation -int2time intHi32 intLo32`
- Convert a real number to time using the current time scale.
`time -operation -real2time real`
- Convert time to a real number using the current time scale.
`time -operation -time2real time`
- Return the value of time multiplied by a scale factor.
`time -operation -scaletime time scale_factor`

- **Comparison Operations**

- Evaluate for equal.

```
time -operation -eqtime time time
```

- Evaluate for not equal.

```
time -operation -neqtime time time
```

- Evaluate for greater than.

```
time -operation -gttime time time
```

- Evaluate for greater than or equal.

```
time -operation -gtetime time time
```

- Evaluate for less than.

```
time -operation -lttime time time
```

- Evaluate for less than or equal.

```
time -operation -ltetime time time
```

- **Arithmetic Operations**

- Add two time values.

```
time -operation -addtime time time
```

- Subtract two time values.

```
time -operation -subtime time time
```

- Divide two time values.

```
time -operation -divtime time time
```

- Multiply two time values.

```
time -operation -multitime time time
```

time Command Examples

Consider this `xmsim` command, which loads a design snapshot `worklib.board:module` in interactive mode. The Tcl `run` command runs the simulation for 100 ns.

```
% xmsim -nocopyright -tcl board
```

```
Loading snapshot worklib.board:module ..... Done
```

```
xcelium> run 100 ns
```

```
5 count= X, f=x, af=x
Ran until 100 NS + 0
```

The following command displays the current simulation time in ns.

```
xcelium> time ns
100 NS
```

The following command displays the current simulation time in fs.

```
xcelium> time fs
100000000 FS
```

The following command displays the current simulation time in 100 times the base unit of fs.

```
xcelium> time 100fs
1000000 100FS
```

The following commands illustrate the `auto` keyword, which displays the time using the largest base unit that makes the numeric part of the time an integer.

```
xcelium> time fs
100000000 FS
xcelium> time auto
100 NS
```

The following command displays the current simulation time using the timescale of the current debug scope.

```
xcelium> time module
100 NS
```

The following command displays the current simulation time using the timescale of the current debug scope and including the delta cycle count.

```
xcelium> time module -delta
100 NS + 0
```

The following command displays the current simulation time with no time unit.

```
xcelium> time -nounit
100
```

The following shows the output of various `time -operation` commands.

```
xcelium> time -operation -addtime 50 ns 30 ns      ;# Add two time values
80 NS
```

```
xcelium> time -operation -addtime 5 100us 3 10ns
500030 NS
```

```
xcelium> time -operation -multitime 30 ns 30 ns    ;# Multiply two time values
900 MS
```

```
xcelium> time fs -operation -multitime 30 ns 30 ns ;# Show output in FS
```

9000000000000000 FS

```
xcelium> time -operation -multitime 5 ns 3 10ps
150 US
```

```
xcelium> time -operation -divtime 300 ns 30 ns      ;# Divide two time values
10 FS
```

```
xcelium> time -operation -divtime 500 10us 50 10ns
10 PS
```

```
xcelium> time -operation -subtime 100 ns 200 ns    ;# Subtract two time values
-100 NS
```

```
xcelium> time -operation -subtime 50 10us 30 10ns
499700 NS
```

```
xcelium>
xcelium> ;# Convert two 32-bit pieces to one 64-bit integer
```

```
xcelium> time -operation -int2time 100 200
429496729800 FS
```

```
xcelium> ;# Convert real number to time using current time scale
```

```
xcelium> time -operation -real2time 34.00
34 FS
```

```
xcelium> ;# Convert time to real number using current time scale
```

```
xcelium> time -operation -time2real 300
300.00
```

```
xcelium> ;# Return value of time multiplied by scale factor
```

```
xcelium> time -operation -scaletime 300 ns 2
600 NS
```

```
xcelium> time -operation -scaletime 50 100ps 2
10 NS
```

```
xcelium>
```

```
xcelium> time -operation -eqtime 1 ns 1 ps      ;# Evaluate for equal
0
```

```
xcelium> time -operation -eqtime 50 10us 50000 10ns
1
```

```
xcelium> time -operation -neqtime 1 ns 1 ps     ;# Evaluate for not equal
1
```

```
xcelium> time -operation -neqtime 50 10us 50 fs
1
```

```
xcelium> time -operation -lttime 1 ns 1 ps      ;# Evaluate for less than
0
```

```
xcelium> time -operation -lttime 50 10us 30 10us
0
```

```
xcelium> time -operation -ltetime 1 ns 1 ps    ;# Evaluate for less than or equal
0

xcelium> time -operation -ltetime 30 10us 30 100s
1

xcelium> time -operation -gttime 1 ns 1 ps      ;# Evaluate for greater than
1

xcelium> time -operation -gttime 80 100fs 30 10ns
0

xcelium> time -operation -gtetime 1 ns 1 ps     ;# Evaluate for greater than or equal
1

xcelium> time -operation -gtetime 300 10ns 30 100ns
1
```

value

The Tcl `value` command prints the current value of the specified objects using the last format specifier preceding the object name argument. If no format is specified, a default format is used.

If the object is an analog branch, you can specify whether you want to display the flow of the branch by including the `-flow` option, or its potential by including the `-potential` option. If neither option is specified, the potential is displayed.

Objects specified as arguments to the `value` command must have read access. An error is printed if an object does not have read access.

You can apply the `value` command to the following SystemC objects: `sc_signal`, `sc_clock`, `sc_in`, `sc_out`, `sc_inout`, `ncsc_viewable`, `sc_fifo`, `sc_fifo_in`, `sc_fifo_out`, `sc_mutex`, `sc_semaphore`, `tlm_fifo`, `tlm_req_rsp_channel`, and SystemC processes. For the templated objects in this list, `value` is supported when the object is templated by a C++ data type or by a SystemC built-in data type, or by a user-defined data type that implements the `<<` operator. For `sc_port` and `sc_export` objects, `value` is supported if the `sc_object` bound to the port or export supports `value`.

For details on using the `value` command on the SystemC objects `sc_fifo`, `sc_mutex`, `sc_semaphore`, `tlm_fifo`, and `tlm_req_rsp_channel`, see [Debugging SystemC Models](#).

value Command Syntax

```
value [formats] [pot_flow specifier] object_name [object_name ...]  
    -attr <attribute name>  
    -classlist [class_definition]  
    -dereference  
    -flow  
    -fullpath  
    -keys associative_array  
    -lps  
        -newline  
    -packed  
    -potential  
    -pst  
        -active  
        -newline  
    -saved  
    -signed  
    -twocomp  
    -verbose
```

value Command Options

This section describes the options that you can use with the Tcl `value` command.

formats

You can use the following format specifiers to specify the formats:

- %c: character
- %s: string
- %b: binary
- %d: unsigned decimal

Use the `-signed` option to print the value as a signed decimal value.

- %o: octal
- %x: hexadecimal
- %h: same as %x
- %f: floating-point number
- %e: real number in mantissa-exponent form
- %g: use %e or %f, whichever is shorter
- %t: decimal time scaled from the timescale of the object's module to the simulation's timescale
- %v: strength value—wires only

To revert to the default format, use %.

If no format is specified, the default format depends on the object type. The following are the default formats:

- %d: time
- %d: integer
- %g: real
- \$vlog_format87: reg
- \$vlog_format: wire

The \$vlog_format is a predefined Tcl variable that defaults to %h. You can set this variable to %b, %o, or %d.

For VHDL, values are returned in a format that resembles the appropriate VHDL syntax for the object type. If one of the radix format specifiers (%b, %o, %d, or %x) is given, the format affects the format of integer values and of `bit` and `std_logic` values. The value is shown without any added VHDL syntax, such as double or single quotes. Otherwise, the format specifier is ignored for VHDL values. The VHDL literal format (%v) is the default format. For example:

```
xcelium> value :vector
"101010111100"
xcelium> value %b :vector
101010111100
xcelium> value %h :vector
ABC
xcelium> value :enum
'1'
xcelium> value %d :enum
1
xcelium> value %b :enum
1
```

You can use wildcard characters in the argument to a `value` command.

- The asterisk (*) matches any number of characters.
- The question mark (?) matches any one character.

You cannot use wildcard characters inside escaped names.

`-attr <attribute
name>`

Prints the value given in the user-defined attribute name.

`-classlist
[class_definition]`

Prints a list of class instances in instance handle format (for example, @l_1).

By default, the `-classlist` option produces a list of all class instance handles regardless of the current debug scope. Only class instances that exist at the time the command is issued are listed.

Include the `class_definition` argument if you want to list the class instances with a common definition.

See [value Command Examples](#) for an example.

`-dereference`

Prints the content value of the object referenced by a class variable.

| | |
|--------------------------------------|--|
| <code>-flow</code> | Returns the flow value of analog branches that are specified on the command line immediately following the <code>-flow</code> option. Returns the flow value of analog objects that have existing Tcl current probes. This option is ignored for any other kind of object. |
| <code>-fullpath</code> | Print the full hierarchical name of the object when combined with <code>-verbose</code> . |
| <code>-keys associative_array</code> | <p>Prints a list of the keys associated with the specified SystemVerilog associative array object.</p> <p>When an associative array object is provided as the argument, the <code>value</code> command returns the size of the array (number of elements). For example, the following command tells you that there are two elements in the array:</p> <pre>xcelium> value aa 2</pre> <p>The <code>value -keys</code> command lets you obtain a list of indices or keys for an associative array.</p> <pre>xcelium> value -keys aa 4 15</pre> <p>The format of the generated list is compatible with the Tcl <code>list</code> command, which lets you manipulate the result through the Tcl programming language. See value Command Examples for an example.</p> |
| <code>-lps</code> | <p>Gets the value for a low-power simulation object.</p> <p>Supply net format is: {state, voltage}</p> <p>Supply set format is: {state name, simstate, pwell, nwell, deeppwell, deepnwell}</p> <p>You can use the <code>-newline</code> option with the <code>value -lps</code> command to display each value on a separate line, if more than one supply set is specified when using a wildcard character.</p> |

`-packed`

Prints the value of an unpacked array of any dimension as packed array.

If this option is used, all the bits of the unpacked array are concatenated and form a Verilog number literal. Concatenation of elements of an array is always done from left to right, regardless of the direction of the array.

Currently, this option is supported only for the following:

- Unpacked Verilog static arrays
- Numeric output formats
 - binary (`%b`)
 - octal (`%o`)
 - decimal (`%d`)
 - hex (`%x` or `%h`)
 - signed (`-signed` option in the `value` command)
 - 2's complement (`-twocomp` option in the `value` command)

For unsupported array types and output formats, the `-packed` option will not have any effect.

See [-packed command examples](#) for more information on using the `-packed` option.

Limitations

- If the total number of bits in an array is more than 64, the following commands may produce incorrect results:
 - `value -packed -signed arr`
 - `value -packed -twocomp arr`
- For an array dimension greater than 1, using `-packed` option on an unpacked array with default output format may not produce the same output as for a packed array.

`-potential`

Returns the potential of analog branches that are specified on the command line immediately following the `-potential` option. This option is ignored for any other kind of object.

`-pst`

Prints the value for a LPS IEEE 1801 Power State Table.

You can use the following option with the `value -pst` command:

- **-active**: Prints the value of only the active states of the Power State Table.
- **-newline**: Lists objects in a new line instead of the default single line output.

`-saved`

Displays the saved value of a state retention variable.

In a low-power simulation, the state of sequential elements (registers, latches, flip-flops) in a switchable power domain must be saved and retained for the entire shutoff period. When the power domain is powered back up, the saved states must be written back into the registers.

To ensure that a powered-down block resumes normal operation after power up, these sequential elements can be replaced by special state retention cells. The design registers or instances that must be replaced with state retention cells, the control signals, and the conditions that control the save and restore operation of the retention registers is specified by:

- A `set_retention` command in the IEEE 1801 file.
- A `create_state_retention_rule` command in the CPF file.

Use the `-saved` option to print the saved value of a state retention variable. For example, if SR1 is a state retention register in the current scope, the following command prints the current value of the variable, which is x if the domain is powered down:

```
xcelium> value SR1
```

The following command returns the saved value.

```
xcelium> value -saved SR1
```

Using the `-saved` option on non-state retention variable generates an error.

`-signed`

Prints the current value of each specified object as a signed decimal value in sign-magnitude form.

This option overrides any format specified with the command.

The `-signed` option is ignored if the value cannot be converted to signed decimal.

`-twocomp`

Prints the current value of each specified object as a signed decimal value in two's complement form.

This option overrides any format specified with the command.

The `-twocomp` option is ignored if the value cannot be converted to signed decimal.

`-verbose`

Prints the name of the object and its current value in name=value format.

This option is useful if you are displaying the values of several signals or if you are using wildcard characters. For example:

```
xcelium> value *
4'ha 1'h0 1'h0 1'h0
xcelium> value -verbose *
count=4'ha clock=1'h0 f=1'h0 af=1'h0
xcelium> value %d -verbose *f
f=1'd0 af=1'd0
```

The `-verbose` option must precede the object name(s) on the command line.

value Command Examples

The following command displays the value of the signal `data`:

```
xcelium> value top.u1.data
4'h2
```

If the current debug scope has been set to instance `u1`, you can display the value of `data` with the following command:

```
xcelium> value data
4'h2
```

The following command displays the value of all objects in the current scope that have names that start with the letter `c`:

```
xcelium> value c*
4'ha 1'h0
```

The following command displays the value of all objects in the current scope that have names that are six characters long and that start with `rst` and end with `p5`. The `-verbose` option is included so that the signal names are included in the output:

```
xcelium> value -verbose rst?p5
rst1p5=1'h0 rst2p5=1'h0 rst3p5=1'h0
```

The following sequence of `value` commands display the current value of `data` in a variety of formats:

```
xcelium> value data
4'h2
xcelium> value %o data
4'o02
xcelium> value %b data
4'b0010
xcelium> value %d data
4'd2
xcelium> value %g data
2
xcelium> value %f data
2.000000
xcelium> value %e data
2.000000e+00
xcelium> value %b data %d q
4'b0010 4'd1
xcelium> value % data %d data %b data
4'h2 4'd2 4'b0010
```

The following example illustrates the `-saved` option. In this example, the CPF `create_power_domain` command creates a power domain called `PDau`, which contains instance `alu_inst/au1`. The CPF `create_state_retention_rule` command specifies that register `alu_inst.au1.z` is to be replaced with a state retention cell. The current value of the register is saved when the `save_edge` expression changes from `false` to `true`. The saved value is restored when the `restore_edge` condition changes from `false` to `true`:

```
create_power_domain -name PDau \  
                    -instances alu_inst/au1 \  
                    -shutoff_condition {pcu_inst/pau[2]}  
  
create_state_retention_rule -name PDau_sr \  
                            -instances {alu_inst/au1/z* } \  
                            -save_edge {pcu_inst/pau[1]}  
                            -restore_edge {!pcu_inst/pau[1]}
```

The following example illustrates the `-signed` and `-twocomp` options:

```
wire [3:0] count;  
  
xcelium> deposit count 4'b0110  
xcelium> value count  
4'h6  
xcelium> value -signed count  
6  
xcelium> value -twocomp count  
6  
xcelium> deposit count 4'b1010  
xcelium> value count  
4'ha  
xcelium> value -signed count  
-2  
xcelium> value -twocomp count  
-6
```

The following example uses the `-keys` option to get a list of keys for a SystemVerilog associative array, and illustrates how you can manipulate the output of the command using the Tcl programming language:

```
module top;  
    int i;  
    int aa [ int ];  
    // Load address .eq. content  
    initial  
        for (i=0; i<10; i++) begin  
            aa[ i ] = i;  
        end  
endmodule
```

```
xcelium> value -keys aa  
0 1 2 3 4 5 6 7 8 9  
xcelium> foreach idx [ value -keys aa ] {  
> puts -nonewline "AA( $idx ) = "  
> puts [ value aa[$idx] ]  
> }
```

```
AA( 0 ) = 0
AA( 1 ) = 1
AA( 2 ) = 2
AA( 3 ) = 3
AA( 4 ) = 4
AA( 5 ) = 5
AA( 6 ) = 6
AA( 7 ) = 7
AA( 8 ) = 8
AA( 9 ) = 9
xcelium>
```

The following command includes the `-classlist` option, which generates a list of class instances handles:

```
module top;
  class A; int value; endclass
  class B; int neg; endclass
  int i;
  A cla [];
  B clb [];
  initial begin
    cla = new[3];
    clb = new[3];
    for (i=0; i<3; i++) begin
      cla[i] = new;
      clb[i] = new;
    end
  end
endmodule
```

```
xcelium> run 1 ns
Ran until 1 NS + 0
xcelium> ;# List all instance handles
xcelium> value -classlist
@3_1 @4_1 @5_1 @6_1 @7_1 @8_1
xcelium> ;# List instance handles for class B
xcelium> value -classlist B
@4_1 @6_1 @8_1
xcelium> ;# Use the instance handle list with the describe command
xcelium> describe [value -classlist top.A]
top.A@3_1...handle class top.A {
    int value = 0
}
top.A@5_1...handle class top.A {
    int value = 0
}
top.A@7_1...handle class top.A {
```

```
        int value = 0
    }
xcelium>
```

The following command shows the error message that is displayed when you run in regression mode (no read, write, or connectivity access to simulation objects) and then use the `value` command on an object that does not have read access.

```
xcelium> value clk
xmsim: *E,OBJACC: Object must have read access: clk.
```

In the following VHDL example, the `stack -show` command displays the current subprogram call stack. The process (`process1`) is displayed as nest-level 0, the base of the stack. The subprogram `function1` is `:process1[1]`, and the subprogram `function2` is `:process1[2]`.

- The first `value` command displays the value of the variable `tmp5` in the current debug scope.
- The second `value` command displays the value of the variable `var1` in `:process1`.
- The third `value` command displays the value of the signal `tmp4` in `:process1[1]` (that is, in `function1`).
- A `scope` command is then executed to set the scope to `function1`. The last `value` command displays the value of `tmp4`.

```
xcelium> stop -subprogram function1
Created stop 1
xcelium> run
0 FS + 0 (stop 1: Subprogram :function1)
./test.vhd:36 tmp4_local := function2 (tmp4);
xcelium> run -step
./test.vhd:29 tmp5_local := tmp5 + 1;
xcelium> stack -show                ;# Display the current call stack
2: Scope: :process1[2] Subprogram:@work.e(a):function2
   File: /usr1/belanger/xm/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 29

1: Scope: :process1[1] Subprogram:@work.e(a):function1
   File: /usr1/belanger/xm/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 36

0: Scope: :process1
   File: /usr1/belanger/xm/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 52
xcelium> scope -show                ;# Display the current debug scope
Directory of scopes at current scope level:

Current scope is (:process1[2])
```


Highest level modules:

Top level VHDL design unit:

entity (e:a)

VHDL Package:

STANDARD

ATTRIBUTES

std_logic_1164

TEXTIO

```
xcelium> value tmp5                ;# Display the value of tmp5 in :process1[2]
                                     ;# i.e., function2

1
xcelium> value :process1:var1        ;# Display the value of var1 in :process1
'0'
xcelium> value :process1[1]:tmp4     ;# Display the value of tmp4 in :process1[1]
                                     ;# i.e., function1

1
xcelium> scope -set :process1[1]     ;# Set scope to function1 (:process1[1])
xcelium> value tmp4                 ;# Display the value of tmp4

1
```

The following example shows you how to use the `value` command to display the contents of a VHDL record. You can do this by appending `.all` to the object name:

For example, suppose that a record is defined as follows:

```
type MailboxBlock_Typ is record
    Data      : Data_Typ;
    command   : Command_Typ;
    NextP     : MailboxPntr_Typ;  -- next pointer
    PrevP     : MailboxPntr_Typ;  -- previous pointer
end record;
```

The following variable is defined:

```
variable MB_Temp : MailboxPntr_Typ;
```

The following values are assigned to the variable:

```
MB_Temp          := new MailboxBlock_Typ; -- new object
MB_Temp.Data      := "1010";
MB_Temp.command   := READ;
MB_Temp.NextP     := null;
MB_Temp.PrevP     := MB_Head;
```

Now if you set a breakpoint and simulate, the contents of the record can be viewed with the following command:

```
xcelium> value MB_Temp.all
("1010", READ, 0x3f88, 0x0)
```

The following example show how to use the `-newline` option to output the same `VSS1` and `VSS2` supply set values shown above to separate lines, which can improve readability:

```
xcelium> value -lps -verbose -newline tb.dut.VSS*  
VSS1={FULL_ON, 1.200000}  
VSS2={OFF, 0.000000}
```

`-packed` command examples

The following examples show you how to use `value` command to print the value of an unpacked array of any dimension as packed array, using the `-packed` option:

```
module top();

    reg upk[0:3];
    reg upk_2d[0:3][0:3];
    reg [0:3]pk;
    reg [0:3][0:3]pk_2d;

    reg [0:3]mix[0:3][0:3];

    reg big[0:1023];

    wire [1:0][3:0] wpk_rl;
    wire [0:1][0:3] wpk_lr;

    wire wupk_rl [1:0][3:0];
    wire wupk_lr [0:1][0:3];

    struct
    {
        reg[0:3] r_pk;
        reg r_upk[0:3];

        struct
        {
            reg[0:3] ir_pk;
            reg ir_upk[0:3];
        } in_struct;
    } out_struct;

    /** Unsupported array types **/

    string s = "str"; // Strings
    reg dyn_arr[]; // Dynamic array
    reg queue[$] = {1, 0, 1, 1}; // Queue
    reg aa[int]; // Associative array

    initial begin
        dyn_arr = new[3];
        dyn_arr = '{1, 0, 1, 1};

        aa = '{1:1, 2:0, 3: 1, 4:1};
    end
endmodule
```

xcelium> **value upk_2d**

((1'h1,1'h1,1'h1,1'h1), (1'h1,1'h1,1'h1,1'h1), (1'h1,1'h1,1'h1,1'h1), (1'h1,1'h1,1'h1,1'h1))

```
xcelium> value -packed upk_2d
```

```
16'hffff
```

```
xcelium> value %o upk_2d
```

```
((1'o1,1'o1,1'o1,1'o1), (1'o1,1'o1,1'o1,1'o1), (1'o1,1'o1,1'o1,1'o1), (1'o1,1'o1,1'o1,1'o1))
```

```
xcelium> value -packed %o upk_2d
```

```
16'o177777
```

```
// Assume all bits set to 1
reg [0:3]pk;
reg [0:3][0:3] pk_2d;

reg upk[0:3];
reg upk_2d[0:3][0:3];
```

```
xcelium> value %b upk
```

```
(1'b1, 1'b1, 1'b1, 1'b1)
```

```
xcelium> value -packed %b upk
```

```
4'b1111
```

```
xcelium> value %h upk
```

```
(1'h1, 1'h1, 1'h1, 1'h1)
```

```
xcelium> value -packed %h upk
```

```
4'hf
```

```
xcelium> value -packed %b upk_2d
```

```
16'b1111111111111111
```

Related Topics

- [Wildcards Characters in Tcl Commands](#)
- [Displaying Low-Power Values \(IEEE 1801\)](#)
- [Displaying the Saved Value of a State Retention Variable \(CPF\)](#)

verisium

The Tcl `verisium` command allows you to:

- Invoke Verisium and connect to the current simulation (`verisium` command with no options).

- Send a `verisium` command (or multiple commands) to Verisium (`verisium -submit`). If you are not currently in Verisium, this command invokes the GUI and runs the command(s).

verisium Command Syntax

```
verisium  
[-submit] verisium_command
```

verisium Command Options

This section describes the options that you can use with the Tcl `verisium` command.

| | |
|---|---|
| | Runs the specified Verisium command. |
| <code>[-submit] verisium_command</code> | If you are not currently in the Verisium environment, a <code>verisium -submit</code> command invokes Verisium and then runs the command. |
| or | |
| <code>[-submit] {verisium_command1;verisium_command 2}</code> | You can specify multiple commands inside the curly brackets and separate the commands using ";" . The <code>-submit</code> option is optional. |

verisium Command Examples

The following command invokes Verisium and connects to the current simulation:

```
xcelium> verisium
```

The following command runs a Verisium command to create a new waveform window:

```
xcelium> verisium [-submit] waveform new
```

The following command runs a Verisium command to create multiple waveform windows:

```
xcelium> verisium [-submit] {waveform new;waveform new}
```

version

The Tcl `version` command displays the version number of *xmsim*.

You can also print the version number by invoking the simulator with the `-version` option.

```
% xmsim -version
```

version Command Syntax

```
version
```

version Command Options

None.

version Command Examples

```
xcelium> version  
TOOL:    xmsim    15.20-p001
```

warn

The Tcl `warn` command allows you to enable or disable warning messages for invalid index access during read and write operations during simulation.

warn Command Syntax

```
warn [-on|-off] <warning_mnemonic> -scope <scope_name> -depth <levels|all>
```

warn Command Options

This section describes the options that you can use with the Tcl `warn` command.

`[-on|-off]`
`<warning_mnemonic>`

Specifies whether to enable or disable warning message for a mnemonic. The `warn` command has the following options:

- `[-on] <warning_mnemonic>` Enables the warning messages for the specified warning mnemonic.
- `[-off] <warning_mnemonic>` Disables the warning message for the specified warning mnemonic.

The `warning_mnemonic` can be `IIWAOB`, `IIRAOb`, `IIWXZI`, or `IIWOOb`. Specifying any other warning mnemonic results in an error.

`-scope <scope_name>` To enable or disable a warning under the specified scope hierarchy. This is an optional switch. Using the `-warn` command without the `-scope` option enables or disables a warning throughout the design hierarchy.

scope_name is a string referring to a scope in the design. It can be full path from top of design or relative path from current TCL scope. Currently, you can set the warning status for the following scopes:

- Top module
- Instance of module
- Interface or program
- Generate block
- Package

`-depth <levels|all>` To enable or disable a warning for sub-scopes of the specified scope name.

You can specify a positive integer to set the level of depth in the design hierarchy. For example:

- `-depth 1` means that the warning is set for the specified scope
- `-depth 2` means that the warning is set for the current scope and its immediate sub-scope
- `-depth 3` means that the warning is set for the current scope, and the following 2 sub-scopes.

If `-depth all` is specified, then the warning is set for the specified scope hierarchy. By default, the `-depth` is set as 1.

Using the `-depth` option without specifying the `-scope` option results in a command error.

warn Command Examples

The examples in this section use the following HDL code:

```
module test1();
  int a, b;
  initial #1 a[100] = 1;
  initial #1 tb.t2.b[100] = 1;
endmodule

module test2();
  int a, b;
  initial #2 a[100] = 1;
  initial #2 tb.t1.a[100] = 1;
endmodule

module tb;
  int a;
  test1 t1();
  test2 t2();
endmodule
```

The following command disables the IIWAOB warning for `tb`, `tb.t1`, `tb.t2`:

```
xcelium> warn -off IIWAOB -scope tb -depth all
```

The following command enables the IIWAOB warning for `tb.t1`:

```
xcelium> warn -on IIWAOB -scope tb.t1
```

where

The Tcl `where` command displays the current location of the simulation.

where Command Syntax

```
where
```

where Command Options

None.

where Command Examples

The output of the `where` command depends on the phase of a delta cycle in which the simulator is stopped. The two phases are called Behavior Execution and Wire Resolution (for Verilog) and Process Execution and Signal Resolution (for VHDL). You can use the Cycle View window in the SimVision analysis environment to see which phase the simulator is currently in.

For example, suppose that line 13 of scope `:dff_inst` in your VHDL model is as follows:

```
q <= d;
```

- If you set a line breakpoint on line 13 and then simulate, the simulator stops when it reaches line 13. The simulator is in the Process Execution phase, and this time is different from the time at which the object `q` on line 13 actually gets its value. A `where` command executed at this point shows the line number, file, and the scope for the line.
- If you also set an object breakpoint on `q`, and advance the simulation, the simulator stops when `q` changes value. The simulator is now in the Signal Resolution phase, where `q` actually gets its value. In this case, a `where` command displays the current simulation time.

```
xcelium> stop -create -line 13 :dff_inst      ;# Set line breakpoint
Created stop 1
xcelium> stop -object :dff_inst:q           ;# Set object breakpoint
Created stop 2
xcelium> run
0 FS + 0 (stop 1: ./test.vhd:13)
xcelium> where                          ;# Stopped at line 13 in Process Execution phase
Line 13, file "./test.vhd", scope (:dff_inst.dff)
Scope is (:dff_inst.dff)
xcelium> run
0 FS + 1 (stop 1: ./test.vhd:13)
xcelium> where                          ;# Stopped when q gets its value in Signal Resolution phase
Line 13, file "./test.vhd", scope (:dff_inst.dff)
Scope is (:dff_inst.dff)
xcelium> run
0 FS + 2 (stop 2: :dff_inst:q = '0')
xcelium> where                          ;# Stopped when q gets its value in Signal Resolution phase
TIME: 0 FS + 2
Scope is (:dff_inst.dff)
```

xprof

The Tcl `xprof` command is an alias for the `profile` command, which lets you invoke the profiler and control its behavior. Use the `xprof` command with the Advanced Profiler.

Related Topics

- [profile](#)
- [xProf Command-Line Interface](#)

xprop

The Tcl `xprop` command dynamically controls X-Propagation at simulation time. You can also use this command during simulation to display the status of X-Propagation, and enable or disable verbose X-Propagation messages.

xprop Command Syntax

```
xprop
    [-off]
    [-on]
    [-status]
    [-verbose] <on | off>
```

xprop Command Options

This section describes the options that you can use with the Tcl `xprop` command.

| | |
|--|--|
| <code>-off</code> | Disables X-Propagation for the current simulation. |
| <code>-on</code> | Enables X-Propagation for the current simulation. |
| <code>-status</code> | Displays the status of X-Propagation for the current simulation. |
| <code>-verbose <on off></code> | Modifies the output of verbose X-Propagation messages. When using this option, you specify one of the two following arguments: <ul style="list-style-type: none">• on: This switches on verbose reporting.• off: This switches off verbose reporting. |

xprop Command Examples

This section demonstrates how to use the Tcl `xprop` command.

Example: Using xprop to Enable and Disable Verbose Reports at Simulation Time

If the `-xverbose` command-line option is specified at elaboration time, there may be a number of X-Propagation messages output during the reset phase. In order to record only useful messages, you can use the Tcl `xprop` command to temporarily disable these messages and re-enable verbosity after the reset

has been de-asserted.

```
Loading snapshot worklib.top:tb ..... Done
This snapshot is X-propagation enabled.
xcelium> xprop -verbose off
X-propagation verbosity is disabled
X-propagation is enabled
xcelium> xprop -status
X-propagation verbosity is disabled
X-propagation is enabled
xcelium> # Run till reset
xcelium> run 1 ns
Ran until 1 NS + 0
xcelium> xprop -verbose on
X-propagation verbosity is enabled
X-propagation is enabled
xcelium> xprop -status
X-propagation verbosity is enabled
X-propagation is enabled
xcelium> run 50 ns
XPROP: 'X' detected in if block
FILE: /temp/top.vhd
LINE: 39
SCOPE: :top:
TIME: 19 NS + 1
...
```

Example: Using xprop to Enable and Disable X-Propagation at Simulation Time

The following example shows how to use the Tcl xprop command to dynamically control X-Propagation during a simulation run:

```
xcelium> xprop -status
X-propagation verbosity is enabled
X-propagation is enabled
xcelium> run 20 ns
XPROP: 'X' detected in if block
FILE: /temp/top.vhd
LINE: 39
SCOPE: :top:
TIME: 19 NS + 1
Ran until 20 NS + 0
xcelium> xprop -off
X-propagation is disabled
xcelium> run 20 ns
Ran until 40 NS + 0
xcelium> xprop -on
```

```
X-propagation is enabled
xcelium> run 10 ns
XPROP: 'X' detected in if block
FILE: /temp/top.vhd
LINE: 39
SCOPE: :top:
TIME: 41 NS + 1
...
```

Related Topic

- [Introduction to X-Propagation](#)

ida_database

The Tcl `ida_database` command opens or closes a debug database. During the initial simulation run, this command can be used to create the database that contains the debugging information. Later, while debugging the test environment, this command can be used to reopen the database with read access.

ida_database Command Syntax:

```
ida_database [-h[elp]]
[-open]
[-close]
[-name]
[-save_files]
[-wave_glitch_recording]
[-wave_db_args]
```

ida_database Switches

| | |
|---------------------------|---|
| -h[elp] | Prints this help description. |
| -open | Opens a trace database. |
| -close | Closes a trace database. |
| -name= <db_name> | IDA database name. |
| -save_files <= directory> | Copy all files that participated in the simulation to a snapshot area (<i>directory</i> location). |
| -wave_glitch_recording | <p>The default behavior is not to dump glitches. You can open SST2 DB with <code>-event</code> option. Dumps wave information to the database using the default configuration level of <i>xmsim</i>. This option includes delta cycle information in waveforms. For a lower level of debug, this switch adds the delta cycle information and consequently increases the accuracy of HDL cause analysis operations. This option may affect waveform recording performance.</p> <div style="border: 1px solid #f0e68c; padding: 10px; margin-top: 10px;"> <p>This option is valid only when first opening the wave DB (using the first <code>ida_database</code> command).</p> </div> |
| -wave_db_args | Lists arguments for waveform database. See the table below. |

-wave_db_args Switches (Waveform Database Args)


| | |
|--------------------------|--|
| -compress/-gzip | Add a compression step when dumping an SHM to reduce its size. |
| -maxsize <byte-size> | <p>Maximum size in bytes for the database - the oldest data in the database is discarded to keep the size near this limit (minimum size is between 2Mb and 4Mb).</p> <p>The size can also be specified in KiloBytes (K), MegaBytes (M) and GigaBytes(G).</p> |
| -incfiles <file-count> | <p>Specifies the maximum number of incremental files that will be kept for the SHM database. When more than this number of complete incremental files have been written, the oldest files are automatically deleted.</p> <p>The default value of <file-count>=0, meaning all incremental files are kept.</p> |
| -incsize <Megabyte-size> | <p>Specifies the incremental file size in Megabytes for the SHM database. When the current database file size reaches approximately the size specified, a new incremental file is started automatically. The size can be specified in MegaBytes (M) and GigaBytes(G).</p> |
| -inctime <time_stamp> | <p>Specifies the incremental time-stamp for the SHM database.</p> <p>When the current simulation time reaches the time specified, a new incremental file is started automatically.</p> |

This example opens a database named `my_waves.db` with a maximum size of 2Gb and files split into chunks of 100 μ s of simulation time and with glitch recording:

```
ida_database -open -name my_waves.db -wave_db_args "-maxsize 2G -inctime 100us"  
            -wave_glitch_recording
```

Managing the SHM Database

You can open a database, probe signals, and save the results in the database by entering Tcl commands at the prompt or by using the graphical user interface.

 The objects you want to probe to an SHM or VCD database must have read access. By default, objects in the design are not given read access. Use the `-access +r` option or the `-afile access_file` option when you elaborate the design to provide read access.

For Verilog, you also can use a set of system tasks to open an SHM database, probe signals, and save the results in the waveform database. When working with waveforms, there are two basic use models:

- Invoke the simulator with the SimVision analysis environment so that you can simulate and view the waveforms as the simulation progresses. You can also invoke SimVision separately and then connect to a running simulation either on the same system or over a network.
- Simulate and generate a database. Then invoke the SimVision analysis environment in post-processing mode (PPE).

You must enter the system tasks into the Verilog description before simulation. The system tasks are:

- `$shm_open()`: Opens a simulation database.
- `$shm_probe()`: Specifies the signals whose simulation value changes are entered into the simulation database.
- `$shm_close`: Closes a simulation database.

Consider the following example.

```
initial
begin
    $shm_open("waves.shm");
    $shm_probe();
    #1 $stop; // stop simulation at time 1
end
```

- For backward compatibility with Verilog code that contains calls to the Signalscan tasks for recording data during a Verilog simulation (`$recordvars`, `$recordfile`, `$recordsetup`, and so on), Cadence has implemented these tasks as system tasks native to the simulator. You can keep these calls in your Verilog code, and they will function as intended.
- In addition to the implementation in which the RecordVars system tasks are built into the Xcelium Simulator, Cadence also provides two other implementations of RecordVars system tasks, each with support for different simulators, platforms, and some differences in behavior. See the *RecordVars User Guide* for details on these implementations.

Related Topics

- [Opening a Database](#)
- [Setting and Deleting Probes](#)
- [Using the Waveform Viewer](#)
- [Opening a Database with `\$shm_open`](#)
- [Probing Signals with `\$shm_probe`](#)
- [Using `\$recordvars` and Related Tasks](#)

Opening a Database with `$shm_open`

Use the `$shm_open` system task to open an SHM database.

Syntax

```
$shm_open ("db_name", is_sequence_time, db_size, is_compression, incsize, incfiles);
```

Arguments

The following table describes the `$shm_open` task arguments. All arguments are optional.

| | |
|------------------|--|
| db_name | <p>Specifies the filename of the simulation database. If you do not specify the database name, a database called <code>waves.shm</code> is opened in the current directory.</p> |
| is_sequence_time | <p>Dumps all value changes to the database.</p> <p>By default, when probing to an SHM database, the simulator discards multiple value changes for an object during one simulation time and dumps only the final value at the end of that simulation time. Specify <code>1</code> for the <code>is_sequence_time</code> argument if you want to dump all value changes to the SHM database. You can then use the SimVision Waveform Viewer to expand a single moment of simulation time to show the sequence of value changes that occurred at that time. For example:</p> <pre>\$shm_open("mywaves.shm", 1, ,);</pre> |
| db_size | <p>Specifies the maximum size (in bytes) of the transition file (<code>.trn</code> file).</p> <p>The simulator maintains the size limit of the transition file by discarding the earliest recorded values as new values are dumped, such that the database always contains the most recent values for each probed object.</p> <p>When the size limit is exceeded, the waveform window displays an "unknown" value for each object from the beginning of the simulation to the time of the first non-discarded value.</p> <p>The SHM database uses approximately 2.5 MB of disk space, even if you specify a lower limit. However, the database size will not exceed the limit if the limit is greater than 2.5 MB. For example:</p> <pre>\$shm_open("mywaves.shm", 1, 250000,);</pre> |

`is_compression`

Compresses the SHM database to reduce its size. The default setting is 0. Specify 1 to compress the database file. For example:

```
$shm_open("mywaves.shm", 1, , 1);
```

If you open an SHM database with the `$shm_open` system task in your Verilog code, the name of the database that is created is preceded by an underscore character. For example, the following system task opens a database called `_waves.shm`.

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

`incsize`

Specifies the incremental file size for the SHM database.

By default, there is no limit on the size of an SHM database. Because a database for a large simulation can be very big, you might want to break up the signal transition information (the `.trn` file) into multiple files. These files are called *incremental files*, and each file corresponds to a range of simulation time.

Breaking up a large database file into incremental files can make the simulation results more manageable. You can open just one incremental file, or any subset of the files, in SimVision so that you can view the waveforms for the time range(s) corresponding to that file or set of files. This can improve viewer performance and memory usage. Incremental files can also be used to ensure that database files are kept under some specified size (for example, 2 GB), and files corresponding to uninteresting time ranges can be deleted to save disk space.

Use the `incsize` argument to specify a size limit for the SHM database file. The `incsize` argument is an integer that is interpreted as megabytes. The default is 0, which means unlimited size. When the current SHM database file size reaches approximately the size specified with `incsize`, a new incremental file is started automatically. The incremental files are numbered (for example, `waves-1.trn`, `waves-2.trn`, and so on).

Use the `incfiles` argument to set a limit on the number of incremental files that can be stored on disk.

See the description of the `incfiles` argument below for examples of using the `incsize` and `incfiles` arguments.

incfiles

Sets a limit on the number of incremental files that can be stored on disk.

This argument is an integer. The default is 0, which means that the simulator can create and store as many incremental files as needed.

Examples:

The following `$shm_open` task opens a database called `waves.shm`. Default values are used for the `is_sequence_time`, `db_size`, and `is_compress` arguments. The incremental file size is set at 1 MB. No value is specified for the *incfiles* argument, which means that the simulator will create as many incremental files as necessary. Each incremental file will contain approximately 1 MB of data.

```
$shm_open("waves.shm", , , , 1, );
```

In the following `$shm_open` task, the incremental file size is set at 2 MB. The *incfiles* argument specifies that four incremental files can be stored on disk. Each incremental file will contain approximately 2 MB of data. If the simulation generates more than 8 MB of data, the first incremental file (`waves.trn`) is deleted, and a `waves-4.trn` is created.

```
$shm_open("waves.shm", , , , 2, 4);
```

In the following `$shm_open` task, the incremental file size is set at 2 GB. The *incfiles* argument specifies that three incremental files can be stored on disk.

```
$shm_open("waves.shm", , , , 2048, 3);
```

Related Topics

- [Managing the SHM Database](#)
- [Probing Signals with \\$shm_probe](#)

Probing Signals with \$shm_probe

The `$shm_probe` system task lets you specify the signals whose value changes you want to record in your SHM database, and lets you specify the nodes at which value changes are recorded.

Syntax

```
$shm_probe[ ( scope1, "node_specifier1", scope2, "node_specifier2", ... ) ];
```

Arguments

You can specify the following arguments with this task. The arguments to `$shm_probe` are optional. If you do not specify any arguments, `$shm_probe` writes the value changes that occur at all inputs, outputs, and inouts in the current scope to your SHM database.

- *scope1*, *scope2*, ...
Specifies the scope or scopes whose signals you want to probe. If you do not specify a scope, `$shm_probe` records the signal changes that occur in the current scope.

- "*node_specifier1*", "*node_specifier2*", ...
Specifies a code, called a node specifier, to indicate the nodes at which you want to record value changes for the specified signals.

Node specifiers apply to the specified scopes in order of appearance. That is, *node_specifier1* applies to *scope1*, *node_specifier2* applies to *scope2*, and so on. If a node specifier does not have a corresponding scope, it applies to the current scope. If a scope does not have a corresponding node specifier, `$shm_probe` records value changes at all inputs, outputs, and inouts in that scope.

The node specifiers are shown in the following table. The specifiers are shown in uppercase, but they are case-insensitive. The characters in a node specifier can be entered in any order.⁴

| Node Specifier Character | Signals That Enter the Database |
|--------------------------|---------------------------------|
| | |

| | |
|-----|--|
| "A" | All nodes (including inputs, outputs, and inouts) in the specified scope, excluding memories. |
| "S" | Inputs, outputs, and inouts in the specified scope, and in all instantiations below the specified scope, except nodes inside library cells. |
| "C" | Inputs, outputs, and inouts in the specified scope, and in all instantiations below the specified scope, including nodes inside library cells. |
| "M" | Inputs, outputs, inouts, and memories in the specified scope. |
| "T" | Objects declared in task scopes. By default, task scopes are not included unless the specified scope is a task scope. |
| "F" | Objects declared in function scopes. By default, function scopes are not included unless the specified scope is a function scope. |

Examples

The following examples show you how to use `$shm_probe` to choose the signals and nodes whose value changes you want to record in your SHM database.

- `$shm_probe () ;`

Record value changes at all inputs, outputs, and inouts in the current scope.

- `$shm_probe (alu) ;`

Record value changes at all inputs, outputs, and inouts in the scope `alu`.

- `$shm_probe (top.dut1, top.dut2) ;`

Record value changes at all inputs, outputs, and inouts in the scopes `top.dut1` and `top.dut2`.

- `$shm_probe ("A") ;`

Record value changes at all nodes in the current scope, excluding memories.

- `$shm_probe ("AM") ;`

Record value changes at all nodes in the current scope, including memories.

- `$shm_probe ("AMC") ;`

Record value changes at all nodes (including inputs, outputs, inouts, and memories) in the current scope, and in all instantiations below it, including nodes inside library cells.

- `$shm_probe("ACMTF");`

Record value changes at all nodes (including inputs, outputs, inouts, memories, and objects declared in task and function scopes) in the current scope, and in all instantiations below it, including nodes inside library cells.

- `$shm_probe(top.dut1, "S", top.dut2, "AC");`

Record value changes at:

- All inputs, outputs, and inouts in the scope `top.dut1` and in all scopes below `top.dut1`, excluding nodes in library cells.
- All nodes in the scope `top.dut2` and in all scopes below `top.dut2`, including nodes in library cells.

- `$shm_probe("S", top.dut1, "ACMT");`

Record value changes at:

- All inputs, outputs, and inouts in the current scope and below, excluding nodes in library cells.
- All nodes in the scope `top.dut1` and in all scopes below `top.dut1`, including nodes in library cells, and including memories and objects declared in task scopes.

Using \$recordvars and Related Tasks

If you have Verilog code that contains calls to the Signalscan, `$recordvars` and related tasks (`$recordfile`, `$recordsetup`, and so on), you can use these calls to record data in an SHM (SST2) database. The PLI tasks that were used for recording data during a Verilog simulation have been implemented as system tasks native to the simulator. That is, it is not necessary to write PLI code and to link this into the simulator.

The following Signalscan PLI tasks have been implemented as system tasks native to the simulator:

- [`\$recordvars`](#)
- [`\$recordfile`](#)
- [`\$recordsetup`](#)
- [`\$recordon/\$recordoff`](#)

- [\\$recordclose](#)
- [\\$recordabort](#)
- [\\$signalscan](#)

Only the tasks that are used for recording data have been implemented as system tasks. The following Signalscan tasks for interfacing with Signalscan from your Verilog code are not supported, and using them will generate warning messages: `$signalscanconnect`, `$signalscancommand`, `$signalscankill`, `$signalscanabort`.

In addition, some options to the `$record*` tasks have not been implemented. These are the options that have to do with writing incremental files (`incsize`, `inctime`, `inccpu`, `incfiles`, `summary`, and `nosummary`). Using these options will generate warning messages.

If you must use the `$signalscan*` tasks listed above or the incremental file options, you can revert to using the PLI interface support. There are two easy ways to do this:

- Add the following path to your definition of the `LD_LIBRARY_PATH` (Solaris) or `SHLIB_PATH` (HP) variable:

```
your_install_dir /tools/simvisdai/lib
```

- Create a link to the PLI interface library with the following command:

```
ln -s your_install_dir /tools/simvisdai/lib/record-scb.so  
your_install_dir /tools/lib/.
```

There are a few differences between the database that is dumped using the new built-in system tasks and the database that is dumped using the PLI interface support. These differences include the following:

- The database dumped using the PLI interface support includes the highconns for ports. These are not always included in the database that is dumped using the new built-in system tasks.
- The database dumped using the PLI interface support includes continuous assignments that are not always included in the database that is dumped using the new built-in system tasks.
- For primitive terminals, the database that is dumped using the new built-in system tasks always dumps the signal to which the terminal is connected.

\$recordvars

The only system task required to record data to an SHM (SST2) database is `$recordvars`.

Syntax


```
$recordvars[("options")];
```

If you do not specify any options, value changes on all signals in the design hierarchy (with no driver or primitive information) are recorded.

Only one database may be written at a time, but you can add variables to be recorded to the database at any time by using another `$recordvars` task.

The following table lists the options that you can use with `$recordvars`. Options apply to all following variables and scopes in the call, or to the default scopes if none are specified.

| Option | Effect | Default |
|--------------|---|----------------|
| "depth=n" | Limit the depth if a scope is specified. If "depth=1", no child scopes are included. | "depth=0" |
| "drivers" | Record drivers (an output terminal of a primitive). Drivers are recorded for all recorded variables that have more than one driver. | "nodrivers" |
| "primitives" | Record primitives. For all scopes that are recorded, record their primitives in addition to their variables. | "noprimitives" |
| "nocells" | Do not record variables within a cell, or within any scopes below the cell. By default, modules defined in a library are cells and other modules are not cells. A non-library module can be defined as a cell using the Verilog <code>`celldefine</code> directive. | "cells" |
| "noports" | Do not record port connectivity information. This option is used primarily to work around a simulator defect that affects some designs. If this option is used, SimVision waveform viewer does not display ports in different colors, the Schematic Tracer does not display port connectivity, and the Add Trace and Add Module Inputs commands do not display ports. | "ports" |

| | | |
|--------------|--|--|
| "trace" | <p>Record statement trace information.</p> <p>If you use this option, you must also use the "sequence" option on either the \$recordfile or \$recordsetup task.</p> <p>Recording statement trace information is independent of what variables you are recording. You must record variables in separate \$recordvars task statements. Do not specify other options in the same \$recordvars statement where you specify the "trace" option.</p> | |
| Any variable | <p>Record a variable. A variable can be a net, register, integer, time, real, or named event. For example,</p> <pre>top.u1.u32.a.</pre> | |
| Any scope | <p>Record a scope. By default, all variables within the scope and all variables in all child scopes are recorded in the database. Use "depth=n" to limit the depth. A scope can be a module, task, function, or named block. For example,</p> <pre>top.control.</pre> | All top-level modules and all subscopes. |

If you open an SHM database with the \$record* system task in your Verilog code, the name of the database that is created is preceded by an underscore character. For example, assuming that the top-level module is called `top`, the following system task opens a database called `_top`.

```
$recordvars;
```

The following system tasks open a database called `_results`.

```
$recordfile("results");
$recordvars;
```

This lets you interact with databases opened with \$record* in the same way that you interact with databases that you open with the `database` command or with `$shm_open`. That is, you can use the `database` command to disable, enable, or display information about the databases.

The \$recordvars task generates two output files:

- A design file (`.dsn`), which contains information about the design.

By default, the name of this file is `design_name-version_name.dsn`. For example, `top-1.dsn`.

- A transition file (`.trn`), which contains the transition information.

By default, the name of this file is `design_name-version_name-run_name.trn`. For example, `top-1-1.trn`.

Use the `$recordsetup` task to override the default `design_name`, `version_name`, and `run_name`.

If you revert to using the PLI interface support, the file naming convention is the same as that described above if you do not include a `$recordfile` task to specify the name of the database. If you use `$recordfile` to specify a database name, the files are called `database_name.dsn` and `database_name.trn`. These files are overwritten every time the simulator is run.

For example the following code generates `results.dsn` and `results.trn`:

```
$recordfile("results");  
$recordvars;
```

Example 1

In the following example, no options, variables, or scopes are specified in the `$recordvars` call. All top-level modules are used by default, and all variables in the design are recorded.

```
module record;  
    initial $recordvars;  
endmodule
```

Example 2

In the following example:

- The first `$recordvars` records all variables within scope `top.mod1` and all of its descendants, but records only the variables three levels deep for scope `top.mod2`.
- The second `$recordvars` illustrates how options apply to all variables specified later in that `$recordvars` task unless overridden. This task records driver information for variables in `mod1` and driver and primitive information for variables in `mod2`.
- The third `$recordvars` records two explicitly named variables.

```
module record;  
    initial  
        begin  
            $recordvars(top.mod1, "depth = 3", top.mod2);  
            $recordvars("drivers", mod1, "primitives", mod2);  
            $recordvars(top.io.mux1.q0, top.io.mux2.q0);  
        end  
endmodule
```

Example 3

In the following example:

- The first `$recordvars` records three levels of variables within scope `top.mod1`. Drivers are also recorded if the variables have more than one driver. Scope `top.mod2` is not depth restricted and no driver information is recorded for variables in this scope.
- The second `$recordvars` records driver information for variables in `mod1` and driver and primitive information for variables in `mod2`.
- The third `$recordvars` records `top.middle.clock` and all variables in `module2` and its subscopes.
- The fourth `$recordvars` records statement trace information. Recording statement trace information is independent of what variables you are recording. You must record variables in `$recordvars` task statements, and specify the trace option in a separate `$recordvars` statement.
- The `$recordsetup` task in this example specifies the recording of sequence information. Sequence information is needed to correlate statements and transitions. If you collect trace information but do not collect sequence information, you will receive a warning message during simulation.

The recording of statement trace information is either on or off for the entire simulation. The `$recordon` and `$recordoff` statements have no effect on recording statement trace information. Other `$recordvars` options, such as specifying depth or scopes, have no effect on how much statement trace information is recorded.

```
module record;
  initial
    begin
      $recordsetup("design = mydesign", "sequence");
      $recordvars("depth = 3", "drivers", top.mod1,
                 "depth = 0", "nodrivers", top.mod2);
      $recordvars("drivers", mod1, "primitives", mod2);
      $recordvars(top.middle.clock, module2);
      $recordvars("trace");    // Must be alone
    end
endmodule
```

\$recordfile

The `$recordfile` task records basic design information and sets up the recording options for variables recorded with `$recordvars`. This task is optional. If you use it, the task should be placed before the first `$recordvars` task.

Syntax

```
$recordfile( filename [,"options"] );
```

where, *filename* is the name of the database. This can be a string enclosed in double quotes, or the name of a variable that contains the file name. Although not required, the extension `.trn` is recommended to identify the transition database. A `.dsn` file is also created with the same base name as the `.trn` file.

The following `$recordfile` options, all of which have to do with writing incremental files, are not supported. Using them will generate a warning message:

- `"incsize = size"`
- `"inctime = simtime"`
- `"inccpu = cputime"`
- `"incfiles = count"`
- `"summary[=file]"` and `"nosummary"`

The following table lists options that you can use with the `$recordfile` task.

| Option | Effect | Default |
|------------------------------|---|---------|
| <code>"wrapsize=size"</code> | <p>Limit the size of the <code>.trn</code> file before data is wrapped into another file.</p> <p>The size argument is a number followed by <code>B</code> (bytes), <code>K</code> (kilobytes), <code>M</code> (megabytes), or <code>G</code> (gigabytes). The default is <code>M</code>.</p> <p>When the transition data exceeds the specified size, the oldest transitions are overwritten by newer transitions. However, transitions are written to the file, and discarded from the file, in blocks of about 4-5 Mb. This means that the actual size of the database can be considerably larger than, or smaller than, the specified size.</p> <p>It is recommended that the maximum size be at least 10 Mb, if specified.</p> | |

| | | |
|------------|---|--------------|
| "sequence" | Save sequence information (the sequence in which events occurred). This is necessary for tracing. | "nosequence" |
| "compress" | Compress the database. Sequence information is not compressed. | "nocompress" |

Example

In the following example, a design file named `adder-1.dsn` and a transition file named `adder-1-1.trn` is created. The transition file is compressed. Sequence time is recorded in the database. You can record sequence information with either the `$recordfile` or the `$recordsetup` task.

You can use the "compress" and "sequence" options together, but only transition information is compressed; sequence information is not compressed.

```
$recordfile("adder", "compress", "sequence");  
$recordvars;
```

\$recordsetup

The `$recordsetup` task records basic design and hierarchy information and sets up the recording options for variables recorded with `$recordvars`. This task is optional. If you use it, the task should be placed before the first `$recordvars` task.

When `$recordsetup` is called, the scope hierarchy is recorded in the design file immediately. However, primitives and variables are not recorded until `$recordvars` is called.

Syntax

```
$recordsetup( ["options"] );
```

The following \$recordsetup options, all of which have to do with writing incremental files, are not supported. Using them will generate a warning message:

- "incsize = *size*"
- "inctime = *simtime*"
- "inccpu = *cputime*"
- "incfiles = *count*"
- "summary[=*file*]" and "nosummary"

The following table lists the options that you can use with the \$recordsetup task.

| Option | Effect | Default |
|---------------------------|--|---|
| "design= <i>name</i> " | Create a name for the design. | Name of the first top scope found. |
| "version= <i>name</i> " | Name this version of the design. | Next number (based on the files in the current directory or the directory specified with the "directory" option). |
| "run= <i>name</i> " | Name this particular simulation run. | Next number (based on the files in the current directory or the directory specified with the "directory" option). |
| "directory= <i>path</i> " | Specify the directory where the files will be saved. If the specified directory does not exist, it is created for you. | Current working directory. |

| | | |
|-----------------|--|--------------|
| "wrapsize=size" | <p>Limit the size of the .trn file before data is wrapped into another file.</p> <p>The size argument is a number followed by B (bytes), K (kilobytes), M (megabytes), or G (gigabytes). The default is M.</p> <p>When the transition data exceeds the specified size, the oldest transitions are overwritten by newer transitions. However, transitions are written to the file, and discarded from the file, in blocks of about 4-5 Mb. This means that the actual size of the database can be considerably larger than, or smaller than, the specified size.</p> <p>It is recommended that the maximum size be at least 10 Mb, if specified.</p> | |
| "sequence" | Save sequence information (the sequence in which events occurred). This is necessary for tracing. | "nosequence" |
| "compress" | Compress the database. Sequence information is not compressed. | "nocompress" |

Example 1

In the following example, a design file named `data/adder-1.dsn` is created. If `adder-1.dsn` already exists in the `data` directory, `adder-2.dsn` is created. A transition file named `data/adder-1-1.trn` is created. If this file already exists, a file called `adder-2-1.trn` is created.

```
module record;
    initial
        begin
            $recordsetup("directory = data", "design = adder");
            $recordvars;
        end
endmodule
```

Example 2

In the following example, a design file named `data/adder-algo1.dsn` is created, or replaced if it exists. The database is compressed.


```
module record;
  initial
  begin
    $recordsetup("directory = data", "design = adder", "version = algo1",
                "compress");
    $recordvars;
  end
endmodule
```

\$recordon/\$recordoff

Use the `$recordon` and `$recordoff` tasks to turn recording on or off, respectively. Recording can be turned on or off at selected times or based on conditions in Verilog.

The `$recordoff` task does not close the database file. Variable transitions are not recorded during the period where recording is off. All recorded variables are updated to their current values when recording is turned back on.

Example

In the following example, the `$recordon` and `$recordoff` tasks are used to record variables for a portion of the total simulation time.

```
module record;
  initial
  begin
    $recordvars;
    $recordoff;
  end
endmodule

module top;
  reg clock;
  initial
  begin
    #0 clock=0;
    #100 clock=1;
    #100 clock=1;
    #100 clock=0; $recordon;
    #100 clock=1;
    #100 clock=0;
    #100 clock=1; $recordoff;
    #100 clock=0;
  end
endmodule
```

```
#100 clock=1;  
end  
endmodule
```

\$recordclose

Use the `$recordclose` task to close an open database. This task stops the recording of data, flushes buffered data to the database, and closes the database.

Example

```
$recordclose;
```

\$recordabort

Use the `$recordabort` task to abort recording to a database that is no longer wanted. Any buffered information not yet written to the database is discarded, and the database is deleted. Any current interactive connection to Signalscan is also aborted.

Example

```
$recordabort;
```

\$signalscan

You can launch Signalscan from your Verilog code with the `$signalscan` task. This task is used to interactively view simulation results in Signalscan while the simulation is running. All variables being recorded to the database are available for viewing. Conversely, a variable must be recorded in the database in order to view it with Signalscan. This means that there must be at least one call to `$recordvars` in order for `$signalscan` to be useful.

Syntax

```
$signalscan( ["path = path_to_signalscan_executable"] [, "arguments"] );
```

The path to the Signalscan executable is optional. If you do not specify a path, the `PATH` environment variable is used to find the executable, which must be named `signalscan`.

You can also pass Signalscan arguments as parameters to the `$signalscan` task. The parameter is a string enclosed in double quotes.

There can be only one interactive Signalscan connection at a time for each simulation. If you exit Signalscan or the simulator, or use the `$recordclose` or `$recordabort` tasks, the interactive connection is closed. You can then call the `$signalscan` task again to start a new interactive

connection.

Example

In the following example, the `$signalscan` task specifies an absolute path to the Signalscan executable. It also includes an argument to load a Do-File.

```
module record;
    initial
        begin
            $recordvars;
            $signalscan("path=/usr/tools/signalscan-6.2/signalscan", "-do my.do");
        end
    endmodule
```

Xcelium VWDB Probing

VWDB is Cadence’s next-generation Waveform Dump Database that offers much faster probing with a much smaller memory and size footprint than SHM. The Xcelium VWDB flow currently supports two industry-standard hardware description languages, Verilog and VHDL.

There are two ways to generate VWDB files, via TCL command and system task, as shown in the example below:

1. Generate a default Xcelium VWDB with TCL command, `call xmDumpvars`.
2. Use the System Task `$xmDumpvars()` ; in your Verilog code.

The `xmDumpvars` dumps the signals' value changes (VC) to the Xcelium VWDB database.

VWDB also supports command-line simulation options and relevant environment variables, which you can use along with the TCL commands or System Calls.

The default VWDB Database name is `ida.db`, and it is the Native Verisium Debug Database. Therefore, it is easy to migrate to VWDB dumping tasks and debug designs with Verisium Debug.

No change in the Verisium Debug setup or usage is required.

- It is not allowed to mix VWDB `xmDump*` and `ida_probe` commands. If this happens, the following warning is issued, and the probe command is not executed.
`xmsim: *W,VNDMPM: VWDB dumping task can not be executed when hybrid mode is enabled.`
- If the environment already uses VWDB with `ida_probe`, you need to remove `+vwdb+shm2vwdb=[1|2]` or/and `CDNS_ENABLE_SHM2VWDB`.

Related Topics

- [Commonly Used VWDB Probe Commands](#)
- [Commonly Used VWDB Probe Examples](#)
- [Dumping Commands for Xcelium VWDB Flow](#)
- [Priority of Specifying VWDB Probe Options](#)
- [Simulator Argument for Global Settings](#)

Commonly Used VWDB Probe Commands

This topic includes a detailed description of the VWDB Probe Commands.

Database name (default: ida.db)

| | |
|--|---|
| Tcl | <code>call xmDumpfile <filename></code> |
| System Call | <code>\$xmDumpfile ("<code><filename></code>") ;</code> |
| ida_probe equivalent | <code>ida_database -open -name <filename></code> |
| Notes: <ul style="list-style-type: none"> • The default database, <code>ida.db</code>, is automatically created, and no <code>xmDumpfile</code> command is required if you do not want to change its name. • Subsequent VWDB Probe commands are probed to the database name created by the most recent <code>xmDumpfile</code> command. | |

SmartLog Probing

| | |
|-----|-----------------------------|
| Tcl | <code>call xmDumpLog</code> |
|-----|-----------------------------|

| | |
|-----------------------------|-----------------------------|
| System Call | <code>\$xmDumpLog();</code> |
| ida_probe equivalent | <code>ida_probe -log</code> |

Probing from specified instance <top> all RTL, including memories, to all depths

| | |
|--|---|
| Tcl | <code>call xmDumpvars <top> 0 +all</code> |
| System Call | <code>\$xmDumpvars("<top>",0,"+all");</code> |
| ida_probe equivalent | <code>ida_probe -wave -wave_probe_args="<top> -all -memories -depth all"</code> |
| Notes: <ul style="list-style-type: none"> "+all": dumps all signals, including memories, packed array, structure, packed union, and power in the specified depth and given scope. "0": Equivalent to "<code>-depth all</code>." | |

Probing all <top> instances of all RTL, including memories, to all depths

| | |
|-----------------------------|---|
| Tcl | <code>call xmDumpvars 0 +all</code> |
| System Call | <code>\$xmDumpvars(0,+all);</code> |
| ida_probe equivalent | <code>ida_probe -wave -wave_probe_args="[scope -tops] -all -memories -depth all"</code> |

Probing all <top> instances of all RTL, including memories, to cells depth

| | |
|-----------------------------|--|
| Tcl | <code>call xmDumpvars +all +cell_instance=0</code> |
| System Call | <code>\$xmDumpvars("+cell_instance=0","+all");</code> |
| ida_probe equivalent | <code>ida_probe -wave -wave_probe_args="[scope -tops] -all -memories -depth to_cells"</code> |

Probing all <top> instances of all RTL, including Low Power including memories, to all depths

| | |
|--|--|
| Tcl | <code>call xmDumpvars 0 +all +power</code> |
| System Call | <code>\$xmDumpvars(0,+all, +power);</code> |
| ida_probe equivalent | <code>ida_probe -wave -wave_probe_args="[scope -tops] -all -memories -depth all -power"</code> |
| Note: <ul style="list-style-type: none"> +power is included in +all and does not have to be provided explicitly. | |

Probing all <top> instances of all RTL, including Low Power Shadow Registers including memories, to all depths

| | |
|-----------------------------|---|
| Tcl | <code>call xmDumpvars 0 +all +power +power_sr_save</code> |
| System Call | <code>\$xmDumpvars(0,"+all", "+power", "+power_sr_save");</code> |
| ida_probe equivalent | <code>ida_probe -wave -wave_probe_args="[scope -tops] -all -memories -depth all -power -sr_save_all"</code> |

Probing Signal Strength for all <top> instances of all RTL, including memories, to all depths

| | |
|---|---|
| Tcl | <pre>call xmDumpvars 0 +all</pre> <p>In addition, you need to provide a simulation command line option:</p> <pre>%> xrun +vwdb+strength</pre> |
| System Call | <pre>\$xmDumpvars(0,"+all");</pre> <p>In addition, you need to provide a simulation command line option:</p> <pre>%> xrun +vwdb+strength</pre> |
| ida_probe equivalent | <pre>ida_probe -wave -wave_probe_args="[scope -tops] -all -memories -depth all"</pre> |
| Note: <ul style="list-style-type: none"> • <code>ida_probe</code> by default, probes the signal's strength. | |

Probing Glitches for all <top> instances of all RTL, including memories, to all depths

| | |
|-----------------------------|---|
| Tcl | <pre>call xmDumpvars 0 +all</pre> <p>In addition, you need to provide a simulation command line option:</p> <pre>xrun +vwdb+seq_mode=vc</pre> |
| System Call | <pre>\$xmDumpvars(0,"+all");</pre> <p>In addition, you need to provide a simulation command line option:</p> <pre>xrun +vwdb+seq_mode=vc</pre> |
| ida_probe equivalent | <pre>ida_database -open -name ida.db -wave_glitch_recording</pre> <pre>ida_probe -wave -wave_probe_args="[scope -tops] -all -memories -depth all"</pre> |

Probing Basic SVA assertions for all <top> instances of all RTL, including memories, to all depths

| | |
|--|---|
| Tcl | <pre>call xmDumpSVA</pre> |
| System Call | <pre>\$xmDumpSVA();</pre> |
| ida_probe equivalent | <pre>ida_probe -wave -wave_probe_args="[scope -tops] -all -depth all"</pre> |
| Note: <ul style="list-style-type: none"> • <code>ida_probe -all</code> includes basic SVA assertion probing. | |

Probing Xcelium SVA assertions and keeping debug information of transactions for all instances to all depths

| | |
|-----------------------------|---|
| Tcl | <pre>call xmDumpSVA +assertdebug</pre> |
| System Call | <pre>\$xmDumpSVA("+assertdebug");</pre> |
| ida_probe equivalent | <pre>ida_probe -wave -wave_probe_args="[scope -tops] -assertion -assertdebug"</pre> |

Probing Dynamic Types and all <top> instances of all RTL, including memories, to all depths

| | |
|-----------------------------|--|
| Tcl | <pre>call xmDumpvars <top> 0 +all +dynamic</pre> |
| System Call | <pre>\$xmDumpvars(0,"+all", "+dynamic");</pre> |
| ida_probe equivalent | <pre>ida_probe -wave -wave_probe_args="<top> -all -memories -depth all -dynamic"</pre> |

Probing Dynamic Types and UVM Testbench and <top> instances of all RTL, including memories, to all depths

| | |
|-----------------------------|---|
| Tcl | <code>call xmDumpvars <top> 0 +all +dynamic \$uvm:{uvm_test_top}</code> |
| System Call | <code>\$xmDumpvars(0, <top>, "+all", "+dynamic", "\$uvm:{uvm_test_top}");</code> |
| ida_probe equivalent | <code>ida_probe -wave -wave_probe_args="<top> \$uvm:{uvm_test_top} -all -memories -depth all -dynamic"</code> |

Commonly Used VWDB Probe Examples

The following sections include a detailed description of the VWDB Probe Examples.

Probe Full RTL Designs - All depths with SmartLog

| | |
|---|--|
| Tcl | <code>call xmDumpLog</code> <code>call xmDumpvars 0 +all</code> <code>call xmDumpSVA</code> |
| System Call | <code>\$xmDumpLog();</code> <code>\$xmDumpvars(0, "+all");</code> <code>\$xmDumpSVA();</code> |
| ida_probe equivalent | <code>ida_probe -log</code> <code>ida_probe -wave -wave_probe_args="[scope -tops] -all -memories -depth all"</code> |
| Note: <ul style="list-style-type: none"> For System Call, it will probe from the current scope and below, and not all the scopes. | |

Probe Full RTL Design - To Cells Depth with SmartLog

| | |
|-----------------------------|---|
| Tcl | <code>call xmDumpLog</code> <code>call xmDumpvars +all +cell_instance=0</code> <code>call xmDumpSVA</code> |
| System Call | <code>\$xmDumpLog();</code> <code>\$xmDumpvars("+cell_instance=0", "+all");</code> <code>\$xmDumpSVA();</code> |
| ida_probe equivalent | <code>ida_probe -log</code> <code>ida_probe -wave -wave_probe_args="[scope -tops] -all -memories -depth to_cells"</code> |

Probe Full RTL Design with Low Power – Depth to_cells with SmartLog

| | |
|------------|--|
| Tcl | <code>call xmDumpLog</code> <code>call xmDumpvars +all +cell_instance=0</code> or <code>call xmDumpvars +all +cell_instance=0 +power</code> <code>call xmDumpSVA</code> |
|------------|--|

| | |
|-----------------------------|---|
| System Call | <pre>\$xmDumpLog(); \$xmlDumpvars("+cell_instance =0", "+all"); \$xmlDumpvars("+cell_instance =0", "+all", "+power"); \$xmlDumpSVA();</pre> |
| ida_probe equivalent | <pre>ida_probe -log ida_probe -wave -wave_probe_args="[scope -tops] -all -memories -depth to_cells -power"</pre> |

Dumping Commands for Xcelium VWDB Flow

The following dumping tasks are available for Xcelium:

- `$xmDumpfile`
- `$xmDumpvars`
- `$xmDumpMDA`
- `$xmDumpon/$xmDumpoff`
- `$xmDumpvarsByFile`
- `$xmDumpMDAByFile`
- `$xmSuppress`
- `$xmDumpFinish`
- `$xmDumpflush`
- `$xmDumpSVA`
- `$xmDumpLog`
- `$xmDumpRandDebug`

`$xmDumpfile`

Opens an Xcelium VWDB with the name specified.

Specifies the name of the Xcelium Waveform database. This command is only effective before any dumping task is executed.

| | |
|------------------|---|
| Syntax | <code>\$xmDumpfile("File_name" File_name_var, Size_limit Size_limit_var);</code> |
| Arguments | <ul style="list-style-type: none"> • "File_name": specifies the DB file name in the string • File_name_var: specifies the DB file name in a variable • Size_limit: specifies the maximum size of DB in a constant number (unit: MB) <ul style="list-style-type: none"> ◦ the minimal valid value is 10 • Size_limit_var: specifies the maximum size of DB in a variable (unit: MB) <ul style="list-style-type: none"> ◦ the minimal valid value is 10 |
| Example | <pre>\$xmDumpfile("mywave"); Or string wave_db = "mywave"; \$xmlDumpfile(wave_db); Create a Database named mywave.db for Xcelium VWDB flow.</pre> |

Notes:

- If `$xmDumpfile` is called without specifying a filename, it will throw a warning message.
- This command is effective only before any dumping task is executed, for example, `$xmDumpvars`, and is ignored if specified after them. For `$xmDumpvars`, `$xmDumpMDA`, `$xmDumpvarsByFile`, and `$xmDumpMDAByFile`, you can use the option `" +vwdbfile+filename "` to specify the target DB for dumping. It is equivalent to using `$xmDumpfile` and then `$xmDumpvars`.

\$xmDumpvars

This command dumps the signals' value changes (VC) to the database.

| Syntax | <code>\$xmDumpvars([depth, "level=", depth_var,] [instance "instance=", instance_var] [, "option" , "option=", option_var]*);</code> | | | | | | | | | | |
|------------------|--|-------|-----------|-------------|--------------------------------|---|---------------------------------------|---|--|---|---|
| Arguments | <ul style="list-style-type: none"> • depth: specifies the levels of hierarchy to dump. If the instance argument is a signal, only the signal will be dumped. If the instance is a module instance, the depth indicates the hierarchy levels for the subsequent modules. All the scopes inside a module are in the same depth as this module. The default value is 0, and other values are explained in the table below: <table border="1"> <thead> <tr> <th>Value</th><th>Signifies</th></tr> </thead> <tbody> <tr> <td>0 (default)</td><td>dump all signals at all levels</td></tr> <tr> <td>1</td><td>dump all signals in the current level</td></tr> <tr> <td>2</td><td>dump all signals in the current level and all scopes one level below the current level</td></tr> <tr> <td>n</td><td>dump all signals in the current level and all scopes n-1 levels below the current level</td></tr> </tbody> </table> • "level=": specifies the next argument as a number in the variable. • instance: specifies the module, module instance, or signals in the full hierarchy format to be dumped. (Specified scope, part-select, bit-select, and struct/union member will be ignored.) • "instance=": specifies the content of the next argument as a module or signal in the variable. (Specified scope, part-select, bit-select, and struct/union member will be ignored.) <p>For options <code>instance/ "instance="</code>, specifying members of scope, part-selection, bit-selection, struct, or union is not supported.</p> | Value | Signifies | 0 (default) | dump all signals at all levels | 1 | dump all signals in the current level | 2 | dump all signals in the current level and all scopes one level below the current level | n | dump all signals in the current level and all scopes n-1 levels below the current level |
| Value | Signifies | | | | | | | | | | |
| 0 (default) | dump all signals at all levels | | | | | | | | | | |
| 1 | dump all signals in the current level | | | | | | | | | | |
| 2 | dump all signals in the current level and all scopes one level below the current level | | | | | | | | | | |
| n | dump all signals in the current level and all scopes n-1 levels below the current level | | | | | | | | | | |
| Options | <ul style="list-style-type: none"> • " +vwdbfile+filename ": specifies the DB name. If not specified, the default VWDB will be named <code>"ida.db"</code>. • " +all ": dump all signals, including memories, packed array, structure, and packed union, in the specified depth and given scope. • " +IO_Only ": indicates dumping only I/O ports of a given module. The <code>" +IO_Only "</code> and <code>" +ports "</code> options behave the same, so you can choose any. <ul style="list-style-type: none"> ◦ priority is higher than <code>+vwdb+trace_process</code> and <code>+vwdb+functions</code> <ul style="list-style-type: none"> ■ In VHDL, <code>+vwdb+io_only +vwdb+trace_process</code> will filter non-io variables inside the process. ■ In SystemVerilog, <code>+vwdb+io_only +vwdb+functions</code> will filter non-io variables inside functions or tasks. • " +ports ": indicates dumping only I/O ports of the given module • " +Reg_Only ": indicates dumping only register-type signals in the given scope. The <code>" +Reg_Only "</code> and <code>" +reg "</code> options behave the same, so you can choose any. • " +reg ": indicates dumping only reg-type signals in a given scope • " +mda ": indicates dumping multiple dimension arrays in the given scope. • " +packedmda ": indicates dumping packed signals in the given scope • " +packedmda+struct ": indicates dumping packed signals and struct signals (the struct and the struct in packed MDA) in the | | | | | | | | | | |

given scope

- **"`+struct`"**: indicates dumping all struct signals in the given scope.
- **"`+cell_instance=<mode>`"**: enable or disable dumping of cell instances, including all the cells that are defined using "`celldefine`" and cell libraries including by "`-v/-y`."

| Value | Signifies |
|-------------|---|
| 0 | disable cell instance dumping; skip the cell instance information |
| 1 | dump all port information |
| 2 (default) | dump all cell information |
| 3 | skip dumping ' <code>celldefine</code> cells, keep dumping <code>-y/-v</code> cells |
| 4 | skip dumping ' <code>celldefine</code> cells but keep dumping ports info, keep dumping <code>-y/-v</code> cells |

- **"`+functions`"**: specifies dumping all functions and tasks in a given scope
- **"`+parameter`"**: specifies dumping all parameters in a given scope
- **"`+trace_process`"**: indicates the dumping of VHDL variables under processes. (for VHDL design only)
- **"`+packed_limit=<value>`"**: specifies the size limitation for packed arrays. The unit of measurement is bits.
 - The default value is 4*1024 bits
 - The maximum value that can be set is 2^32 bits
 - If `<value>` exceeds the maximum value, the size limitation will be bound at the maximum value
 - If `<value>` is 0, it indicates "no size limitation."
- **"`+unpacked_limit=<value>`"**: specifies the size limitation for unpacked arrays. The unit of measurement is the element count of the array.
 - The default value is 16*1024 elements
 - The maximum value that can be set to 2^22 elements
 - If `<value>` exceeds the maximum value, the size limitation will be bound at the maximum value
 - If `<value>` is 0, it indicates "no size limitation."
- **"`+power`"**: Enable dumping the all-power nets except for the saved value of state retention (shadow register) in low-power simulation
- **"`+power_sr_save`"**: Enable dumping the saved value of state retention (shadow register) variables in the low-power simulation
- **"`+dynamic`"**:
 - Dump dynamic objects (including class, signals underclass, queue, dynamic array, associative array, and string)
 - ("`+all`" do not invoke dumping dynamic objects)
- **"`+uvm`"**:
 - Includes UVM (or OVM) base class objects with class object probes.
 - (By default, the option "`+dynamic`" filters out UVM (or OVM) base class to limit the size of DB.)
- **"`+sc`"**:
 - Dump SystemC scopes and variables
- **"`+sc_process`"**:
 - Dump SystemC processes
- **"`+sc_future_process`"**:

- Dump new SystemC processes created in the scope(s) after the probe is created
- "+sc_activation":
 - Create a summary probe of the SystemC processes in the scope(s)
- "+skip_cell_instance=<mode>":
 - "+cell_instance" has a higher priority than "+skip_cell_instance."

Mapping table

| - | VWDB Option (Preferred) | VWDB Option |
|---|----------------------------|-----------------------|
| skip all cell info | +cell_instance=0 | +skip_cell_instance=1 |
| skip cell content, but keep dumping ports | +cell_instance=1 | +skip_cell_instance=2 |
| (Default) Dump all cell info | +cell_instance=2 | +skip_cell_instance=0 |
| skip dumping `celldefine` cells, keep dump -y/-v cells | +cell_instance=3 | +skip_cell_instance=3 |
| skip dumping `celldefine` cells but keep dumping ports info, keep dumping -y/-v cells | +cell_instance=4 | +skip_cell_instance=4 |

Examples

```
$xmDumpvars();
Dump all signals in module instances under the top module to ida.db database (the default database name).

$xmlDumpvars(0, dut);
Dump all signals of all levels under "dut" to ida.db database.

$xmlDumpvars(0, dut, "+vwdbfile+mydb");
Dump all signals of all levels under "dut" to mydb.db database.

$xmlDumpvars(1, dut.i_mcu.ctrl0);
Dump all current level signals "dut.i_mcu.ctrl0" to ida.db database.

Verilog:
reg [7:0] depth_level = 3;
$xmlDumpvars("level=", depth_level, dut);

Dump all current level signals for module instances "dut" and all signals under 1 and 2 levels from "dut."

string opt_str = "+reg_only";

$xmlDumpvars("option=", opt_str);
Dump all signals of data type "register" to the ida.db database.
```

\$xmDumpMDA

\$xmDumpMDA

Dumps value changes of Multi-dimensional Array (MDA) signals. All memories and MDAs within the specified instance are dumped.

| | |
|-----------------|--|
| Syntax | <code>\$xmDumpMDA([depth, "level=", depth_var,] [instance "instance=", instance_var] [, "option" "option=", option_var]*);</code> |
| Argument | Refer to <code>\$xmDumpvars</code> |
| Options | <ul style="list-style-type: none"> • "+vwdbfile+filename": specifies the Xcelium VWDB name. If not specified, the default name is <code>ida.db</code>. • "+cell_instance=mode": enable or disable dumping of cell instances. <p>Refer to the Mapping Table in <code>\$xmDumpvars</code>.</p> |

| | |
|----------------|---|
| Example | <pre>reg [7:0] mem [63:0][127:0]; // The only MDA variable in the module top \$xmlDumpMDA(0, top); Dump all signals in memory to the DB. \$xmlDumpMDA(0, top, "+vwdbfile+mywave", "+cell_instance=0"); Dump all memories and multi-dimensional arrays under the "top" scope, but skip all cell instances under this scope.</pre> |
|----------------|---|

The comparison between `$xmDumpvars` with options and `$xmDumpMDA` dumping rules is shown in the table below:

| Option | reg [] M; (packed 1d) | reg [][]M; (packed 2d) | reg M[]; (unpacked 1d) | reg M[][]; (unpacked 2d) | reg []*M[]; |
|---------------------------|-----------------------|------------------------|------------------------|--------------------------|-------------|
| <code>\$xmDumpvars</code> | | | | | |
| None | Yes | | | | |
| +mda | Yes | Yes | Yes | Yes | Yes |
| +packedmda | Yes | Yes | | | |
| +all | Yes | Yes | Yes | Yes | Yes |
| <code>\$xmDumpMDA</code> | | | | | |
| None | | Yes | Yes | Yes | Yes |

`$xmDumpon/$xmDumpoff`

These two commands turn waveform dumping action on and off during simulation.

`$xmDumpon` and `$xmDumpoff` have the highest priority and can override all other dumping commands such as `$xmDumpvars`, `$xmDumpMDA`, `$xmDumpvarsByFile` or `$xmDumpMDAByFile`.

- If multiple DBs are open for dumping at one simulation run, `$xmDumpon` and `$xmDumpoff` affect every open DB dumping task; in other words, `$xmDumpon` and `$xmDumpoff` can only affect a specific DB by specifying the file name.
- You don't need to call `$xmDumpon` to enable dumping after setting a dumping task (`$xmDumpvars`), as it is enabled by default. You only need to re-enable it after you call `xmDumpoff` to continue dumping.

| | |
|-----------------|--|
| Syntax | <pre>\$xmDumpon(["option" option_var]) \$xmlDumpoff(["option" option_var])</pre> |
| Argument | " <code>+vwdbfile+filename</code> ": specifies the on/off command to be applied to the specific DB. |
| Example | <pre>\$xmDumpoff(); Stop dumping signals to all opened DBs. \$xmlDumpon(); Resume dumping signals to all opened DBs. \$xmlDumpoff("+vwdbfile+mywave1"); \$xmlDumpoff("+vwdbfile+mywave2"); Stop dumping signals to mywave1.db and mywave2.db database for VWDB \$xmlDumpon("+vwdbfile+mywave1"); Resume dumping signals to mywave1.db for VWDB. Nothing will happen if the task is already in the "dumpson" status.</pre> |

`$xmDumpvarsByFile`

This command dumps several depths and scope signals defined in a text file.

`$xmDumpvarsByFile` can be invoked more than once to dump several scopes/signals defined in different text files to specific Xcelium DBs. If no

name is specified, the default VWDB name is `ida.db`. You can specify the name in `$xmDumpvarsByFile` with the argument `" +vwdbfile+filename"` to a designated VWDB filename.

| | |
|------------------|---|
| Syntax | <code>\$xmDumpvarsByFile("textFileName" textFileName_var [, "option" option_var])</code> |
| Arguments | <ul style="list-style-type: none"> <code>"textFileName"</code>: specifies a string for the text file that consists of depth and scope <code>textFileName_var</code>: specifies the text file consisting of depth and scope as a variable |
| Option | <code>" +vwdbfile+filename"</code> : specifies the waveform DB filename. If not specified, the default file name <code>"ida.db"</code> is for VWDB. |
| Example | <p><i>dump_cases.list</i> file content:</p> <pre># ["option"]* [depth] instance_name -- comment +reg 0 dut.i_mcu +parameter 2 dut +cell_instance=0 1 dut.i_ccu</pre> <p><code>\$xmDumpvarsByFile("./dump_cases.list");</code> Dump signals are specified by each line in <i>dump_cases.list</i> file.</p> <pre>string dumpingList = "dumping.list"; string dumping_opt = "+vwdb+mywave";</pre> <p><code>\$xmDumpvarsByFile(dumpingList, dumping_opt);</code> Dump signals specified by each line in the <i>dumping.list</i> to <i>mywave.db</i> database.</p> |

- If this dumping task has only one argument, which is either `"textFileName"` or `textFileName_var`, the waveform DB will be the default `"ida.db"` or the last DB created in `$xmDumpfile` or `" +vwdbfile+fileName"` in the previous dumping task.
- The file format for the text file is as follows:
 - `#` at the beginning indicates the whole line is a comment
 - Otherwise, each line has the following syntax
`["option"]* [depth] instance_name`
 - `instance_name`: specifies the module, module instance, or signals in the full hierarchy format to be dumped. Specified scope, part-select, bit-select, and struct/union member will be ignored.
- The option string defined in the text file is only applied and effective for this dumping task. For the valid option string, please refer to `$xmDumpvars`. The depth and hierarchy `instance_name` is also similar to the argument in `$xmDumpvars`.

\$xmDumpMDAByFile

Dumps the value changes of the multi-dimensional array (MDA) signals in a text file to a waveform DB.

`$xmDumpMDAByFile` can be invoked more than once to dump several scopes and signals defined in different text files to specific DBs.

There are two ways to specify target DBs:

- No DB name is specified in `$xmDumpMDAByFile` (only one argument is needed). If no name is specified, the default name is `ida.db`, or the previously specified one (first `$xmDumpfile`) is taken as the default.
- Specify the DB name in `$xmDumpvars` (two arguments needed). The argument `+vwdbfile+filename` can be used to specify the designated DB name.

| | |
|------------------|--|
| Syntax | <code>\$xmDumpMDAByFile("textFileName" textFileName_var [, "option" option_var])</code> |
| Arguments | <ul style="list-style-type: none"> <code>"textFileName"</code>: specifies a string for the text file that consists of depth and scope <code>textFileName_var</code>: specifies the text file consisting of depth and scope as a variable |
| Option | <code>" +vwdbfile+filename"</code> : specifies the waveform DB name. If not specified, the default file name available is <code>"ida.db"</code> . |

Example

```
dumping_cases.list file content:
# [option_in_file] [depth] instance
0 top.i_dut
+cell_instance=1 0 top

$xmlDumpMDAByFile("dumping_cases.list", "+vwdbfile+mywave");
Dump MDA signals by specification in each line of dumping_cases.list file to mywave.db database.
```

The file format for the text file is as follows:

- Any content after "#" and before the end of the line is treated as a code comment. In other words, the items after "#" are ignored.
- [option_in_file] [depth] instance
- instance_name specifies the module, module instance, or signals in the full hierarchy format to be dumped. Specified scope, part-select, bit-select, and struct/union member will be ignored.
- option_in_file:
 - "+cell_instance=mode": enables/disables dumping of cell instances.

Refer to the [Mapping Table](#) in `$xmDumpvars`.

\$xmSuppress

To exclude certain design hierarchies or modules being dumped by `xmDumpvars`, `xmDumpvarsByFile`, `xmDumpMDA`, or `$xmDumpMDAByFile` commands, you can use `$xmSuppress` to suppress them by file name or target instances passed as arguments. The supported targets could be modules, instances, or signals in a full hierarchical format.

The task needs to be invoked before by `xmDumpvars`, `xmDumpvarsByFile`, `xmDumpMDA`, or `$xmDumpMDAByFile`. All suppression lists will be reset after `$xmDumpfinish`.

Syntax

```
$xmSuppress( "suppress_file" )

$xmlSuppress( instance[,instance]* )

$xmlSuppress( "file=", file_variable )

$xmlSuppress( "instance=", instance_variable[, instance_variable]* )

$xmlSuppress( "module_file=", "filename" )

$xmlSuppress( "module_base=", "module_name"[, "module_name"]* )

$xmlSuppress( "signal_prefix=", "prefix_name"[, "prefix_name"]* );
```

| | |
|-----------------------------|--|
| Arguments | <ul style="list-style-type: none"> • "suppress_file": specifies the file name containing the names of to-be-suppressed signals, modules, and instances. • instance: specifies the to-be-suppressed modules, module instances, or signals in the full hierarchy format. (Specify scope, part-select, bit-select, or member of struct and union will be ignored) • "file=",file_variable: specifies the file name in a variable. • "instance=",instance_variable: specifies the to-be-suppressed modules, module instances, or signals in a variable. (Specified scope, part-select, bit-select, and struct/union member will be ignored) • "module_file=", "filename": specifies the file name containing the names of modules to be suppressed. • "module_base=", "module_name": specifies the names of modules to be suppressed. • "prefix_name": specifies the signal prefix pattern. Signals that match this pattern are not dumped. <ul style="list-style-type: none"> ◦ <code>\$xmSuppress</code> must be specified and invoked before <code>\$xmDumpvars</code>, <code>\$xmDumpvarsByFile</code>, <code>\$xmDumpMDA</code>, and <code>\$xmDumpMDAByFile</code>; otherwise, <code>\$xmSuppress</code> will not work. ◦ The setting of <code>\$xmSuppress</code> is reset after <code>\$xmDumpFinish</code> is executed. ◦ When the <code>suppress_file</code> name is the same as the instance name, <code>suppress_file</code> will be chosen for the <code>\$xmSuppress</code> command. |
| Example TCL | <pre>call xmSuppress "suppress.list" Suppress dumping for all signals in each of the scopes listed in <i>suppress.list</i> file. call xmSuppress top.a top.b Suppress dumping for all signals under the <i>top.a</i> and <i>top.b</i> scope. call xmSuppress top.abc Suppression failure. Scope specification for suppression is invalid. module top; always @(posedge clk) begin:abc op_a = op_b; end endmodule</pre> |
| Example Dumping Task | <pre>\$xmSuppress("signal_prefix=", "_zy", "_zz", "zz_", "zy_");</pre> |

\$xmDumpFinish

This command stops all dumping tasks of signals and closes all opened DBs in the current simulation. Although all DBs will be stopped and closed automatically at the end of the simulation, this command can explicitly force to stop dumping, close all DBs in the current simulation session, and continue the simulation.

Another DB file can be created after calling `$xmDumpFinish`. New dumping data cannot be appended to a closed DB file. If a closed DB file name is specified again during the same simulation run, it will be renamed automatically.

| | |
|----------------|--|
| Syntax | <code>\$xmDumpFinish;</code> |
| Example | <pre>\$xmDumpFinish; Close all opened VWDB databases in the current simulation and stop dumping all signals.</pre> |

| | |
|------------------------|---|
| Example TCL | <pre>call xmDumpfile abc call xmDumpfile def run 10ns call xmDumpFinish</pre> <p>In VWDB flow, stop dumping all signals and close "abc.db" and "def.db".</p> <pre>call xmDumpfile abc</pre> <p>The specific DB name "abc" (abc.db) is the same as the previous one, which is already closed by xmDumpFinish, it will be renamed to "abc_r0.db" to avoid conflict.</p> |
|------------------------|---|

\$xmDumpflush

This command forces the signal values and status, temporarily buffered, to be flushed to the waveform DB immediately while the simulation is running. Therefore, the current simulation results can be checked whenever necessary instead of waiting for the simulation to be completed or for the internal buffer to fill.

| | |
|----------------|--|
| Syntax | <code>\$xmDumpflush;</code> |
| Example | <pre>\$xmDumpFlush;</pre> <p>Force to flush all dumped signal data in the buffer to get the newest status.</p> |

\$xmDumpSVA

This command dumps the SVA results to the specified DBs. The results of SVA assertions within the specified scopes and/or modules can be dumped and saved to the same DB files that contain dumped results for design signals or dumped to the designated VWDB files.

| | |
|------------------|---|
| Syntax | <code>\$xmDumpSVA([depth, ["level=",depth_var,] [instance "instance=", instance_var] [, "option" , "option=",option_var]*);</code> |
| Arguments | <ul style="list-style-type: none"> <code>depth</code>: specifies the hierarchy levels to dump for the subsequent scopes. <code>"level="</code>: specifies the next argument is a number in variable <code>depth_var</code>: specifies how many levels of hierarchy to dump for the subsequent scopes in a variable. <code>instance</code>: specifies the instance name for which all SVA assertions below it are dumped. <code>"instance="</code>: identifies the content of the next argument as a module scope or signal. <code>instance_var</code>: gives the module scope or signal represented by <code>instance_var</code> that specifies the object to dump. |
| Options | <ul style="list-style-type: none"> <code>"+vwdbfile+filename"</code>: specifies the waveform DB filename. If not specified, the default file name is "ida.db" for the VWDB. <code>"+functions"</code>: indicates dumping all functions and tasks in a given scope <code>"+cell_instance=<mode>"</code>: enable or disable dumping of cell instances, including <code>`celldefine</code> and <code>-v/-y</code> cell libraries. Refer to the Mapping Table in <code>\$xmDumpvars</code>. <code>"+assertdebug"</code>: Dump additional debug for assertions with transactions for all attempts and local variables wherever applicable. |

Example

```
initial begin

$xmlDumpvars("+vwdbfile+test1");

$xmlDumpSVA(1,top); //dump current layer sva to test1.db

end

$xmlDumpSVA; //Dump all SVA assertions from the design top and its descendants.

$xmlDumpSVA(0, test_top); //Dump all SVA assertions under "test_top" and its descendants

$xmlDumpSVA(test_top, "+vwdbfile=SVA"); // dump all sva under test_top and its descendants to the
waveform DB file, SVA.db
```

Related Options Mapping Table

| Task/TCL Option | Environment Variable | Simulator Argument |
|-----------------------|-------------------------|----------------------------|
| +vwdbfile+<filename> | CDNS_VWDB_FILE | +vwdbfile+<filename> |
| +cell_instance=<mode> | CDNS_VWDB_CELL_INSTANCE | +vwdb+cell_instance=<mode> |
| +functions | CDNS_VWDB_FUNCTIONS | +vwdb+functions[=on off] |
| +assertdebug | CDNS_VWDB_ASSERTDEBUG | +vwdb+assertdebug[=on off] |

\$xDumpLog

This command dumps Smart Log to the database.

| | |
|----------------|---|
| Syntax | <code>\$xDumpLog(["option", "option=", option_var,]*);</code> |
| Options | <ul style="list-style-type: none"> • "+log_objects": <ul style="list-style-type: none"> ◦ Allow recording objects in messages • "+log_verbosity=NONE LOW MEDIUM HIGH FULL": <ul style="list-style-type: none"> ◦ Set run's verbosity and level can be NONE, LOW, MEDIUM, HIGH, or FULL. (default is FULL) • "+log_destination_file=stdout all": <ul style="list-style-type: none"> ◦ Record log information printed to stdout only or record log information for all destination files, including stdout. (default is stdout) • "+off": <ul style="list-style-type: none"> ◦ Pausing smart log dumping. For example: <ul style="list-style-type: none"> ▪ call xxDumpLog run 10ns call xxDumpLog +off |

Related Options Mapping Table

| Task/TCL Option | Environment Variable | Simulator Argument |
|------------------------------|--------------------------------|---|
| +vwdbfile+<filename> | CDNS_VWDB_FILE | +vwdbfile+<filename> |
| +log_objects | CDNS_VWDB_LOG_OBJECTS | +vwdb+log_objects[=on off] |
| +log_verbosity=<mode> | CDNS_VWDB_LOG_VERBOSITY | +vwdb+log_verbosity=NONE LOW MEDIUM HIGH FULL |
| +log_destination_file=<mode> | CDNS_VWDB_LOG_DESTINATION_FILE | +vwdb+log_destination_file=stdout all |

| | | |
|------|--|--|
| +off | | |
|------|--|--|

\$xmDumpRandDebug

This command dumps information for Randomization Debugging.

| | |
|----------------|---|
| Syntax | \$xmDumpRandDebug; |
| Example | \$xmDumpRandDebug; Enable the waveform dumping for debugging randomization |

Priority of Specifying VWDB Probe Options

The Xcelium VWDB flow supports the following three methods for specifying options:

1. On the simulator command line
2. Using an environment variable
3. By a TCL dumping or System Calls command

If the same option is set using multiple methods, the priority will be Method 1 > Method 2 > Method 3. For example, the Xcelium dump file name can be set using one of the following methods:

- xrun +vwdbfile+HighPriority
- setenv CDNS_VWDB_FILE MidPriority
- \$xmDumpvars ("vwdbfile+LowPriority");

If all three of the following options are specified at the same time, the dumping file will be named "HighPriority": That is, in this example, the generating name priority is "high_priority" > "mid_priority" > "low_priority."

If multiple simulator arguments or task/Tcl options are specified simultaneously, the first one will be adopted.

If you use an umbrella option like +all, you cannot partially disable a few of its subparts using a CLI option.

| Simulator Argument (Priority 1) | Environment Variable (Priority 2) | Task/TCL Option (Priority 3) |
|---------------------------------|-----------------------------------|--|
| +vwdbfile+<filename> | CDNS_VWDB_FILE | +vwdbfile+<filename> |
| +vwdb+all [=on off] | CDNS_VWDB_ALL | +all |
| +vwdb+io_only | CDNS_VWDB_IO_ONLY | +IO_Only |
| +vwdb+ports | CDNS_VWDB_PORTS | +ports |
| +vwdb+reg_only | CDNS_VWDB_REG_ONLY | +Reg_Only |
| +vwdb+reg | CDNS_VWDB_REG | +reg |
| +vwdb+mda [=on off] | CDNS_VWDB_MDA | +mda |
| +vwdb+packedmda [=on off] | CDNS_VWDB_PACKEDMDA | +packedmda |
| +vwdb+struct [=on off] | CDNS_VWDB_STRUCT | +struct |
| +vwdb+cell_instance=<mode> | CDNS_VWDB_CELL_INSTANCE | +cell_instance=<mode> |
| +vwdb+functions [=on off] | CDNS_VWDB_FUNCTIONS | +functions |
| +vwdb+strength [=on off] | CDNS_VWDB_STRENGTH | See Simulator Argument for Global Settings for the description |

Xcelium Simulator Tcl Command Reference

Xcelium VWDB Probing

| | | |
|---|------------------------------|--|
| | | |
| +vwdb+parameter [=on off] | CDNS_VWDB_PARAMETER | +parameter |
| +vwdb+trace_process [=on off] | CDNS_VWDB_TRACE_PROCESS | +trace_process |
| +vwdb+packed_limit=<value> | CDNS_VWDB_PACKED_LIMIT | +packed_limit=<value> |
| +vwdb+unpacked_limit=<value> | CDNS_VWDB_UNPACKED_LIMIT | +unpacked_limit=<value> |
| | | +packedmda+struct |
| +vwdb+skip_cell_instance=<mode> | CDNS_VWDB_SKIP_CELL_INSTANCE | +skip_cell_instance=<mode> |
| +vwdb+dump_limit=<size> | CDNS_VWDB_DUMP_LIMIT | See Simulator Argument for Global Settings for the description |
| +vwdb+time_precision=<timeunit> | CDNS_VWDB_TIME_PRECISION | See Simulator Argument for Global Settings for the description |
| +vwdb+glitch=<number> | CDNS_VWDB_ENV_MAX_GLITCH_NUM | See Simulator Argument for Global Settings for the description |
| +vwdb+seq_mode= none vc | CDNS_VWDB_SEQ_MODE | See Simulator Argument for Global Settings for the description |
| +vwdb+nowarn [=on off] | CDNS_VWDB_NOWARN | See Simulator Argument for Global Settings for the description |
| +vwdb+log_file=<log_file_name> | CDNS_VWDB_LOG_FILE | See Simulator Argument for Global Settings for the description |
| +vwdb+power [=on off] | CDNS_VWDB_POWER | +power |
| +vwdb+power_sr_save [=on off] | CDNS_VWDB_POWER_SR_SAVE | +power_sr_save |
| +vwdb+force [=on off] | CDNS_VWDB_FORCE | See Simulator Argument for Global Settings for the description |
| +vwdb+dynamic [=on off] | CDNS_VWDB_DYNAMIC | +dynamic |
| +vwdb+uvm [=on off] | CDNS_VWDB_UVM | +uvm |
| +vwdb+sc [=on off] | CDNS_VWDB_SC | +sc |
| +vwdb+sc_process [=on off] | CDNS_VWDB_SC_PROCESS | +sc_process |
| +vwdb+sc_future_process [=on off] | CDNS_VWDB_FUTURE_PROCESS | +sc_future_process |
| +vwdb+sc_activation [=on off] | CDNS_VWDB_ACTIVATION | +sc_activation |
| +vwdb+assertdebug [=on off] | CDNS_VWDB_ASSERTDEBUG | +assertdebug |
| +vwdb+log_objects [=on off] | CDNS_VWDB_LOG_OBJECTS | +log_objects |
| +vwdb+log_verbosity=NONE LOW MEDIUM HIGH FULL | CDNS_VWDB_LOG_VERBOSITY | +log_verbosity=<mode> |

| | | |
|--|--------------------------------|--|
| | | |
| +vwdb+log_destination_file=stdout all | CDNS_VWDB_LOG_DESTINATION_FILE | +log_destination_file=<mode> |
| | | +off |
| +vwdb+dump_on+time_low[+time_high] or +vwdb+dump_on+time_low[+time_unit] | | See Simulator Argument for Global Settings for the description |
| +vwdb+dump_off+time_low[+time_high] or +vwdb+dump_off+time_low[+time_unit] | | See Simulator Argument for Global Settings for the description |
| +vwdb+local_var[=on off] | CDNS_VWDB_LOCAL_VAR | See Simulator Argument for Global Settings for the description |
| +vwdb+local_var_verbose[=on off] | CDNS_VWDB_LOCAL_VAR_VERBOSE | See Simulator Argument for Global Settings for the description |
| +vwdb+no_shm_dir[=on off] | CDNS_VWDB_NO_SHM_DIR | See Simulator Argument for Global Settings for the description |

Simulator Argument for Global Settings

This table lists the simulator arguments for global settings with their descriptions.

| Simulator Argument | Description |
|----------------------------------|--|
| +vwdb+force[=on off] | Enable force/release/deposit recording (external force/release/deposit from tcl/vpi only) |
| +vwdb+strength[=on off] | Indicates dumping of wire strength |
| +vwdb+glitch=<number> | Specify the glitch dumping number <ul style="list-style-type: none"> If <number>=0: all glitch values will be stored If <number>=1 (default): only stable values will be stored If <number> is greater or equal to 2 is not available, a warning message will be printed, and the <number> will be set to 1 |
| +vwdb+seq_mode=<mode> | Specify the sequence mode <ul style="list-style-type: none"> <mode>=none: (default) Do not dump sequence <mode>=vc: Dump sequence number |
| +vwdb+time_precision=<time_unit> | Specify the time precision of DB <ul style="list-style-type: none"> If not set, use the default time precision of the current simulation <time_unit> can be 1fs 10fs 100fs 1ps 10ps 100ps 1ns 10ns 100ns 1us 10us 100us 1ms 10ms 100ms 1s 10s 100s |
| +vwdb+dump_limit=<size> | Specify the size limit of DB <ul style="list-style-type: none"> The default of <size> is 0 (unlimited), min value of <size> is 10, the unit is MB |
| +vwdb+nowarn[=on off] | Disable all the VWDB-related warnings in the screen and simulation log |

Xcelium Simulator Tcl Command Reference

Xcelium VWDB Probing

| | |
|--|--|
| +vwdb+log_file=<log_file_name> | <p>Set the log filename for VWDB</p> <ul style="list-style-type: none"> The default of <log_file_name> is "cdns_dump.log" (+xdmp_log_file=<log_file_name> and CDNS_XDMP_LOG_FILE are deprecated since 22.11a.) |
| +vwdb+dump_on+time_low[+time_high] or +vwdb+dump_on+time_low[+time_unit] | <ul style="list-style-type: none"> Turn on dumping commands at the specified time {time_high,time_low}. Default: no dump-on control time_low or time_high is 32bit integers, time_unit can be s ms ns ps fs, e.g., +vwdb+dumpon=150ps, +vwdb+dumpon+150,1 ((32'd1, 32'd150) = 4,294,967,446 sim time) When the time_unit is not specified, the simulation time unit will be used as the default time unit SHM hybrid mode does not support this option |
| +vwdb+dump_off+time_low[+time_high] or +vwdb+dump_off+time_low[+time_unit] | <ul style="list-style-type: none"> Turn off dumping commands at the specified time {time_high,time_low}. Default: no dump-off control time_low or time_high is 32bit integer, time_unit can be s ms ns ps fs, e.g., +vwdb+dumppoff=150ps, +vwdb+dumppoff+150,1 ((32'd1, 32'd150) = 4,294,967,446 sim time) When the time_unit is not specified, the simulation time unit will be used as the default time unit SHM hybrid mode does not support this option |
| +vwdb+local_var[=on off] | Enable or disable the dumping of local variables |
| +vwdb+local_var_verbose[=on off] | Enable or disable the verbosity for local variables |
| +vwdb+no_shm_dir[=on off] | Do not create an SHM part in native VWDB dumping (unified probe) at the beginning of creating VWDB, but still create the SHM part when unsupported signals (e.g., dynamic objects, systemC, etc.) are probed |

General Notes

- " +vwdb+<option>=on|off" means users can specify "+vwdb+<option>=on" or "+vwdb+<option>=off"
- " +vwdb+<option>[=on|off]" means users can specify "+vwdb+<option>" or "+vwdb+<option>=on" or "+vwdb+<option>=off"
- " +vwdb+<option>" is equal to "+vwdb+<option>=on"
- Simulator Arguments, except for +vwdb+<option>, +vwdb+<option>=on, +vwdb+<option>=On, +vwdb+<option>=oN, +vwdb+<option>=ON, others are viewed as disabled.

The Value Change Dump (VCD) File

A value change dump (VCD) file is an ASCII file that contains information about value changes on selected variables in the design. This file contains header information, variable definitions, and the value changes for all specified variables.

The Verilog LRM describes two types of VCD files: *four-state* and *extended*. A four-state VCD file is one that represents variable changes in 0, 1, x, and z with no strength information. With Xcelium, four-state VCD file generation follows two different use models. One for Verilog designs and another for VHDL designs. For detailed information on a particular use model, click one of the links below.

- [Generating a VCD File for a Verilog Design](#)
- [Generating a VCD File for a VHDL Design](#)

You can only dump objects that have read access. If you specify a scope as an argument to the `probe` command, objects within that scope that do not have read access are excluded from the dump, and the simulator prints a warning message. In case of a `probe` command, if you specify an individual variable that does not have read access, the simulator will print an error message.

However, the `$dumpvars` system task automatically adds read access to the complete scope specified as the argument.

Related Topics

- [Access to Simulation Objects](#)
- [The Extended Value Change Dump \(EVCD\) File](#)
- [Generating a Value Change Dump \(VCD\) File for a Mixed-Language Design](#)

Generating a VCD File for a Verilog Design

There are two ways to generate a four-state VCD file for a Verilog design:

- Open a VCD database with the Tcl `database -open -vcd` command, and then probe signals

to the database with the `probe -create -vcd` command.

- Use the VCD system tasks in your HDL code.

For Verilog, you cannot probe arrays of variable data types to a VCD database. This includes Verilog memories, which are one-dimensional arrays of type `reg`. You cannot probe variables declared as multi-dimensional arrays.

Generating a VCD File with Tcl Commands

To generate a VCD file by using Tcl commands:

1. Open a VCD database with the database command. The syntax is as follows:

```
database [-open] dbase_name -vcd
        [-compress | -gzip]
        [-default]
        [-into filename]
        [-maxsize max_size]
        [-timescale timescale_value]
        [-vcdmap vcd_mapping]
```

The following command opens a VCD database named `vcddb`. The filename is `verilog.dump`. The `-timescale` option sets the `$timescale` value in the VCD file to 1 ns. Value changes in the VCD file are scaled to 1 ns.

```
xcelium> database -open -vcd vcddb -into verilog.dump -default -timescale ns
Created default VCD database vcddb
```

2. Probe objects to the database with the `probe` command. The syntax is as follows:

```
probe [-create] [{object | scope_name}...] {-vcd | -database dbase_name}
      [-all]
      [-depth {n | all | to_cells}]
      [-functions]
      [-inputs]
      [-memories]
      [-name probe_name]
      [-outputs]
      [-ports]
      [-screen [-format format_string] [-redirect filename] objects]
      [-tasks]
```

The optional `-create` modifier can be followed by an argument that specifies:

- The object(s) to be traced
- The scope(s) to be traced
- A combination of object(s) and scope(s) to be traced

If you do not specify an argument, the current debug scope is assumed, but you must include an option that specifies which objects to include in the trace (`-all`, `-inputs`, `-outputs`, or `-ports`).

If more than one database is open, you must include an option to specify the database into which you want to dump values. You can do this either by specifying a database name with the `-database` option or by using the `-vcd` option to send the probe to the default VCD database. If no default database is open, the simulator opens a default database called `xcelium.vcd`.

The following `probe` command creates a probe on all ports in the scope `top.counter`. Data is sent to the default VCD database.

```
xcelium> probe -create -vcd top.counter -ports
Created probe 1
```

Generating a VCD File with VCD System Tasks

Several system tasks can be inserted in the source description to create a four-state VCD file. The simulator supports all of the value change dump system tasks specified in the IEEE Verilog Language Reference Manual. This section summarizes the system tasks.

See the *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364-1995 or 1364-2001) for details on these VCD system tasks and for information on the syntax and format of the VCD file.

The following table describes VCD system tasks.

| System Task | Description |
|---------------------------------------|---|
| <code>\$dumpfile ("filename");</code> | Specifies the name of the VCD file. If you do not use this system task to specify a file name, the simulator creates a file called <code>verilog.dump</code> by default. (The default filename is different from that specified in the Verilog LRM, which is <code>dump.vcd</code> .) |

| | |
|--|---|
| <code>\$dumpvars;</code> | <p>Specifies which variables to dump into the VCD file. When invoked with no arguments, <code>\$dumpvars</code> dumps all variables in the design, except those in source-protected regions.</p> |
| <code>\$dumpvars (levels [, list_of_modules_or_variables]);</code> | <p>Specifies which variables to dump into the VCD file. The <code>levels</code> argument indicates the number of hierarchical levels below each specified module instance that <code>\$dumpvars</code> affects. Subsequent arguments specify which scopes of the model to dump to the VCD file. These subsequent arguments can specify entire modules or individual variables within a module.</p> <p>For example, the following invocation dumps all variables within the module <code>top</code>; it does not dump variables in any of the modules instantiated by module <code>top</code>.</p> <pre>\$dumpvars (1, top);</pre> <p>The following invocation dumps all variables in the module <code>top</code> and in all module instances below module <code>top</code> in the design hierarchy.</p> <pre>\$dumpvars (0, top);</pre> <p>The following example shows how the <code>\$dumpvars</code> task can specify both modules and individual variables. This call dumps all variables in module <code>mod1</code> and in all module instances below <code>mod1</code>, along with variable <code>net1</code> in module <code>mod2</code>. Note that the argument 0 applies only to the module instance <code>top.mod1</code>, and not to the individual variable <code>top.mod2.net1</code>.</p> <pre>\$dumpvars (0, top.mod1, top.mod2.net1);</pre> <p>If you want to dump individual bits of a vector net, first make sure that the net is expanded. Declaring a vector net with the keyword <code>scalared</code> guarantees that it is expanded. Using the <code>xmelab -expand</code> command-line option expands all nets, but this procedure is not recommended due to its negative impact on memory usage and performance.</p> <p>Xcelium dumps each bit of an expanded vector net</p> |

| | |
|--|---|
| | <p>individually. That is, each bit has its own identifier code and is dumped only when it changes, not when other bits in the vector change.</p> <div style="border: 1px solid #fde725; padding: 10px; margin-top: 10px;"> <p>You cannot dump part of a vector. For example, you cannot dump only bits 8 through 15 (8:15) of a 16-bit vector. You must dump the entire vector (0:15). In addition, you cannot dump expressions, such as <code>a + b</code>.</p> </div> |
| <code>\$dumpoff;</code> and <code>\$dumpon;</code> | <p>These tasks let you specify the simulation period during which the dump takes place.</p> <p><code>\$dumpoff</code> suspends the dump. A checkpoint is created in which every selected variable is dumped as an <code>x</code> value. To resume the dump, invoke <code>\$dumpon</code>.</p> |
| <code>\$dumpall;</code> | <p>Creates a checkpoint in the dump file that shows the current value of all selected variables.</p> |
| <code>\$dumplimit (filesize);</code> | <p>Sets a limit on the size of the VCD file. The <i>filesize</i> argument specifies the maximum size of the dump file in bytes.</p> |
| <code>\$dumpflush;</code> | <p>Empties the operating system's dump file buffer to ensure that all the data in that buffer is stored in the dump file. You can also use the PLI <code>tf_dumpflush()</code> function in your application program C code.</p> |

Example Source Description Containing VCD Tasks

In the following example, the name of the dump file is `verilog.vcd`. The simulator dumps value changes for all variables in the design. Dumping begins when event `do_dump` occurs. The dumping continues for 500 clock cycles, then stops and waits for event `do_dump` to be triggered again. At every 10000 time steps, the current values of all VCD variables are dumped.

```
module dump;
  event do_dump;
  initial
    $dumpfile("verilog.vcd");           // Default file name is verilog.dump

  initial @do_dump
    $dumpvars;                          // Dump variables in the design

  always @do_dump                      // Begin the dump at event do_dump
  begin
    $dumpon;                           // No effect the first time through
    repeat (500) @(posedge clock);     // Dump for 500 cycles
    $dumpoff;                           // Stop the dump
  end

  initial @(do_dump)
    forever #10000 $dumpall;            // Dump all variables for checkpoint
endmodule
```

Related Topics

- [probe](#)
- [database](#)

Generating a VCD File for a VHDL Design

For VHDL, you can dump all signals, ports, and variables, including those declared as multi-dimensional arrays, to a VCD database, with the following limitations:

- The signal, port, or variable must be of type `std_ulogic`, `bit`, `integer`, `real`, or any user-defined type that is a subset of `std_ulogic`.
- Objects that are declared inside a subprogram cannot be probed.
- Signals and variables that correspond to records are not dumped to the VCD file.
- The type of the object cannot be:
 - A non-standard integer type whose bounds require more than 32 bits to represent
 - Access and file types
 - Any composite type that contains one of the above types

To generate a VCD file for VHDL:

1. Open a VCD database with the Tcl `database -open -vcd` command. The syntax is as follows:

```
database [-open] dbase_name -vcd
        [-compress | -gzip]
        [-default]
        [-into filename]
        [-maxsize max_size]
        [-timescale timescale_value]
        [-vcdmap vcd_mapping]
```

Use the `-timescale` option to set the `$timescale` value in the VCD file to the specified timescale. This option lets you output a different timescale in the VCD file than the timescale being used during simulation. In the output file, the times that are shown for the signal changes reflect the simulation times at the precision that you specify with the `-timescale` option.

The `timescale_value` argument can be:

- fs, 10fs, 100fs
- ps, 10ps, 100ps
- ns, 10ns, 100ns
- us, 10us, 100us
- ms, 10ms, 100ms

For example:

```
xcelium> database -open test_mp -vcd -timescale 10ps
```

Use the `-vcdmap` option to specify a user-defined mapping of VHDL `std_logic` values to the four states for VCD (1, 0, X, Z). By default, the nine values of `STD_LOGIC` (U, X, 0, 1, Z, W, L, H, -) are mapped to (X, X, 0, 1, Z, X, 0, 1, X).

The `vcd_mapping` argument is a string of nine valid VCD values. For example:

```
xcelium> database -open myvcd.vcd -vcd -vcdmap XXXXZ1111
```

You can also specify the mapping by setting the Tcl `vhdl_vcdmap` variable. For example:

```
xcelium> set vhdl_vcdmap XXXXZ1111
```

✔ This `set` command can be included in an input Tcl file that is executed when you invoke `xmsim` with the `-input` option. The `vhdl_vcdmap` variable can also be set in the SimVision GUI. Select *Simulation>Show-Variables*, select the `vhdl_vcdmap` variable, and then set the value to your mapping.

The mapping specified by setting the Tcl `vhdl_vcdmap` variable sets the mapping to be used for all VCD databases that you open. If a VCD mapping is defined by setting a Tcl variable and by specifying the `database -vcd -vcdmap` option, the mapping defined by the `database` command is used.

The following command opens a default VCD database named `vcddb`. The filename is `sim1.dump`. The `-timescale` option sets the `$timescale` value in the VCD file to 1 ns. Value changes in the VCD file are scaled to 1 ns.

```
xcelium> database -open vcddb -vcd -default -into sim1.dump -timescale ns
Created default VCD database vcddb
```

2. Probe signals to the database with the `probe -create -vcd` command. The syntax is as follows:

```
probe [-create] [{object | scope_name}...] {-vcd | -database dbase_name}
      [-all]
      [-depth {n | all | to_cells}]
      [-inputs]
      [-name probe_name]
      [-outputs]
      [-ports]
      [-screen [-format format_string] [-redirect filename] objects]
      [-variables]
```

The optional `-create` modifier can be followed by an argument that specifies:

- The object(s) to be traced
- The scope(s) to be traced
- A combination of object(s) and scope(s) to be traced

If you do not specify an argument, the current debug scope is assumed, but you must include an option that specifies which objects to include in the trace (`-all`, `-inputs`, `-outputs`, or `-ports`).

If more than one database is open, you must include an option to specify the database into which you want to dump values. You can do this either by specifying a database name with

the `-database` option or by using the `-vcd` option to send the probe to the default VCD database. If no default database is open, the simulator opens a default database called `xcelium.vcd`.

The following `probe` command creates a probe on all ports in the scope `:dut`. Data is sent to the default VCD database, `vcddb`.

```
xcelium> probe -create -vcd :dut -ports  
Created probe 1
```

Related Topics

- [Access to Simulation Objects](#)
- [The Extended Value Change Dump \(EVCD\) File](#)
- [Generating a Value Change Dump \(VCD\) File for a Mixed-Language Design](#)
- [database](#)
- [probe](#)

The Extended Value Change Dump (EVCD) File

A value change dump (VCD) file is an ASCII file that contains information about value changes on selected variables in the design. The file contains header information, variable definitions, and the value changes for all specified variables.

The Verilog LRM describes two types of VCD files: *four-state* and *extended*. An extended VCD (EVCD) file is one that represents variable changes in all states and strength information. With Xcelium, extended VCD file generation follows two different use models. One for Verilog designs and another for VHDL designs. For detailed information on a particular use model, click one of the links below.

- [Generating an EVCD File for a Verilog Design](#)
- [Generating an EVCD File for a VHDL Design](#)

To generate an EVCD file, you must provide access to simulation objects by including the `-access rc` option when you elaborate the design (`xrun -access rc` or `xmelab -access rc`, or an access file with `+rcd` set for the particular instance). If you are generating the file by using the Verilog `$dumpports` system task, the elaborator automatically sets the required access.

By default, the signal identifier codes in an EVCD file differ from what is specified in the IEEE standard. The LRM specifies that the identifier code is to be an integer preceded by `<`, which starts at zero and ascends in one unit increments for each port. For example:

```
$scope module board.counter $end
$var port      4 <0          value $end
$var port      1 <1          clock $end
$var port      1 <2          fifteen $end
$var port      1 <3          altFifteen $end
```

Xcelium uses space-efficient identifiers in order to minimize the size of the file. For example:

```
$scope module board.counter $end
$var port      4 !          value $end
$var port      1 "          clock $end
$var port      1 #          fifteen $end
```

```
$var port 1 $ altFifteen $end
```

You can specify the `-use_ieee_dumpport_ids` option on the *xrun* or *xmsim* command line if you want the identifier codes to conform to the LRM when you invoke the simulator.

After generating an EVCD file, use the *xmgentb* utility to generate a testbench.

Related Topics

- [Format of the EVCD File](#)
- [The Value Change Dump \(VCD\) File](#)

Generating an EVCD File for a Verilog Design

You can generate an EVCD file for Verilog by:

- [Using Tcl commands.](#)
- [Using the `\$dumpports` system task in your Verilog source description.](#)

The implementation of the `$dumpports` task differs in some ways from the description of the task in the IEEE Std 1364-2001 standard.

The simulator also supports a `$dumpports_close` system task. It does not support the other extended VCD system tasks described in the IEEE 1364-2001 standard (`$dumpportsoff`, `$dumpportson`, `$dumpportslimit`, and so on).

For Verilog, you cannot probe arrays of variable data types to a VCD database. This includes Verilog memories, which are one-dimensional arrays of type `reg`.

The IEEE SystemVerilog standard has not yet extended the EVCD output format to directly support the many new constructs and scope contexts introduced by SystemVerilog. However, to retain compatibility with existing EVCD readers, the IEEE 1800-2005 standard (Section 24) specifies a set of mapping rules that allow some basic SystemVerilog constructs to be encoded as IEEE standard 1364 Verilog equivalents. See [SystemVerilog and EVCD](#) in the *Debugging SystemVerilog* manual for details on the support for mapping SystemVerilog types to a Verilog type for EVCD dumping.

Generating an EVCD File with Tcl Commands

To generate an EVCD file, you must open an EVCD database and then probe the primary ports of a specified module instance to the database. You can open a database with the `database` command and then probe the ports with the `probe` command. You can also probe objects to a database simply by using the `probe` command. In this case, the `probe` command opens a default EVCD database called `xcelium.evcd` and probes the ports for the specified scope.

Opening an EVCD Database

You can open an EVCD database with the `database` command.

Syntax

```
database [-open] [direction] dbase_name -evcd
        [-compress | gzip]
        [-default]
        [-into filename]
        [-maxsize max_byte_size]
        [-timescale timescale_value]
```

Examples of Tcl database Command

The following command opens an EVCD database called `testoutput`. By default, the associated file is called `db_name.evcd`. In this example, the file will be called `testoutput.evcd`. The file is placed in the current working directory.

```
xcelium> database -open testoutput -evcd
```

The following command opens an EVCD database called `evcd`. The `-default` option specifies that this is the default database for all EVCD signal tracing. The `-into` option specifies that the output file is called `testoutput.evcd`. This file is placed in the current working directory.

```
xcelium> database evcd -evcd -default -into testoutput.evcd
```

The following command opens a default EVCD database named `evcddb`. The filename is `sim.dump`. The `-timescale` option sets the `$timescale` value in the EVCD file to 1 ns. Value changes in the file are scaled to 1 ns. The `-compress` option compresses the output file and generates a file called `sim.dump.Z`.

```
xcelium> database evcddb -evcd -default -into sim.dump -timescale ns -compress
Created default EVCD database evcddb
```

Probing Verilog Objects to an EVCD Database

You can probe Verilog objects, such as ports, in an EVCD database using the `probe` command.

Syntax

```
probe [-create] scope_name [...]
      -evcd
      [[{splitio | simple}]
      [-evcdformat format_number]]
      [-database dbase_name]

      [-all]
      [-depth {n | all | to_cells}]
      [-functions]
      [-inputs]
      [-name probe_name]
      [-outputs]
      [-ports]
      [-tasks]
```

Examples of Tcl probe Command

The following command probes all primary ports of the scope `test_bench.dut` to the default EVCD database opened with a `database -open -evcd -default` command.

```
xcelium> probe -create test_bench.dut -evcd
Created probe 1
```

The following command probes all primary ports of the scope `test_bench.dut` and its subscopes (that is, two levels of depth) to the EVCD database.

```
xcelium> probe test_bench.dut -evcd -depth 2
```

The following command probes all primary ports of the scope `test_bench.dut` to the EVCD database called `testoutput`.

```
xcelium> probe test_bench.dut -evcd -database testoutput
```

You do not have to open a database with the `database` command before probing objects. You can probe objects to a database simply by using the `probe` command. In this case, the `probe` command opens a default EVCD database called `xcelium.evcd` and probes the ports for the specified scope. For example:

```
xcelium> probe -create test_bench.dut -evcd
Created default EVCD database xcelium.evcd
Created probe 1
```

Examples of Using Multiple Commands to Generate a EVCD Database for Verilog

The following examples illustrate the usage of multiple Tcl commands for generating an EVCD database, and all share the Verilog `top.v` source below.

top.v

```
`timescale 1 ns / 1 ns
module test_bench;
    reg AS_3, AS_2, AS_1;
    tri TF;

    // The following 3 continuous assignments are the drivers from the test fixture
    assign (strong1, strong0) TF = AS_1;    // Will make TF have strength 6
    assign (pull1, pull0)     TF = AS_2;    // Will make TF have strength 5
    assign (weak1, weak0)     TF = AS_3;    // Will make TF have strength 3

    initial
    begin
        assign AS_3 = 1'bz;
        assign AS_1 = 1'bz;
        assign AS_2 = 1'bz;
        #10;
        assign AS_3 = 1'bz;
        assign AS_2 = 1'bz;
        assign AS_1 = 0;
        #10;
        assign AS_3 = 0;
        assign AS_1 = 1'bz;
        assign AS_2 = 1'bz;
        #10;
    end

    buf_w_strength dut(TF);
endmodule

module buf_w_strength (IO);
    inout IO;

    wire IO_wire;
    pullup (pull1, pull0) (IO_wire);    // This is the driver within the DUT

    // Assignment will make IO have a strength of 5
    assign (pull1, pull0) IO = IO_wire;
endmodule
```

To compile and elaborate the design in single-step *xrun* mode, and then bring up the `xcelium>` prompt, invoke the following command:

```
% xrun top.v -access rwc -tcl
```

For direct invocation mode, specify separate *xmvlog*, *xmelab*, and *xmsim* commands:

```
% xmvlog top.v
% xmelab -access rwc worklib.test_bench:module
% xmsim -tcl worklib.test_bench:module
```

Example 1

In this example, the `database` command opens an EVCD database called `test_default`. The `-into` option specifies that the associated filename is `default.evcd`.

The `probe` command creates an EVCD probe on the ports for the scope `test_bench.dut`. The `-database` option specifies that the database is `test_default`.

```
xcelium> database -open test_default -evcd -into default.evcd
Created EVCD database test_default
xcelium> probe -create test_bench.dut -evcd -database test_default
Created probe 1
xcelium> run
xcelium> exit
```

The simulator dumps data using the default mode. Final value changes are dumped in extended format for both vector and scalar ports. For vector ports, the simulator dumps value changes for the entire vector, not for individual bits of the vector.

```
% cat default.evcd
```

```
$date
    Jan 30, 2024  14:35:59
$end
$version
    TOOL:      xmsim    24.01-a071
$end
$timescale
    1 ns
$end

$scope module test_bench $end

$scope module dut $end
$var port      1 !      IO  $end
$upscope $end

$upscope $end

$enddefinitions $end

#0
$dumpports
pH 0 5 !
$end
#10
pD 6 0 !
#20
pH 0 5 !
#30
```

Example 2

In this example, the `database` command is used to open an EVCD database called `test_splitio`. The `-into` option specifies that the associated filename is `splitio.evcd`.

The `probe` command creates an EVCD probe on the ports for the scope `test_bench.dut`.

The `splitio` argument is included as an argument to the `-evcd` option. This argument causes the simulator to dump the value of the input port if any driver into the module instance changes value, and to dump the value of the output port if any driver within the module instance changes value. For inout ports, the simulator dumps two values: one corresponding to value changes of drivers into the module instance, and the other corresponding to value changes of drivers within the module instance.

The `-database` option specifies that the database is `test_splitio`.

```
xcelium> database -open test_splitio -evcd -into splitio.evcd
Created EVCD database test_splitio
xcelium> probe -create test_bench.dut -evcd splitio -database test_splitio
Created probe 1
xcelium> run
xcelium> exit
% cat splitio.evcd
```

```
$date
    Jan 30, 2024    14:37:21
$end
$version
    TOOL:          xmsim    24.01-a071
$end
$timescale
    1 ns
$end

$scope module test_bench $end

$scope module dut $end
$var port      1 !      IO $end
$upscope $end

$upscope $end

$enddefinitions $end

#0
$dumpports
pZ 0 0 !
pH 0 5 "
$end
#10
pD 6 0 !
#20
pD 3 0 !
#30
```

Example 3

In this example, the `probe` command creates a default EVCD database called `xcelium.evcd`, and then creates a probe on the ports for the scope `test_bench.dut`.

The `probe` command includes the `-evcd -evcdformat` option. The argument to `-evcdformat` is 1,

which reports the strengths for both the zero and one components of the value.

See the [Effect of -evcdformat Argument](#) table for a summary of the output using all `-evcdformat` arguments.

```
xcelium> probe -create test_bench.dut -evcd -evcdformat 1
Created default EVCD database xcelium.evcd
Created probe 1
xcelium> run
xcelium> exit
% cat xcelium.evcd
```

```
$date
    Jan 30, 2024    14:52:59
$end
$version
    TOOL:      xmsim    24.01-a071
$end
$timescale
    1 ns
$end

$scope module test_bench $end

$scope module dut $end
$var port      1 !      IO  $end
$upscope $end

$upscope $end

$enddefinitions $end

#0
$dumpports
pH 0 5 !
$end
#10
pD 6 5 !
#20
pH 3 5 !
#30
```

The following table shows the output for the example using the various `-evcdformat` arguments:

| 0 | 1 | 2 | 3 |
|-----------|-----------|-----------|-----------|
| #0 | #0 | #0 | #0 |
| pH 0 5 <0 | pH 0 5 <0 | pH 0 5 <0 | pH 0 5 <0 |
| #10 | #10 | #10 | #10 |
| pD 6 0 <0 | pD 6 5 <0 | pA 6 5 <0 | pA 6 5 <0 |
| #20 | #20 | #20 | #20 |
| pH 0 5 <0 | pH 3 5 <0 | pH 0 5 <0 | pH 3 5 <0 |

Generating an EVCD File using the \$dumpports System Task

The `$dumpports` system task scans the primary ports of a specified module instance and monitors the ports for both value and drive level. The task generates an output file that contains the value, direction, and strength information for all the ports of a device.

Syntax

```
$dumpports ([[scope_list], ["file_pathname"], [ID], [format_flag]]);
```

All arguments are optional. If you do not specify any arguments, both of the following are allowed:

```
$dumpports;  
$dumpports();
```

If an argument is null, you must use a comma before specifying the following argument. For example:

```
$dumpports( , "output.evcd");
```

The following table describes the arguments of `$dumpports`.

scope_list

One or more module identifiers. If more than one module is specified, separate the module identifiers with a comma. You cannot specify a top-level module.

This argument is optional if `$dumpports` is called from a module instance. If you do not specify a module, the scope is the module from which `$dumpports` has been called. However, if `$dumpports` is called from the top-level module, you must specify a *scope_list* argument because a top-level module is not a valid argument to `$dumpports`.

Paths to modules are allowed, using the period hierarchy separator.

"file_pathname"

A string containing the name of the output file.

This argument is optional. If you do not include this argument, the simulator creates a file called `verilog.evcd` in the current working directory.

ID

An integer data type that identifies a running `$dumpports` task with the `$dumpports_close` system task. For example,

```
module ...;
...
    integer evcd_id;
...
    initial
        begin
            $dumpports(dut1, "dut1.evcd", evcd_id);
            ...
            #4000 $dumpports_close(evcd_id);
        end
...

```

This argument is optional.

format_flag

An integer value that determines the value/strength format. The *format_flag* argument is optional. It can be one of the following values, or a sum of any of the following values.

- **0**: Default behavior.
- **1**: Keep losing value.
- **2**: Generate output according to the IEEE 1364-2001 standard.

The IEEE standard states that the values 0 (both input and output are active with value 0) and 1 (both input and output are active with value 1) are conflict states. The standard then defines two strength ranges:

- Strong: strengths 7, 6, and 5
- Weak: strengths 4, 3, 2, 1

The rules for resolving conflicts are:

- If the input and output are driving with the same range of strength, the resolved value is 0 or 1, and the strength is the stronger of the two.
- If the input is driving a strong strength and the output is driving a weak strength, the resolved value is d or u, and the strength is the strength of the input.
- If the input is driving a weak strength and the output is driving a strong strength, the resolved value is 1 or h, and the strength is the strength of the output.

The `format_flag` also changes the default behavior so that range information is included for non-scalar ports. If you do not specify this value, only size information is included. For example, suppose you have the following declaration:

```
input [31:1] my_wire;
```

In pre-2001 mode (without specifying the flag), the EVCD output is:

```
$var port 31 ! my_wire $end
```

In 2001 mode, the output is:


```
$var port [31:1] ! my_wire $end
```

- **4:** Compress the EVCD output file in `compress` format (.z file extension).
- **8:** Dump port direction information in the node information section of the output file. See [Node Information Section](#) for an example.

- **32:** Compress the EVCD output file in `gzip` format (`.gz` file extension).

There are two values that can be used to specify output file compression (*format_flag* value 4 and *format_flag* value 32). You cannot combine these two values.

The *format_flag* arguments can be combined. For example, if you want to generate an EVCD file that reports the strengths for both the zero and one components of the value (value 1), and that generates output according to the IEEE 1364-2001 standard (value 2), specify the *format_flag* argument as 3. To generate output according to the IEEE 1364-2001 standard (value 2) and compress the output file (value 4), specify a value of 6 for the *format_flag*.

 You can also specify the format by using the `-dumpports_format` option when you invoke the simulator.

Examples of Using \$dumpports System Task to Generate EVCD File

The following example generates an output file that contains the value, direction, and strength information for all the ports of `testbench.dut`. The output file is called `verilog.evcd`.

```
$dumpports(testbench.dut);
```

The following example generates an output file called `testoutput.evcd` in the current working directory.

```
$dumpports(testbench.dut, "testoutput.evcd");
```

The following example generates an output file that contains the value, direction, and strength information for all the ports of `testbench.dut1` and `testbench.dut2`. The output file is called `verilog.evcd`.

```
$dumpports(testbench.dut1, testbench.dut2);
```

The following example generates an output file called `testoutput.evcd` in the current working directory. This file contains output data for the two specified modules.

```
$dumpports(testbench.dut1, testbench.dut2, "testoutput.evcd");
```

The following example generates an output file called `testoutput.evcd` in the `./worklib` directory.

```
$dumpports(testbench.dut, "./worklib/testoutput.evcd");
```

The following example generates an EVCD file called `testoutput.evcd`. This `$dumpports` call includes the *format_flag* argument 1, which specifies that the calculated strengths for both the one and zero components of the value are to be reported.

```
$dumpports(testbench.dut, "testoutput.evcd", , 1);
```

The following example generates a compressed EVCD file called `testoutput.evcd.z` in the current working directory.

Compression of EVCD files is not supported on Windows platforms.

```
$dumpports(testbench.dut, "testoutput.evcd", , 4);
```

The following example generates an EVCD file called `testoutput.evcd`. This `$dumpports` call includes the *format_flag* argument 8, which specifies that information on the direction of ports is to be included in the node information section of the output file.

```
$dumpports(testbench.dut, "testoutput.evcd", , 8);
```

Related Topic

- [Using the `\$dumpports_close` System Task](#)

Generating an EVCD File for a VHDL Design

To generate an EVCD file, you must open an EVCD database and then probe the primary ports of a specified component instance, or specific primary ports, to an EVCD database. The following steps show you how to open a database with the `database` command and then probe the ports with the `probe` command. You can also probe objects to a database simply by using the `probe` command. In this case, the `probe` command opens a default EVCD database called `xcelium.evcd` and probes the ports for the specified scope.

Opening an EVCD Database

You can open an EVCD database with the `database` command. The syntax and options are the same as opening an EVCD database for Verilog. For a summary of the `database` command options and some Tcl command examples, see [Opening an EVCD Database](#) in the Verilog topic.

Probing VHDL Objects to an EVCD Database

You can probe VHDL objects, such as ports, within an EVCD database with the `probe` command.

Syntax

```
probe [-create] [{ object | scope_name }...]
      -evcd
      [[{splitio | simple}]
       [[-mode {lfcompat | lfvectcompat}}] |
       [-evcdformat format_number ]]] |
      [-database dbase_name ]

      [-all]
      [-depth { n | all | to_cells}]
      [-inputs]
      [-name probe_name]
      [-outputs]
      [-ports]
```

Example probe Command Lines

The following command probes all primary ports of the scope `:dut` to the default EVCD database opened with the database command.

```
xcelium> probe -create :dut -evcd
```

The following command probes all primary ports of the scope `:dut` and its subscores (that is, two levels of depth) to the EVCD database.

```
xcelium> probe :dut -evcd -depth 2
```

The following command probes all primary ports of the current scope to the EVCD database called `testoutput`.

```
xcelium> probe -evcd -all -database testoutput
```

You do not have to open a database with the `database` command before probing objects. You can probe objects to a database simply by using the `probe` command. In this case, the `probe` command opens a default EVCD database called `xcelium.evcd` and probes the specified objects. For example:

```
xcelium> probe -create -evcd -all
Created default EVCD database xcelium.evcd
```

Created probe 1

Examples of Using Multiple Commands to Generate a EVCD Database for VHDL

The following examples illustrate the usage of multiple Tcl commands to generate an EVCD database, and all share the VHDL `inout5.vhd` source below.

inout5.vhd

```
Library IEEE;
use IEEE.Std_Logic_1164.all, IEEE.Std_Logic_Arith.all;

entity one is
    port (a, b, c : std_logic;
          e : INOUT std_logic_vector(1 downto 0));
end;

architecture arch of one is
begin
    e <= "ZZ" after 1 ns;
end;

Library IEEE;
use IEEE.Std_Logic_1164.all, IEEE.Std_Logic_Arith.all;
use Std.Textio.All;
use IEEE.Std_Logic_TEXTIO.all;

entity two is
    port (a, b, c : std_logic);
end;

architecture arch of two is
    component one
        port (a, b, c : std_logic;
              e : INOUT std_logic_vector(1 downto 0));
    end component;

    for all : one use entity work.one(arch);
    signal ia, ib, ic : std_logic;
    signal ie : std_logic_vector(3 downto 2);

begin
    i1: one port map (ia, ib, ic, ie);
```

```
ie <= "00" after 1 ns,
"LL" after 2 ns,
"UU" after 3 ns,
"XX" after 4 ns,
"--" after 5 ns,
"WW" after 6 ns,
"ZZ" after 7 ns,
"HH" after 8 ns,
"11" after 9 ns,
"00" after 10 ns;
process(ie)
    variable dline : line;
begin
    write(dline, NOW, Left, 15);
    write(dline, ie);
    writeline(output, dline);
end process;
end;

Library IEEE;
use IEEE.Std_Logic_1164.all, IEEE.Std_Logic_Arith.all;

entity top is
end;

architecture arch of top is
    component two
        port (a, b, c : std_logic);
    end component;
    for all : two use entity work.two(arch);
    signal ia, ib, ic : std_logic;
begin
    ul: two port map (ia, ib, ic);
end;
```

To compile and elaborate the design in single-step *xrun* mode, and then bring up the `xcelium>` prompt, invoke the following command:

```
% xrun inout5.vhd -access rwc -tcl
```

For direct invocation mode, specify separate *xmvhdl*, *xmelab*, and *xmsim* commands:

```
% xmvhdl inout5.vhd
% xmelab -access rwc WORK.TOP:ARCH
% xmsim -tcl WORK.TOP:ARCH
```


Example 1

In this example, the `database` command opens an EVCD database called `test_default`. The `-into` option specifies that the associated filename is `default.evcd`.

The `probe` command creates an EVCD probe on the port `u1.i1:e`. The `-database` option specifies that the database is `test_default`.

```
xcelium> database -open test_default -evcd -into default.evcd
Created EVCD database test_default
xcelium> probe -create u1.i1:e -evcd -database test_default
Created probe 1
xcelium> run
xcelium> exit
```

In this example, the simulator dumps data using the default mode. For vector port `e`, the simulator dumps value changes for the whole vector.

```
% cat default.evcd
```

```
$date
    Jan 25, 2024  11:46:57
$end
$version
    TOOL:      xmsim    24.01-a071
$end
$timescale
    1 fs
$end

$scope module top $end

$scope module u1 $end

$scope module i1 $end
$var port      [1:0] !  e    $end
$upscope $end

$upscope $end

$upscope $end

$enddefinitions $end

#0
$dumpports
```

```
p?? 66 66 !
$end
#10000000
pDD 66 00 !
#20000000
pDD 55 00 !
#30000000
pNN 66 66 !
#60000000
pNN 55 55 !
#70000000
pff 00 00 !
#80000000
pUU 00 55 !
#90000000
pUU 00 66 !
#100000000
pDD 66 00 !
#100000000
```

Example 2

In this example, a `-evcd -mode lfcompat` expression is included with the `probe` command. The `-mode lfcompat` modifier allows the simulator to dump EVCD data as it was dumped by legacy releases. Vectors are dumped as individual bits, and legacy simulator strength mappings are used.

You must also include the `-evcd -mode lfcompat` expression if you want to dump a subelement of a compressed VHDL signal.

```
xcelium> database -open test_lfcompat -evcd -into lfcompat.evcd
Created EVCD database test_lfcompat
xcelium> probe -create ul:i1:e -evcd -mode lfcompat -database test_lfcompat
Created probe 1
xcelium> run
xcelium> exit
% cat lfcompat.evcd
```

```
$date
    Jan 25, 2024  15:24:42
$end
$version
    TOOL:      xmsim    24.01-a071
$end
$timescale
    1 fs
^ . . .
```

```
$end

$scope module top $end

$scope module u1 $end

$scope module i1 $end
$var port      1 <0    e [1] $end
$var port      1 <1    e [0] $end
$upscope $end

$upscope $end

$upscope $end

$senddefinitions $end

#0
$dumpports
p? 7 7 <0
p? 7 7 <1
$end
#1000000
pD 7 0 <0
pD 7 0 <1
#2000000
pD 5 0 <0
pD 5 0 <1
#3000000
pN 7 7 <0
pN 7 7 <1
#6000000
pN 3 3 <0
pN 3 3 <1
#7000000
pf 0 0 <0
pf 0 0 <1
#8000000
pU 0 5 <0
pU 0 5 <1
#9000000
pU 0 7 <0
pU 0 7 <1
#10000000
pD 7 0 <0
pD 7 0 <1
```

```
*  
#100000000
```

Example 3

In this example, the `probe -evcd` command includes the `splitio` argument. This causes the simulator to dump the value of the input port if any driver into the design entity changes value, and to dump the value of the output port if any driver within the design entity changes value. For inout ports, the simulator dumps two values: one corresponding to value changes of drivers into the design entity, and the other corresponding to value changes of drivers within the design entity.

```
xcelium> database -open test_splitio -evcd -into splitio.evcd  
Created EVCD database test_splitio  
xcelium> probe -create u1:i1:e -evcd splitio -database test_splitio  
Created probe 1  
xcelium> run  
xcelium> exit  
% cat splitio.evcd
```

```
$date  
    Jan 25, 2024  15:32:35  
$end  
$version  
    TOOL:      xmsim   24.01-a071  
$end  
$timescale  
    1 fs  
$end  
  
$scope module top $end  
  
$scope module u1 $end  
  
$scope module i1 $end  
$var port      [1:0] !      e  $end  
$var port      [1:0] "      e_O $end  
$upscope $end  
  
$upscope $end  
  
$upscope $end  
  
$enddefinitions $end  
  
#0  
$dumpports
```

```
pNN 66 66 !
pXX 66 66 "
$end
#10000000
pDD 66 00 !
pTT 00 00 "
#20000000
pDD 55 00 !
#30000000
pNN 66 66 !
#60000000
pNN 55 55 !
#70000000
pZZ 00 00 !
#80000000
pUU 00 55 !
#90000000
pUU 00 66 !
#100000000
pDD 66 00 !
#100000000
```

Example 4

In this example, the `probe` command creates a default EVCD database called `xcelium.evcd`, and then creates a probe on the port `:ul:il:e`.

The EVCD file shown in this example was generated using the `probe -evcd -evcdformat` expression. The argument to `-evcdformat` is `1`, which reports the strengths for both the zero and one components of the value.

```
xcelium> probe -create :ul:il:e -evcd -evcdformat 1
Created default EVCD database xcelium.evcd
Created probe 1
xcelium> run
xcelium> exit
% cat xcelium.evcd
```

```
$date
  Jan 25, 2024  15:36:32
$end
$version
  TOOL:      xmsim   24.01-a071
$end
$timescale
```

```
1 fs
$end

$scope module top $end

$scope module u1 $end

$scope module i1 $end
$var port      [1:0] !      e  $end
$upscope $end

$upscope $end

$upscope $end

$enddefinitions $end

#0
$dumpports
p?? 66 66 !
$end
#1000000
pDD 66 00 !
#2000000
pDD 55 00 !
#3000000
pNN 66 66 !
#6000000
pNN 55 55 !
#7000000
pff 00 00 !
#8000000
pUU 00 55 !
#9000000
pUU 00 66 !
#10000000
pDD 66 00 !
#10000000
```

The \$dumpports System Task

The Xcelium implementation of the `$dumpports` system task produces the same value change information as an equivalent TCL EVCD probe. Related *SystemVerilog 1800-2012 LRM* system tasks are also included. For legacy releases, to enable the new system task behavior, you were required to specify the `-use_new_dumpports` switch on the `xrun` or `xmsim` command line. However, as of the 15.1 release of the simulator, the enhanced `$dumpports` implementation is the default, and the `-use_new_dumpports` switch is not required (see the [\\$dumpports System Task Example](#)).

Important

The `-use_old_dumpports` option that enabled non-EVCD compliant `$dumpports` behavior has been removed and is no longer supported.

The following table describes system tasks related to the `$dumpport`, also described in the *SystemVerilog 1800-2012 LRM*.

| System Task | Description | Syntax | Example |
|-----------------------------|---|--|---|
| <code>\$dumpportsall</code> | Used to create a <i>checkpoint</i> value change dump at any point in simulation time. The task dumps values for all monitored ports regardless of recent changes. | <code>\$dumpportsall (filename¹)</code> | <code>\$dumpportsall("all.evcd");</code> |
| <code>\$dumpportson</code> | Used to re-enable value change dumping after it has been disabled. | <code>\$dumpportson (filename¹)</code> | <code>\$dumpportson("onoff.evcd");</code> |

| | | | |
|-------------------------------|--|--|---|
| <code>\$dumpportsoff</code> | Used to disable dumping. | <code>\$dumpportsoff</code> (<i>filename</i> ¹) | <code>\$dumpportsoff("onoff.evcd");</code> |
| <code>\$dumpportslimit</code> | Used to limit the size of the EVCD file. | <code>\$dumpportslimit</code> (<i>dumplimit</i> , <i>filename</i>) where, <ul style="list-style-type: none"> ◦ <i>dumplimit</i>: Specifies the maximum EVCD file size in bytes. The parameter must be the first argument to the <code>dumpportslimit</code> command. ◦ <i>filename</i>: Expression that is a string literal, string data type, or an integral data type containing a character string that specifies the name of the EVCD file. If <i>filename</i> is not specified, the task is executed on all open EVCD files. | <code>\$dumpportslimit(200, "limit.evcd");</code> |

1. *filename* is an expression that is a literal string, string data type, or an integral data type containing a character string that specifies the name of the EVCD file. If *filename* is not

specified, the task is executed on all open EVCD files.

The following example uses the `$dumpports` system task to produce the same value change information as an equivalent TCL EVCD probe, and illustrates the output of the `$dumpportsall`, `$dumpportson`, `$dumpportsoff`, and `$dumpportslimit` system tasks.

\$dumpports System Task Example

```
// filename dumpports.v
module test;
wire IO;
wire IO1;
reg q = 0;
int id;
assign IO = q;
  initial begin
    $dumpports(b1, "all.evcd");
    $dumpports(b1, "onoff.evcd");
    $dumpports(b1, "limit.evcd");
    $dumpportslimit(256, "limit.evcd");
    #5 q = 1;
    #2 $dumpportsoff("onoff.evcd");
    $dumpportsall("all.evcd");
    #3 q = 0;
    # 2 $dumpportson("onoff.evcd");
    #3 q = 1;
  end
buf1 b1 (IO1, IO);
endmodule

module buf1(IO1, IO2);
inout IO1, IO2;
  buf(IO1, IO2);
endmodule
```

Command Line

```
xrun -sv dumpports.v
```

Output

The following figure shows the output for `all.evcd` is shown on the left and the output for `onoff.evcd` is shown on the right.

Example Output for all.evcd (Left) and onoff.evcd (Right)

Output for all.evcd

```
$date
    <DATE>
$end
$version
    <TOOL>
$end
$timescale
    1ns
$end

$scope module test.bl $end
$var port      1 !      IO1 $end
$var port      1 "      IO2 $end
$upscope $end

$enddefinitions $end

#0
pL 6 0 !
pD 6 0 "
#5
pH 0 6 !
pU 0 6 "
#7
$dumpportsall
pH 0 6 !
pU 0 6 "
$end
#10
pL 6 0 !
pD 6 0 "
#15
pH 0 6 !
pU 0 6 "

$comment
$dumpports closed at time: 15
$end
```

Output for onoff.evcd

```
$date
    <DATE>
$end
$version
    <TOOL>
$end
$timescale
    1ns
$end

$scope module test.bl $end
$var port      1 !      IO1 $end
$var port      1 "      IO2 $end
$upscope $end

$enddefinitions $end

#0
pL 6 0 !
pD 6 0 "
#5
pH 0 6 !
pU 0 6 "
#7
$dumpportsoff
pX 6 6 !
pX 6 6 "
$end
#12
$dumpportson
pL 6 0 !
pD 6 0 "
$end
#15
pH 0 6 !
pU 0 6 "

$comment
$dumpports closed at time: 15
$end
```

The output for `limit.evcd` is shown in the following figure.

Example Output For `limit.evcd`

Output for `limit.evcd`

```
$date
    <DATE>
$end
$version
    <TOOL>
$end
$timescale
    1ns
$end

$scope module test.bl $end
$var port      1 !      IO1  $end
$var port      1 "      IO2  $end
$upscope $end

$enddefinitions $end

#0
pL 6 0 !
pD 6 0 "
#5
pH 0 6 !
pU 0 6 "
$comment
    Dump limit exceeded
$end
```

The `$dumpports` Compatibility

The value changes in the EVCD output of the enhanced `$dumpports` implementation are consistent with the output of an equivalent TCL EVCD probe, and existing `$dumpports` output conventions are otherwise maintained. This means, that while the actual value change content of the new EVCD output might change, the EVCD header and comment output remain the same. The only other differences in the output are in the usage of white space, and in the format of the date and tool version information in the header, which do not violate the EVCD format specification of the LRM.

While the original non-EVCD compliant `$dumpports` task will dump the same scope multiple times if the given scope list has duplicates, the new implementation eliminates redundant output without any meaningful loss of information.

Listing Ports and Port Vector Ranges using `$dumpports`

The enhanced `$dumpports` implementation lists ports and port vector ranges based on the actual declarations (`wire` or `reg`, for example) associated with the port, and not by any connections established in the module port list. This is also the behavior of TCL probes. However, this does differ from the old `$dumpports`, which lists named connections and part selects instead of the full port, even if the resulting connection is nameless.

Altering Signal Identifiers

To alter the signal identifiers in the `$dumpports` output (both new and old-style versions of `$dumpports`), you can use `-use_ieee_dumpport_ids` on the `xrun` or `xmsim` command line.

The legacy `+use_old_dumpport_ids` switch has been deprecated and is not supported with Xcelium.

Using the `format_flag` Argument to Control `$dumpports` Output

You can control the output of `$dumpports` with the `format_flag` argument. This argument can be one of the following values:

| Value | Description |
|-------|---|
| 0 | Default behavior. Report the strengths for both the zero and one components of the value if the strengths are the same. If the strengths are different, report only the "winning" strength. That is, the two strength values either match (for example, <code>pA 5 5 !</code>) or the winning strength is shown and the other is zero (for example, <code>pH 0 5 !</code>) |
| 1 | Keep losing value. Report the strengths for both the zero and one components of the value (for example, <code>pD 6 5 !</code>). |

| | |
|---|--|
| 2 | <p>Generate output according to the IEEE 1364-2001 standard.</p> <p>The IEEE standard states that the values 0 (both input and output are active with value 0) and 1 (both input and output are active with value 1) are conflict states. The standard then defines two strength ranges:</p> <ul style="list-style-type: none"> • Strong: strengths 7, 6, and 5 • Weak: strengths 4, 3, 2, 1 <p>The rules for resolving conflicts are:</p> <ul style="list-style-type: none"> ◦ If the input and output are driving with the same range of strength, the resolved value is <code>0</code> or <code>1</code>, and the strength is the stronger of the two. ◦ If the input is driving a strong strength and the output is driving a weak strength, the resolved value is <code>d</code> or <code>u</code>, and the strength is the strength of the input. ◦ If the input is driving a weak strength and the output is driving a strong strength, the resolved value is <code>l</code> or <code>h</code>, and the strength is the strength of the output. <p>This format flag also changes the default behavior so that range information is included for non-scalar ports. If you do not specify this value, only size information is included. For example, suppose you have the following declaration:</p> <pre>input [31:1] my_wire;</pre> <p>In pre-2001 mode (that is, without specifying the flag), the EVCD output is:</p> <pre>\$var port 31 ! my_wire \$end</pre> <p>In 2001 mode, the output is:</p> <pre>\$var port [31:1] ! my_wire \$end</pre> |
| 4 | <p>Compress the EVCD output file.</p> <p>This option generates a compressed output file called <code>file_pathname.Z</code>.</p> <p>Compression of EVCD files is not supported on Windows platforms. On Windows, using <code>4</code> as the value for the <code>format_flag</code> argument generates a warning message.</p> |
| 8 | <p>Dump port direction information in the node information section of the EVCD file. See Node Information Section for an example.</p> |

Example

The source code for this example is the same as that used for the examples shown in the section that describes how to generate an EVCD file using Tcl commands (see [Examples of EVCD Database](#)). The only difference is that a `$dumpports` system task has been inserted into the initial block, as shown below:

```
`timescale 1 ns / 1 ns
module test_bench;
    reg AS_3, AS_2, AS_1;
    tri TF;
    ...
    ...

    initial
        begin
            $dumpports(test_bench.dut, "test_format0.evcd", , 0);
            assign AS_3 = 1'bz;
            assign AS_1 = 1'bz;
            ...
            ...
        end
    ...
endmodule
```

In this example, the `format_flag` argument to the `$dumpports` call was set to 0, 1, 2, and 3 to generate four EVCD files. The following table shows the value change section for the four output files. Notice that, if the `format_flag` argument is set to 1 or to 3, strengths for both the zero and one components of the value are reported.

| format_flag Argument | | | |
|----------------------|----------|----------|----------|
| 0 | 1 | 2 | 3 |
| #0 | #0 | #0 | #0 |
| pH 0 5 ! | pH 0 5 ! | pH 0 5 ! | pH 0 5 ! |
| #10 | #10 | #10 | #10 |
| pD 6 0 ! | pD 6 5 ! | pA 6 5 ! | pA 6 5 ! |
| #20 | #20 | #20 | #20 |
| pH 0 5 ! | pH 3 5 ! | pH 0 5 ! | pH 3 5 ! |

Using the \$dumpports_close System Task

The `$dumpports_close` system task stops a running `$dumpports` task.

Syntax

```
$dumpports_close (ID);
```

The `ID` argument is an integer data type that identifies a particular `$dumpports` system task. If only one `$dumpports` system task is running, the `ID` argument can be omitted.

In the following example, two EVCD files are generated: `dut1.evcd` and `dut2.evcd`. The dump identified with the ID called `id1` is closed after 4000 time units.

```
module top;
  reg A;
  integer id1;
  integer id2;
  ...
  ...
  initial
    begin
      $dumpports(dut1, "dut1.evcd", id1);
      $dumpports(dut2, "dut2.evcd", id2);
      #4000 $dumpports_close(id1);
    end
  ...
  ...
endmodule
```

\$dumpports Restrictions

The following restrictions apply to the `$dumpports` system task:

- The `$dumpports` system task does not work with save and restart.
- Continuous assignments cannot have delays.
- The following wire types are the only ones permitted to be connected to the ports:
`wire, tri, tri0, tri1, reg, trireg, supply0, supply1`
- Directional information can be lost when a port is driven by one or more drivers of the different `tran` elements (`tran`, `rtran`, `rtranif0`, ...).

In the following example, the direction of the port `out` cannot be determined because the value

that the tran gate is transporting from its other terminal is unknown.

```
bufif1 u1(out, 1'b1, 1'b1);
DUT udut(out);
...
module DUT(out)
    tran t1(out, int);
```

Format of the EVCD File

The format of the extended VCD file is similar to that of the four-state VCD file, which is described in the IEEE Verilog LRM. The file contains three sections:

- Header information
- Node information
- Value changes

Header Information Section

The EVCD file begins with a header in the following format:

```
$date
    <date and time file was generated>
$end
$version
    <version of xmsim>
$end
$timescale
    <timescale used for the simulation>
$end
```

The following is an example header section from an EVCD file that was generated using Tcl commands:


```
$date
    Dec 11, 2015   15:42:26
$end
$version
    TOOL:          xmsim    16.11-p001
$end
$timescale
    1 ns
$end
```

Node Information Section

This section of the EVCD file begins with the `$scope` keyword command, which defines the scope of the primary ports being dumped, and ends with `$enddefinitions $end`, which marks the end of the header and node information sections.

The Node section has the following format:

```
$scope module <module_instance_name> $end
$var <var_type>    <port_size>    <port_identifier_code>    <port_reference> $end
...
...
$upscope $end
$enddefinitions $end
```

The constructs that begin with the `$var` keyword command are defined as follows:

- *var_type*: The type of variable. For EVCD, this is always the keyword `port`, unless you have specified that you want port direction information in the output file. You can do this by using the `-evcd direction` option when you open the database with the Tcl database command, or by setting the *format_flag* argument to 8 if you are using the `$dumpports` task. See the examples below.
- *port_size*: The port size is a decimal number indicating the number of bits in the port.
- *port_identifier_code*: The identifier for the port. This identifier is used in subsequent message dumping.

The port identifier codes in an EVCD file differ from what is specified in the IEEE standard. The IEEE standard specifies that the identifier code is to be an integer preceded by <, which starts at zero and ascends in one unit increments for each port. For example <0, <1, and so on. In an EVCD file generated by Xcelium, space-efficient identifiers are used in order to minimize the size of the file. See the example below.

Use the `xmsim -use_ieee_dumpport_ids` option if you want the identifier codes to conform to the IEEE standard.

- *port_reference*: An identifier that indicates the port name.

Examples of the Node Section

The following is an example of the node information section in an EVCD file:

```
$scope module board.counter $end
$var port      4 !          value $end
$var port      1 "          clock $end
$var port      1 #          fifteen $end
$var port      1 $          altFifteen $end
$upscope $end

$senddefinitions $end
```

The following example shows the node information section if the *format_flag* has been set to 8, as in the following `$dumpports` task:

```
$dumpports(board.counter, "test.evcd", , 8);
```

Notice that port direction information is included in the output.

```
$scope module board.counter $end
$var output    4 !          value $end
$var input     1 "          clock $end
$var output    1 #          fifteen $end
$var output    1 $          altFifteen $end
$upscope $end

$senddefinitions $end
```

In legacy simulator releases, vectors were dumped as individual bits. You can use the `-evcd -mode`

`lfcompat` option when you create the probe to dump individual bits for vectors. This option also preserves the strength mappings of legacy releases. For example:

```
xcelium> probe -create :dut -evcd -mode lfcompat -database xcelium.evcd
```

The following is the node information section that is dumped when the `-mode lfcompat` option is used for the same example:

```
$scope module top $end

$scope module u1 $end

$scope module i1 $end
$var port      1 <0    e [1] $end
$var port      1 <1    e [0] $end
$upscope $end

$upscope $end

$upscope $end

$enddefinitions $end
```

Value Change Section

The value change section shows the actual value changes at each simulation time increment. Only variables that change value during a time increment are listed. The format of the message is as follows:

```
#<simulation_time>
p<port_value> <0_strength_component> <1_strength_component> <identifier_code>
```

The constructs in the message are defined as follows:

- `#simulation_time`: The simulation time.
- `p`: Key character that indicates a port.
- `port_value`: State character that indicates the driving direction and state. The state characters are described in [Port Value Character Mapping](#).
- `0_strength_component`: One of the eight Verilog strengths. This indicates the `strength0` component of the value. See [Strength Mapping](#) for information on strength mapping.
- `1_strength_component`: One of the eight Verilog strengths. This indicates the `strength1`

component of the value.

- *identifier_code*: The identifier code for the port, which was defined in the `$var` construct for the port.

The following example shows the node information section and part of the value change section of an EVCD file:

```
$scope module board $end

$scope module counter $end
$var port      [3:0] !    value  $end
$var port      1 "    clock  $end
$var port      1 #    fifteen $end
$var port      1 $    altFifteen $end
$upscope $end

$upscope $end

$enddefinitions $end

#0
$dumpports
pXXXX 6666 6666 !
pN 6 6 "
pX 6 6 #
pX 6 6 $
$end
#5
pU 0 6 "
#10
pLLLL 6666 0000 !
pL 6 0 #
pL 6 0 $
#50
pD 6 0 "
...
...
```

Port Names

Port names are recorded in the output file as follows:

- A port that is explicitly named is recorded in the output file.

- If a port is not explicitly named, the name of the object used in the port definition is recorded.
- If the name of a port cannot be determined from the object used in the port definition, the port index number is used (the first port being 0).

Drivers

A driver is anything that can drive a value onto a net, including the following:

- Primitives
- Continuous assigns
- Forces
- Ports with objects of type other than net, such as the following:

```
module foo(out, ...)
    output out;
    reg out;
```

If a net is forced, a comment is placed into the output file stating that the net connected to the port is being forced, and giving the scope of the force. Forces are treated differently because the existence of a force is not permanent, even though a force is a driver.

While a force is active, driver collisions are ignored and the level part of the output is determined by the scope of the force definition. If the EVCD was generated using Tcl, the port driving direction is treated as direction input. When the force is released, a note is again placed into the output file.

Port Value Character Mapping

The state information shown in this section is described in terms of input values from a test fixture, the output values of the device under test, and the states that represent unknown direction.

Direction INPUT

Given a device under test (DUT) and a test fixture, the driving direction is INPUT if the drivers from the test fixture are driving some non-tristated value and the drivers inside the DUT are tristated. The resolved value is mapped as shown in the following table. In the table, the term *active* implies that the drivers are in a non-tristated condition.

| | |
|---|---------|
| D | (0) low |
|---|---------|

| | |
|---|--|
| d | (0) low (2 or more drivers active) |
| U | (1) high |
| u | (1) high (2 or more drivers active) |
| N | (X) unknown |
| n | (X) unknown because of a 1-0 collision |
| Z | (Z) tristate |

Direction OUTPUT

If the driving value from drivers inside the DUT is non-tristated, but the value driven by the drivers in the test fixture is tristated, the direction is OUTPUT. The resolved value is mapped as shown in the following table.

| | |
|---|---------------------------------------|
| L | (0) low |
| I | (0) low (more than 2 drivers active) |
| H | (1) high |
| h | (1) high (more than 2 drivers active) |
| X | (X) unknown (don't care) |
| T | (Z) tristate |

Direction UNKNOWN

If both the drivers in the test fixture and drivers inside the DUT are driving some non-tristated value, the direction is UNKNOWN. The resolved value is mapped shown in the following table.

| | |
|---|--|
| 0 | (0) low (both input and output are active with 0 value) |
| 1 | (1) high (both input and output are active with 1 value) |
| ? | (X) unknown (input X and output X) |
| F | tri-state (input and output unconnected) |

| | |
|---|--|
| A | (0-1) unknown (input 0 and output 1) |
| a | (0-X) unknown (input 0 and output X) |
| B | (1-0) unknown (input 1 and output 0) |
| b | (1-X) unknown (input 1 and output X) |
| C | (X-0) unknown (input X and output 0) |
| c | (X-1) unknown (input X and output 1) |
| f | (Z) unknown (input and output tristated) |

The level of a driver is determined by the scope of the driver's placement on a net. Port type has no influence on the level of a signal. Any driver whose definition is outside of the scope of the DUT is at the test fixture level.

In the following example, because the driver is outside the scope of a DUT, the continuous assignment is at the test fixture level:

```
module top;
  reg regA;
  assign dut1.out = regA;
  dut dut1(out, ....);
  initial
    $dumpports(dut1, "testVec.file");
  ...
endmodule

module dut(out, ...
  output out;
  wire out;
  ...
endmodule
```

Strength Mapping

For Verilog, strength values in the EVCD output are shown in the following table. Append 0 or 1 to the keyword as appropriate for the strength component.

| | |
|---|-------------|
| 0 | highz (HiZ) |
| 1 | small (Sm) |
| 2 | medium (Me) |
| 3 | weak (We) |
| 4 | large (La) |
| 5 | pull (Pu) |
| 6 | strong (St) |
| 7 | supply (Su) |

For VHDL, the strength mapping is shown in the following table.

| std_ulogic character | 0's strength | 1's strength |
|----------------------|--------------|--------------|
| 0 | 6 | 0 |
| 1 | 0 | 6 |
| X | 6 | 6 |
| Z | 0 | 0 |
| U | 6 | 6 |
| W | 5 | 5 |
| - | 6 | 6 |
| L | 5 | 0 |
| H | 0 | 5 |

The strength mapping for legacy releases is shown in the Table: [Strength Mapping in Legacy Releases](#).

To dump an EVCD file with legacy strength mappings, use the `-mode lfcompat` or `-mode lfvectcompat` option when you create the probe. For example:

```
xcelium> probe -create :dut -evcd -mode lfcompat -database test_output
```

Both `lfcompat` and `lfvectcompat` preserve the older strength mappings.

The `-mode lfcompat` option dumps individual bits for vector ports, as was done by default in legacy simulator releases.

The `-mode lfvectcompat` option dumps whole vectors. This argument replaces the `database -evcd vector` option, which is no longer supported.

Strength Mapping in Legacy Releases

| std_ulogic character | 0's strength | 1's strength |
|----------------------|--------------|--------------|
| 0 | 7 | 0 |
| 1 | 0 | 7 |
| X | 7 | 7 |
| Z | 0 | 0 |
| U | 7 | 7 |
| W | 3 | 3 |
| - | 7 | 7 |
| L | 5 | 0 |
| H | 0 | 5 |

Appendix A: Extensions of Tcl

The *xmsim* simulator uses the Tool Command Language (Tcl) to control the execution of a simulation. Cadence has extended the functionality of the Tcl command interpreter so that it understands a number of new commands and some new syntax.

The current release uses Tcl 8.6.8. For a complete description of Tcl, refer to *Tcl and the Tk Toolkit* by John K. Ousterhout (Addison-Wesley, Reading, MA, 1994).

Related Topics

- [Extensions to Tcl](#)
- <https://www.tcl.tk/>
- <https://www.activestate.com/products/tcl/>

Extensions to Tcl

Cadence has extended the functionality of the Tcl command interpreter so that it understands a number of new commands and some new syntax. And, also how to handle types and operators of the Verilog and VHDL hardware description languages.

- [Value Substitution](#)
- [The @ Character and Escaped Names](#)
- [Expression Evaluation](#)
- [VHDL Expressions](#)

Value Substitution

Using the value of an object in a command is very common:

```
xcelium> set a [value w]
```

The Tcl interpreter includes a shorthand notation, using the pound sign (#), for this:

```
xcelium> set a #w
```

This notation is completely analogous to the `$` variable substitution that is standard in Tcl (see [Variable Substitution](#)), although a format specifier capability has been added.

To control the format of the value substitution, a format character preceded by a percent sign can follow the `#`. For example:

```
xcelium> value w
8'b11111111
xcelium> set a %#xw
8'hff
```

Because all format specifiers are one character long, no special syntax is needed to separate the format specifier from the HDL name.

The format used when no format specifier is given will be the most verbose and specific format for the type of the object. In some cases, this makes value substitution preferable to command substitution with the `value` command. For example, the first command shown below assigns `12'hXaZ` to the variable `a`, while the second command assigns `12'b000x10101z01` to the variable.

```
xcelium> set a [value w]
12'hXaZ
xcelium> set a #w
12'b000x10101z01
```

If the substituted value is to be used as part of a command-line argument, braces can be used to delineate the extent of the HDL name in the same way that they are used by Tcl (and the c-shell) to delineate variable names. For example, the following command sets `x` to the value of `w` with `10` appended:

```
xcelium> set x %#x{w}10
8'ff10
```

`%x{name}` is syntactically equivalent to `[value %x name]`.

The following `deposit` command sets the value of `x` to the current value of `w`.

```
xcelium> deposit x = #w
```

The @ Character and Escaped Names

To specify HDL escaped names on the *xmsim* Tcl command line, a special syntax using the `@` character has been added.

The backslash (`\`) character used by both Verilog and VHDL make escaped names difficult to specify because the backslash is a special character in the Tcl parser which suppresses the special meaning of the character that follows it. See [Backslash Substitution](#).

For example, consider a VHDL escaped name such as `\ARR"32"\`. To specify this name on the command line, the backslashes and the quotes must be escaped with backslashes, as follows:

```
xcelium> value \\ARR\"32\"\\
```

Also, consider a Verilog escaped name such as `\\ARR\"32\"`. Note that Verilog escaped names are terminated by a space character. This character must be explicitly specified on the Tcl command line:

```
xcelium> value \\ARR\"32\"\\      # a space character follows the final backslash
```

To simplify the specification of HDL escaped names on the command line, a syntax has been added that lets you use the "at" sign (`@`) followed by the escaped name enclosed in curly braces. The meaning of special characters inside the braces is suppressed as if they were each preceded by a backslash. For example:

```
xcelium> value @{\\ARR\"32\"}\\    # VHDL escaped name
xcelium> value @{\\ARR\"32\" }    # Verilog escaped name
```

This syntax matches the opening curly brace with the next closing brace on the command line, collecting all characters between the braces, and ignoring their special meaning. If the escaped name includes a closing curly brace, an error will be generated. For example, if you have a VHDL escaped name such as `\\ARR{32}\\`, the following syntax will result in an error because the closing brace inside the name terminates the `@` sign syntax.

```
xcelium> value @{\\ARR{32}\\}
```

Though it is clear in this example that the inner closing brace matches an inner open brace, a closing brace could appear without a corresponding open brace. To be able to parse such a name, the parser needs to use the cues given by the HDL's escaped name syntax to figure out when a curly brace is inside an escaped name and when it is the terminating curly brace. To do this, the parser has to know what language's escaped name syntax to expect. This can be specified with an extra character following the `@` sign. A backslash indicates VHDL syntax, and an underscore character indicates Verilog syntax.

```
xcelium> value @\\{\\ARR{32}\\}
xcelium> value @_\\{\\ARR{32} }
```

The `@` sign syntax can surround more than one pathname element, as shown in the following example:

```
xcelium> value @_{top.u1.\\ARR{32} }.r}
```

The syntax can also surround one element in a pathname:

```
xcelium> value top.u1. @_\\{\\ARR{32} }.r
```

Example with Parameterized Class Names

In the Tcl interface, a backslash must be placed before the # in a parameterized class name. For example:

```
module test;
  class c #(string s = "hi");
    static reg r;
  endclass
  initial $display(c#("hi")::r);
endmodule
...
```

```
xcelium> value c\#("hi")::r
```

To avoid having to place a backslash before the #, use the `@{name}` syntax when you are trying to access a parameterized class in Tcl. For example:

```
xcelium> value @{c#("hi")::r}
```

or:

```
xcelium> stop -condition {#@{c#("hi")::r} == 1}
```

Expression Evaluation

Tcl has a built-in `expr` command that parses and evaluates numeric expressions. This command handles the standard arithmetic and logical operators on operands that are of integer, real, and string types. This facility has been enhanced for both Verilog and for VHDL to handle types and operators of these languages.

Verilog Expressions

Verilog has one main data type (a four-state logic vector) and the operators on this type are clearly defined in the language. This type and its operators have been incorporated into the expression evaluator so that just about any expression appearing in Verilog code can be mimicked in Tcl.

The new data type added to the Tcl parser to support the evaluation of Verilog expressions is Verilog Register. The Verilog Register type is an unsigned vector type of arbitrary bit size whose values are Verilog literals, such as ``b1001`, `16'h8ff0`, and `8'bxxxxxx01`.

Any expression that has an operand in this form is treated as a Verilog expression, and the expression evaluator follows the rules of Verilog literals and operators when evaluating it.

```
xcelium> expr `hlf + 42
32'b1001001
xcelium> value reg
8'b11111111
xcelium> expr #reg & ~32'hlf
32'b11100000
```

The format of the value returned from the `expr` command is controlled by the `vlog_format` variable.

Array indexes can be specified using square brackets (Verilog style) or parentheses (VHDL style). For example:

```
xcelium> value count[0]
1'h1
xcelium> value count(0)
1'h1
xcelium> value %b count[1:0]
2'b01
xcelium> value %b count(1:0)
2'b01
xcelium> value count['b0000]
1'h1
xcelium> value count('b0000)
1'h1
```

The square brackets specified in the examples do not invoke command substitution. The modified Tcl parser indicates when square brackets are meant to be array indexes.

Operators Added to Tcl

The Tcl expression parser already supports most of the logical and arithmetic operator symbols of Verilog. The following operators have been added:

Table A-1 Binary Operator Extensions

| Operator | Function |
|----------|---|
| === | Case equality See Equality Operators for more information on equality operators. |
| !== | Case inequality |
| ~^ | Bit-wise XNOR |
| ^~ | Another symbol for bit-wise XNOR |

Table A-2 Unary Operator Extensions

| Operator | Function |
|----------|-----------------------------------|
| & | Reduction AND |
| ~& | Reduction NAND |
| | Reduction OR |
| ~ | Reduction NOR |
| ^ | Reduction XOR |
| ~^ | Reduction XNOR |
| ^~ | Another symbol for reduction XNOR |

Concatenation Operators

Because native Tcl syntax uses braces for suppressing substitution, the Tcl interpreter is unable to support braces for use as the Verilog concatenation operator. Instead, you can use a function called `vcat`, which takes an arbitrary number of arguments and returns a Verilog Register value which is the concatenation of all the bits in all the arguments.

```
xcelium> expr vcat(3'b101, 5'b1, #r)
16'b1010000111111111
```

The arguments to the `vcat` function can be numeric literals or strings enclosed in double-quotes. Strings are converted to Verilog's value representation of ASCII characters before they are

concatenated into the return value. This allows the `vcat` function to double as a conversion function from strings to Verilog values, as shown in the following example:

```
xcelium> expr vcat("Hello")
40'b100100001100101011011000110110001101111
```

You can use a function called `vrep` to repeat concatenation. This function takes an integer repetition count and a Verilog value as arguments and returns a value whose bit pattern is that of the second argument repeated the number of times specified by the first argument:

```
xcelium> expr vrep(3, 3'b101)
9'b101101101
```

Logical AND and OR Operators

Tcl defines the logical AND and OR operators to be short-circuiting. That is, if the first operand is sufficient to determine the result of the operator, then the second operand is not evaluated. This conflicts with Verilog in that both operands are needed to determine whether the Verilog version of the operator should be performed, and to determine the bit size of the result.

Because of this, these operators will not behave exactly as they do in Verilog, but will always return an integer, either 1 or 0. If the result of the expression contains `x` or `z` bits, these are treated as zeroes.

```
xcelium> expr 2'b11 || 5'b0
1
xcelium> expr 'bx || 2'b1
1
xcelium> expr 2'b01 && 5'b10
1
xcelium> expr 'bx && 2'b1
0
```

Equality Operators

The single-equality operator (`=`) is used for assignment in Verilog. In Tcl, this operator is a logical comparison operator. In a Tcl expression, `=` is the same as the Verilog logical comparison operator (`==`). The following two expressions are the same:

```
{ #top.load = 1'b1 }
{ #top.load == 1'b1 }
```

These operators return the unknown value (`x`) if either operand is unknown. For example, the following expression returns an unknown result because one of the operands is unknown:


```
{ #top.load == 1'bx }
```

For the case equality operator (`==`) and the case inequality operator (`!=`), bits that are unknown are included in the comparison, and the result of the expression is always 1 (true) or 0 (false).

In a conditional expression, an unknown result is treated as false. For example, in the following `stop` command, the expression returns an unknown result, and is, therefore, false. This conditional breakpoint will not trigger when the signal `top.load` has the value `x`.

```
xcelium> stop -create -condition {#top.load == 1'bx}
```

To set a breakpoint that will stop when the value is `x`, use the case equality operator, as follows:

```
xcelium> stop -create -condition {#top.load === 1'bx}
```

Conversion Functions

The functions `vhex`, `voct`, `vdec` and `vbin` convert a Verilog value or an integer to a Verilog literal in the corresponding format. The result type of these functions is a string. These functions are provided simply for base conversion and output formatting.

```
xcelium> expr vhex(`b111100101)
32'h1e5
xcelium> expr voct(9'b111100101)
9'o745
xcelium> expr vdec(9'o745)
9'd485
xcelium> expr vbin(400 + 80 + 5)
32'b111100101
```

The function `vstr` takes a bit pattern argument and returns the equivalent string:

```
xcelium> expr vstr(16'b100100001101001)
Hi
```

VHDL Expressions

In VHDL, unlike in Verilog, there are many data types, both predefined and user-defined, and the user can redefine the operator functions for these types. Because it is not possible to build all possible VHDL expressions into Tcl, the most common data types and the predefined versions of the operators for these types that are contained in standard packages have been built into the Tcl interpreter.

The types `STD.INTEGER`, `STD.REAL`, and `STD.STRING` are handled in basic Tcl. The expression evaluator has been enhanced to handle enumeration values and vectors of enumeration values.

These are critical because they are used to represent logic vectors. The enhancements allow the expression evaluator to handle enumeration literals and vector literals and the predefined operators that VHDL provides for them.

Because logic vectors are the basis of most simulations, and because the predefined operators in VHDL for arrays of enumeration types are not sufficient to make them useful as logic types, a package that defines a logic type and its operators is generally used. Since this is most commonly `IEEE.STD_LOGIC_1164`, the operators that are defined in this package have also been built into the Tcl expression evaluator.

Enumeration Literals

In basic Tcl, the type of a literal can be determined directly. Integers and reals contain only digits, decimal points, and other well-defined characters. Strings are always enclosed in double-quotes. Verilog literals have a specific format that includes a tick (```) followed by a radix character. In VHDL, a word that is not followed by parentheses (as for a function call) is treated as a VHDL enumeration literal. A character that is enclosed in single quotes is also treated as an enumeration literal. Anything else is an invocation of a math function (such as `sin(1)`) or a syntax error.

To be able to use an enumeration literal, the expression evaluator must also be able to determine the type declaration to which the literal belongs. In order to tell the expression evaluator what type a literal is, you can include the name of the type with the literal. The format is `type:literal`, where `type` is a local unit path name to the enumeration type, and `literal` is the enumeration literal.

Use the `%e` format specifier on the `value` command to generate a fully qualified enumeration literal. Without a format specifier, an unqualified enumeration literal will result, as shown in the following example:

```
xcelium> value color
green
xcelium> value %e color
@mylib.color_pkg:colors:green
```

This syntax assumes that the object called `color` is of type `colors`, which is declared in a package called `color_pkg` in the design library `mylib`.

For types that are declared in architectures, the syntax is:

```
xcelium> value %e x
@mylib.testbench(behav):my_logic:logic_1
```

where `testbench` is the entity, `behav` is the architecture, and `my_logic` is the type.

You almost never have to type this format because:

- If a binary operation is being performed, the expression evaluator only needs to know the type

of one operand. It assumes that the other operand is of the same type.

- When the value of an HDL object is substituted using value substitution, the fully typed format is used. That is, the expression evaluator understands an expression such as `{#color = green}`, where `color` is an enumeration type object, and `green` is a literal of that type.

The operators for enumeration types are those that are implicitly defined in VHDL for all enumeration types. These are the equality and relational operators. Addition and subtraction of two enumeration literals, or one enumeration literal and one integer is also supported. For these operators, operands that are enumeration literals are converted to their position value, the operation is performed as for integers, the result is taken as a position value, and this result is then converted back to the corresponding enumeration literal. For example:

```
xcelium> expr green + 1
xmsim: *E,TCLERR: cannot determine enumeration type for "green" (operator: +).
xcelium> expr @mylib.color_pkg:colors:green + 1
@mylib.color_pkg:colors:blue
```

Enumeration Vectors

Enumeration vectors that can appear in VHDL as strings are handled in the same way as enumeration literals. Strings can be qualified with a type name, just as enumeration literals can. If the type of a string can be deduced from the context, it does not have to be qualified. For example:

```
xcelium> value my_bus
"11001100"
xcelium> value %e my_bus
@std.bit:"11001100"
xcelium> expr #my_bus = "\"11001100\""
@std.standard:boolean:true
```

In these examples, although `my_bus` is of type `bit_vector`, the qualifying type is the element type `bit`. When Tcl generates a qualified enumeration literal, it uses the name of the base enumeration type. If you enter a qualified literal, the qualifying type can be any subtype of an enumeration type or an array type whose element type is an enumeration. Either of these is equivalent to using the enumeration base type directly.

In the last command shown above, the double-quotes around the vector literal are required. The backslashes are necessary so that the quotes are not stripped by the Tcl parser. Alternatively, the expression can be enclosed in braces so that the backslashes are not necessary. For example:

```
xcelium> expr {#my_bus = "11001100"}
@std.standard:boolean:true
```

The result of the `expr` command is a fully qualified literal. If the result is a VHDL enumeration or

vector type, it includes the type qualification. In the example above, the result of the equality operator is of type `STD.STANDARD:BOOLEAN`. Because the operands are VHDL-specific types, the VHDL definition of the operator is used. The LRM defines this operator to return the VHDL boolean type.

Operators on enumeration vector types are those that are implicitly defined in VHDL for string types. These are the equality operators, relational operators, and the concatenation operator.

The concatenation operator symbol in VHDL is an ampersand (&). This is the same as the Verilog bitwise-and operator symbol. This symbol represents both operations. The expression evaluator looks at the types of its operands to determine which operation to perform.

The precedence of the & operator in VHDL is higher than that of the & operator in Verilog, but the expression evaluator cannot distinguish the two at the time it needs to know the operator precedence. Therefore, the VHDL & operator has the same precedence as the Verilog & operator. If you use this operator in a complex expression, use parentheses to enforce the desired precedence.

Standard Logic Type

Additional support is provided for the logic type defined in the IEEE 1164 standard packages. The base enumeration type that this package defines is called `STD_ULOGIC`. It is a nine-state logic type. The packages that define this type and its operators are `STD_LOGIC_1164` and `STD_LOGIC_ARITH`. There is a resolved subtype of `STD_ULOGIC` that is called `STD_LOGIC`, so you can use either name to qualify a literal of this type.

The operators that are defined for operands of type `STD_ULOGIC` are those that are defined in the `IEEE.STD_LOGIC_1164` and `IEEE.STD_LOGIC_ARITH` packages. These operators have definitions other than the standard VHDL definition for one-dimensional discrete arrays. These operations are described in the next section. Here are some examples:

```
xcelium> expr @ieee.std_logic_1164:std_logic:'X'
@ieee.std_logic_1164:std_ulogic:'X'
xcelium> value %e my_bus
@ieee.std_logic_1164:std_ulogic:"00001010"
xcelium> expr #my_bus + 10
@ieee.std_logic_1164:std_ulogic:"00010100"
xcelium> expr #my_bus + 0x1f
@ieee.std_logic_1164:std_ulogic:"00101001"
```

VHDL operators

The following table shows the operators that have been added to support VHDL. In the table, the term *any VHDL type* means "any enumeration literal or vector type". A specific type such as *bit* or *boolean* refers to a literal of this type or to a vector with this as its element type. In all cases, an operand that is an enumeration literal is treated the same as a vector of length 1.

For precise definitions of the operators, see the VHDL Language Reference Manual or the source code for the IEEE standard logic packages.

| Operation | Infix Symbol | Left Operand Type | Right Operand Type | Result Type |
|---|---|---|--|---|
| BITWISE AND BITWISE NAND BITWISE OR BITWISE NOR BITWISE XOR BITWISE XNOR | and nand or nor xor xnor | bit, boolean, or std_ulogic, any length | same type, same length | same type, same length |
| BITWISE NOT | not | N/A | bit, boolean, std_ulogic, any length | same type, same length |
| PLUS | + | any VHDL type, length 1 | same type, length 1 | same type, length 1 |
| | | any VHDL type, length 1 | integer | same VHDL type, length 1 |
| | | integer | any VHDL type, length 1 | same VHDL type, length 1 |
| PLUS MINUS | + - | std_ulogic, any length | std_ulogic, any length | std_ulogic, length of longer operand |
| | | std_ulogic, any length | integer | std_ulogic, length of left operand |
| | | integer | std_ulogic, any length | std_ulogic, length of right operand |

| | | | | |
|----------------------------|--------|---|---------------------------|--|
| LESS THAN LESS OR EQUAL | < | any VHDL type, any length | same type, same length | |
| | <= | std_ulogic, any length | integer | |
| | | integer | std_ulogic, any length | |
| LOGICAL SHIFT LEFT | sll | bit, boolean, or std_ulogic, any length | integer | same VHDL type, same length |
| LOGICAL SHIFT RIGHT | srl | | | |
| ROTATE LEFT | rol | | | |
| ROTATE RIGHT | ror | | | |
| ARITHMETIC SHIFT LEFT | sla | bit, boolean, any length | integer | same VHDL type, same length |
| ARITHMETIC SHIFT RIGHT | sra | | | |
| CONCATENATION | & | any VHDL type, any length | same type, any length | same type, sum of operand's lengths |
| EXPONENTIATION | ** | integer or real | integer | integer or real |
| ABSOLUTE VALUE | abs | N/A | integer or real | same type |
| MODULUS | mod, % | integer | integer | integer |
| REMAINDER | rem | integer | integer | integer |

The bitwise logical operators and the logical shift operators that are shown in this table also exist in Verilog, but with different infix symbols. The symbols are not interchangeable. To get the VHDL version of the operator, you must use the VHDL infix symbol.

For the equality and modulus operators, you can use the Verilog symbol and the VHDL symbol interchangeably. The VHDL definition of these operators will be used if at least one of the operands is a VHDL enumeration type.

Arithmetic operations in which at least one of the operands is `STD_ULOGIC` are defined in the

standard logic package to be integer operations. In these cases, the `STD_ULOGIC` operands are first converted to integers and the result is converted back to `STD_ULOGIC`. The rules for this conversion are found in the standard logic package's `to_integer` function.

Relational operations in which one operand is `STD_ULOGIC` and the other is an integer are also integer operations. The `STD_ULOGIC` operand is first converted to an integer, and an integer comparison is performed. Relational operations on two `STD_ULOGIC` vector values use the standard definitions for one-dimensional VHDL arrays.

Tcl Functions for Type Conversion

Tcl has predefined functions for converting to and from integer and double (floating-point) types. These functions also work with VHDL and Verilog types.

- `int(x)`
Converts its parameter to an integer.
 - If the parameter is a floating-point value, the conversion is as defined by Tcl. The decimal portion is truncated.
 - If the parameter is of `STD_ULOGIC` type, the conversion is that of the function `to_integer` in the `STD_LOGIC_ARITH` package. If the parameter is of type `BIT`, it is converted as an unsigned binary number whose most significant bit is the left-most bit. For both of these VHDL types, the 32nd bit is a sign bit. Values that have fewer than 32 bits are unsigned. If the vector length is greater than 32, only the right-most 32 bits are used.
 - If the parameter is a Verilog vector type, the conversion is as defined by Verilog for arithmetic operations, with x and z valued bits treated as zero bits, and with values longer than 32 bits truncated to the right-most 32 bits.
- `double(x)`
Converts its parameter to a floating-point value.
 - For integers, the conversion is as defined in Tcl.
 - For VHDL types `BIT` and `STD_ULOGIC`, the parameter is first converted to an integer, as described above, and then this integer is converted to a floating-point value.
 - For Verilog values, x and z bits are changed to zero, and the resulting bit pattern is extended with zeroes or truncated to 64 bits. The bit pattern is interpreted as a 64-bit representation of a floating-point number.

Two new type conversion functions have also been added:

- `vlog(x)`

Converts its parameter to the Verilog logic type.

- If the parameter is a vector of type `STD_ULOGIC` or `BIT`, the conversion is the same that occurs across language boundaries in a mixed-language simulation.
- If the parameter is an integer, the result is a Verilog vector of length 32.
- If the parameter is a floating-point value, it is converted to a 64-bit Verilog value that represents the same floating-point value.
- If the parameter is a string enclosed in quotes, it is converted to a Verilog logic vector representation of the ASCII codes for the characters in the string.

- `std_logic(x, size)`

Converts its first parameter to a `STD_ULOGIC` vector of the specified size.

- If the parameter is a Verilog literal, the conversion is the same that occurs across language boundaries in a mixed-language simulation for nets without strength information on the Verilog side.
- If the parameter is an integer, the conversion is that of the function `To_StdUlogicVector` defined in the `STD_LOGIC_ARITH` package.
- If the parameter is a VHDL value, it must either be of type `BIT` or of type `STD_ULOGIC`. It is an error if the first parameter is a floating-point value.

If the given size is too small to contain the converted value, excess bits are truncated from the left end of the vector. If the size is larger than necessary, the resulting value is padded on the left with zeros (0) for integer conversion and with the unknown value (U) for Verilog and VHDL value conversion. If the given size is zero, the size of the converted value is the size of the value being converted (32 for integers). You cannot specify a size less than zero.

Related Topics

- [Xcelium Simulator Tcl Commands](#)

Enabling Tk in the Simulator

You can use Tk with the simulator; however, you need a shared library version of Tk and the Tcl script files library that comes with it. The version of Tk currently required by the simulator is 8.6.8. Later versions of Tk will not work.

Tk is not included as part of the simulator product but is available on the internet. One site from which you can download version 8.6.8 is: <https://www.tcl.tk/>.

To use this website:

1. Navigate to *Software > Source Distributions > SourceForge*.
2. Select the 8.6.8 folder and download the following files:

- `tcl8.6.8-src.tar.gz`
- `tk8.6.8-src.tar.gz`

3. Untar these files in the same directory, and then execute the following commands:

- a. `cd your_tcl_dir/tcl8.6.8/unix`
- b. `./configure`
- c. `make`
- d. `cd ../../tk8.6.8/unix`
- e. `./configure --enable-shared`

This command could change. See the `README` file in the `tcl8.6.8/unix` directory to verify that the command shown above is correct.

- f. `make`

When you run *xmsim*, you can enable Tk with the following two commands:

```
xcelium> set tk_library your_tcl_dir /tk8.6.8/library
xcelium> load your_tcl_dir /tk8.6.8/unix/libtk8.6.so
```