VeriSim™ User Guide

Product Version: V1.22.1

Nov 2022

Contents

Glossary of Terms

Languages

All Language Reference Manual references are noted with the following abbreviations:

Note: Any references to other versions of the IEEE LRMs will explicitly contain the date of publication within the documentation.

- **V-LRM** : Verilog Language Reference Manual IEEE Std 1364™-2005
- **SV-LRM** : SystemVerilog Language Reference Manual IEEE Std 1800™-2017(Revision of IEEE Std 1800-2012)
- **VHDL-LRM** : VHDL Language Reference Manual IEEE Std 1076™-2000

Common Options

Options can be applied on the command line.

Getting Help

`-help` will list all options supported by the executable.

Standard Options

The following options are common to most HDL simulators. Those specific to a particular VeriSim product are annotated in the table.

Option	VeriSim	DVlcom	DVhcom	Description
`+ignore+item [+...]`	Y	Y		Specifies preprocessor directives, valid in other tools, that are to be ignored by VeriSim. When encountered, the directive and any other text on the same line will be ignored. Multiple items can be specified separated by +. This option may be given more than once.
`+incdir+dir[ +...]`	Y	Y		Specifies one or more directories to add to the include file search path. Separate path elements with '+'

Option	VeriSim	DVlcom	DVhcom	Description
				signs. This option may be given more than once.
`-incdir dir[:...]`	Y	Y		Specifies one or more directories to add to the include file search path. Separate path elements with ':' signs. This option may be given more than once.
`+define+NAME [+...]`	Y	Y		Define a preprocessor macro called `NAME` on the command line. `+define+NAME` will define the name such that `ifdef will see it. `+define+NAME=value` will assign a specific value to the name. This option can be given more than once.
`-define NAME[:...]`	Y	Y		Define a preprocessor macro called `NAME` on the command line. `-define NAME` will define the name such that `ifdef will see it. `-define NAME=value` will assign a specific value to the name. Separate define elements with ':' signs. This option may be given more than once
`-f` / `-F file`	Y	Y	Y	Include additional options in file. If `-f` is used, then any items in the file are interpreted relative to the directory in which VeriSim was invoked. Alternatively, `-F` will interpret pathnames relative to the location of the file being read.
`-l logfile`	Y	Y	Y	Specify the log file (default is `verisim.log` which logs stdout and stderr from VeriSim itself). To suppress creation of a log file, use `-l /dev/null`.
`-v file`	Y	Y		Specify a library file, which is scanned if required to resolve references to otherwise undefined module instances.

Option	VeriSim	DVlcom	DVhcom	Description
				This option may be given more than once. Each file may contain one or more module definitions which are used as required to resolve unresolved references.
`-y dir`	Y	Y		Specify a library directory, which is scanned if required to resolve undefined module references. This option may be given more than once. For each unresolved module, a file having the module's name, plus any extension, is searched in all `-y` path directories, and is loaded as required.
`+libext+xyz[+...]`	Y	Y		Specifies a suffix `xyz` to use for searching in `-y` library directories.
`-iter-limit n`	Y			Set scheduler iteration limit to detect infinite zero-delay loops (default: 1000000)
`-profile`	Y			Run time option to enable time profiling. **Note:** `-profile-start` and `-profile-end` also available for better granularity.
`-sv`	Y	Y		Make -sv20XX options apply to all SystemVerilog and unknown files regardless of suffix. **Note:** Exclusion of this option may defer to assumptions made on the filename, see below.
`-sv2005`	Y	Y		Treat SV-LRM'05 keywords as reserved in SystemVerilog and unknown files. **Note:** Exclusion of this option will may defer to assumptions made on the filename, see below.
`-sv2009`	Y	Y		Treat SV-LRM'09 keywords as reserved in SystemVerilog and unknown files. **Note:** Exclusion

Option	VeriSim	DVlcom	DVhcom	Description
				of this option may defer to assumptions made on the filename, see below.
`-sv2012`	Y	Y		Treat SV-LRM'12 keywords as reserved in SystemVerilog and unknown files (default). **Note:** Exclusion of this option may defer to assumptions made on the filename, see below.
`-sv2017`	Y	Y		Treat SV-LRM'17 keywords as reserved in SystemVerilog and unknown files. **Note:** Exclusion of this option may defer to assumptions made on the filename, see below.
`-sysvlog-ext`	Y	Y		Provide a set of one or more extensions, delimited by `:` or `,` that should be understood to represent SystemVerilog files. Initially only `.sv` is included in this set. Each use of this option will replace the known set.
`-timescale <unit/precision>`	Y			Provide a default timescale for any design unit which has no timescale directive. Note: if no timescale directive is given either in the code or via the -timescale option the simulator default of 1ns/1ns will be used.
`-top name:name`	Y			Specifies the top-level module for elaboration. If this option is not given, then all modules which are not instantiated by some other module are used as top-level modules, as per SV-LRM. If this option is given (possibly more than once) then only the named module(s) are used as top-level modules. You may specify multiple top-level modules using `-top` multiple times, or by using colons to separate names from the single

Option	VeriSim	DVlcom	DVhcom	Description
				argument. You need to specify at least one top-level module if pre-compiled libraries are involved in the design elaboration (e.g. such is the case with VHDL).
`-vhdl2000`			Y	Treat VHDL-LRM'00 keywords as reserved in VHDL and unknown files.
`-vhdl2008`			Y	Treat VHDL-LRM'08 keywords as reserved in VHDL and unknown files.
`-vhdl2019`			Y	Treat VHDL-LRM'19 keywords as reserved in VHDL and unknown files.
`-vhdl87`			Y	Treat VHDL-LRM'87 keywords as reserved in VHDL and unknown files.
`-vhdl93`			Y	Treat VHDL-LRM'93 keywords as reserved in VHDL and unknown files.
`-vlog1995`	Y	Y		Treat only V-LRM'95 keywords as reserved in Verilog files
`-vlog2001`	Y	Y		Treat V-LRM'01 keywords as reserved in Verilog files
`-vlog2005`	Y	Y		Treat V-LRM'05 keywords as reserved in Verilog files (default)
`-vlog_sv2005`	Y	Y		Treat SV-LRM'05 keywords as reserved in Verilog files
`-vlog_sv2009`	Y	Y		Treat SV-LRM'09 keywords as reserved in Verilog files
`-vlog_sv2012`	Y	Y		Treat SV-LRM'12 keywords as reserved in Verilog files
`-vlog-ext`	Y	Y		Provide a set of one or more extensions, delimited by `:` or `,` that should be understood to be Verilog files. Each uses of this option will replace the known set.
`-version`	Y	Y	Y	Prints the formal version name

Option	VeriSim	DVlcom	DVhcom	Description
`-version-verbose`	Y	Y	Y	Prints extra information about the build from which the release was built. This matches what VeriSim prints into the log file at the end of a simulation run and is potentially useful when communicating with the VeriSim support team.

Note: A file is recognized as containing code written against a particular version SystemVerilog by their primary or secondary extensions against any known extensions (`-sysvlog-ext` or `vlog-ext`). Unrecognized files can be forced to SystemVerilog using any of the `-sv*` options, otherwise Verilog is assumed.

The primary extension can also be used to identify compression, see Transparent Compression for supported compression types. See Input Filename Examples for examples using different filename extensions with and without compression specification.

Controlling Error Messaging

Text messaging is used to flag various exception conditions during compile, elaboration, or run phase. The header of a message is formatted in accordance with the exception severity, as follows:

- =F:[MessageName] for fatal errors
- =E:[MessageName] for errors
- =W:[MessageName] for warnings
- =N:[MessageName] for informational notes

After flagging an error or fatal error, the compilation phase is aborted. Warnings and informational notes do not affect the rest of the compile, elaboration or run phase.

The severity of a message may be altered using the following options:

Option	Description
`-error msg[:msg...]`	Promote the warning/informational messages to errors.
`-warn msg[:msg...]`	Demote certain error messages to warnings.
`-info msg[:msg...]`	Demote error/warning messages to informational.
`-suppress msg[:msg...]`	Suppress a message altogether.

where `msg` is the MessageName as given inside the square brackets in the header line. For example, to suppress a “missing timescale” error message, use

```
-suppress MissingTimescale
```

These options may be given more than once. For example:

```
-error ConflictAssignToVar -error MultiBlockWrite
```

In order to alter the severity of multiple messages, concatenate their names using a colon. For example:

```
-error ConflictAssignToVar:MultiBlockWrite
```

You can also promote/demote the severity of an entire class of messages by specifying `note`, `warning`, `error` or `fatal` instead of a message name. Overrides of individual messages take priority over such “global” overrides. e.g.:

```
-error warning -suppress MultiBlockWrite
```

will promote all warnings to errors, with the exception of `MultiBlockWrite`, which will be suppressed.

Not all error messages can be demoted. Error messages that can be demoted will be labelled “<demotable>”.

These options apply to VeriSim, DVlcom, and DVhcom.

Error Message Limits

Simulation may be terminated based on the count of run-time errors. You may enable this capability by setting the count threshold.

Option	Description
`-exit-on-error n`	Terminate the simulation if the count of run-time errors reaches the specified threshold `n`. `n` shall be a positive integer. By default, the threshold is undefined.

Effectively this option has the same effect as the following pseudocode.

```
initial begin
   wait (number_of_system_errors == n);
   $fatal(0, "The message limit has been reached");
end
```

This option is available in VeriSim and applies to the simulation.

Debugging

Option	Description
`-debug-verbose category[:...]`	Trace operation of one or more specified categories. Separate categories with ‘:’ signs. The categories are:

Option	Description
	pp - preprocessing, hier - instance hierarchy, lib - library maps.
`-g`	Enable symbolic debugging.

The `-debug-verbose pp` option traces operation of the preprocessor. The following actions are logged when enabled:

- Opening files
- Defining/undefining macros, including defining from command line
- Entering a new include file
- Resuming former input at end of include file

This option is available in VeriSim, DVlcom, and DVhcom.

The `debug-verbose lib` option traces design elements being placed in libraries and default library search paths. This option is available in VeriSim, DVlcom, and DVhcom.

The `debug-verbose hier` option prints the instance hierarchy and the module/configuration-rules for an instance. This option is available in VeriSim.

The `-g` option is provided for future use by a GUI debugger. For the moment, enabling symbolic debugging will allow backtraces printed when a fatal signal happens (e.g. null handle dereference) to have file and line number information.

There is no simulation-time performance penalty incurred with `-g`. However, compiles will be slower.

This option is available in VeriSim.

C Language Support

C or C++ source code files or object files and libraries can be passed to VeriSim to compile and link into the generated image.

Option	Description
`-cc c_compiler_name`	Specifies the name of the gcc-compatible C/C++ compiler
`-c-opts options`	Specifies additional options to pass to the C/C++ compiler
`-ld-opts options`	Specifies additional options to pass to the linker.

These options are available in VeriSim.

Reproduce

Option	Description

Option	Description
`-repro verisim.env`	Reproduce from compile/run file.

When a tool, such as VeriSim, DVlcom, or DVhcom is run, it creates a file named `verisim.env`, `dvlcom.env`, or `dvhcom.env`, respectively. The file contains a record of the invocation environment: the path to the executable binary, the command line arguments and the shell environment variables. This file is useful to support personnel to replicate a run, regardless of the complexity of any scripts that may be wrapped around the tool. The `-repro` option will run the tool using the environment that was recorded in the given environment file.

Controlling Compiler Behaviour

Option	Description
`-j N`	Specify number of parallel compile jobs (default N=1).
`-no-incr-compile`	Indicates that an incremental compile is not necessary and that the related overhead and peristent artifacts will not be needed.
`-build-all`	Rebuild all modules (not incremental)

VeriSim uses an incremental/parallel compile strategy. It turns out that the bulk of the compile time is used in final code generation. To minimize compile time, after parsing/elaboration and optimization, VeriSim determines which modules in the design need code regenerated. The final code generation step is invoked for each such module, potentially in parallel.

The `-j` option is helpful to compile your code faster if you have multiple CPUs in your machine. It specifies the number of compile jobs that can run in parallel (default 1). Increasing the parallelism can reduce compile time, at the expense of compile machine memory.

The incremental build function is designed to be robust: it ought to compile only what is necessary, and never compile less than necessary. However, should there be a need to force a recompile of every module (e.g. to purge suspected stale cache information) the `-build-all` option will do just that.

As well the `-no-incr-compile` will have a similar effect as `-build-all`. In additon no overhead will be used to allow for incremental compiles in the future, and all compilation artifacts are removed after a successful compile. The results are a slighty faster compile and less resulting disk space.

Instantiation Depth

Option	Description
`-max-inst-depth n`	Sets maximum instantiation depth (default 100).

This option sets the maximum number of levels of hierarchy supported. The limit exists to detect runaway recursive instantiations, where a module instantiates itself with the same parameters, directly or indirectly. If the limit is too low for your design, then you can raise it.

NOTE: Controlled recursive instantiation is supported and encouraged! With “controlled” instantiation, a module instantiates itself, albeit with different parameter values. Generate statements are used to test the parameters and terminate the recursion.

Defining Parameters on the Command Line

Option	Description
`-defparam name=value`	Overrides a parameter or generic in hierarchy.

This option is available in VeriSim. It may be given more than once.

The `name` portion may contain a dot, in which case it is a full hierarchical path to some arbitrary point in the Verilog design.

Alternatively, the `name` may not contain a dot, in which case it affects each top-level:

- Verilog module having a parameter by that name
- VHDL entity having a generic by that name

The value must be a constant using proper syntax. Binary/octal/hex/decimal bases are accepted, as are integers, floating-point numbers and string literals. No other lexical element, including assignment patterns, is accepted.

Since the `:` character is used to delimit multiple parameter definitions, if the value is a string literal and contains `:` then it should be encapsulated with either single or double quotes.

In all cases, the effect is to override a parameter in the design, as if the Verilog `defparam` statement was given with the same arguments in some top-level module of the design.

VHDL-Specific Options

Option	Description
`-explain`	Provide verbose explanation of inviability/ambiguity errors
`-vhdl-no-ov-chk`	Disable integral arithmetic overflow checking

VHDL permits operator overloading, and disambiguates expressions based on the types of the operands as well as the type of the result. Each “innermost complete context” provides an expected data type for the expression. The rules for resolving overloads can be quite arcane and complex. The result of a failed constraint is either an ambiguity (more than one solution found) or a “no viable alternatives” (no solutions found) error.

The `-explain` option enables extra detail for ambiguity and no-viable alternatives errors. The detail ought to help debug constraint failures in the more complex cases. However, the detail report can be voluminous, and a distraction if the failure is easy enough to debug without it. It is recommended to enable this option only as required.

VHDL requires overflow checking for integral arithmetic. VeriSim supports overflow checking, but other VHDL implementations do not. `-vhdl-no-ov-chk` will disable overflow checking for compatibility with legacy code, or for slightly improved runtime performance.

Controlling SystemVerilog Assertions

Option	Description
`-vacuous`	Execute the pass action regardless whether success is vacuous or nonvacuous. By default, execute the pass action for nonvacuous successes only.
`-no-sva`	Disables SVA support during compilation, causing sequences, properties and concurrent assertion statements to be ignored.

These options are available in VeriSim and DVlcom and applies to compilation of SystemVerilog assertions.

Controlling UDP update region

By default both sequential and combinational UDP output updates are scheduled in the ACTIVE region. It may be desirable to schedule sequential output updates in the NBA region.

Option	Description
`-seq-udp-nba-region`	Schedule an update of a sequential UDP output into the NBA region

These options are available in VeriSim and apply to simulations of SystemVerilog UDP components.

Controlling optimization

Option	Description
`-noopt`	Disable optimization

These options are available in VeriSim and apply to the compiling of a simulation image.

`-noopt` disables most optimizations. Use only if you suspect a bug in the optimizer. Up to 10x performance degradation can be seen with optimizations disabled.

Gate-Level Simulation

These options are available in VeriSim and apply to simulations of SystemVerilog components.

Min/Typical/Max timing

Option	Description
`-comp-mindelays`	Compile minimum delays
`-comp-typdelays`	Compile typical delays
`-comp-maxdelays`	Compile maximum delays

The above options affect the execution of “mintypmax expressions” which are triples of values in the form (min : typ : max) at compile time. Using one of these options locks the compile of the design to the specified timing delay. Note that using these compile options disables the usage of the run-time options mentioned below since the selected compile options will lock in the selected delay in the design.

When one of the above compile-time options is given. a mintypmax expression whose alternatives are constants is considered to be a true constant, meaning it can be used to assign Verilog parameters.

Option	Description
`-mindelays`	Use minimum delays
`-typdelays`	Use typical delays (default)
`-maxdelays`	Use maximum delays

The above options affect the execution of “mintypmax expressions” which are triples of values in the form (min : typ : max). At run-time, the minimum, typical or maximum value is used depending on which run-time option is selected.

All values of “mintypmax expressions” are compiled in at compile time. They can be “selected” with the above compile time options and at run time only the “selected” expression is evaluated. If the compile time options are not used and only the run time options are used then only the “selected” expression is evaluated, noting that the expressions do not have side effects.

Controlling Timing Imposed by Specify Blocks

The following options interact with timing controls indicated using specify blocks within modules.

Modifying Module Path Delays Globally

Option Description

Option	Description
`+/-nospecify`	The specify block will be completely ignored and will not be available for SDF overrides during runtime.
`-specify-zero`	The result will be all specify path delays are overridden with 0 even in the presence of SDF annotations.
`-specify-unit unit`	The result will be all specify path delays are overridden with the unit given even in the presence of SDF annotations.

These options are available in VeriSim and apply to simulations of SystemVerilog components.

When actual timing values are not needed specify blocks can be ignored entirely or setup to use placeholder values.

Modifying Pulse Reject and Error Limits

Option	Description
`-pathpulse`	The delay model is modified to include filtering based on the $PATHPULSE settings within a specify block. If command line options pulse_e or pulse_r are also specified these shall take precedence over the $PATHPULSE values in the specify block.
`+transport_path_delays`	Modifies the default error and reject limits to be 0, without this option they would be the delay of the trailing edge of a pulse. Please see the Delay Model for Module Paths section for details on how this option interacts with other options.
`-pulse_e percent`	Sets the percent of module path delay for error limit (default: 100). If a $PATHPULSE specparam is used inside a specify block and this invocation option is used (with also -pathpulse option), then the $PATHPULSE will take precedence.
`-pulse_r percent`	Sets the percent of module path delay for reject limit (default: 100). If a $PATHPULSE specparam is used inside a specify block and this invocation option is used (with also -pathpulse option), then the $PATHPULSE will take precedence.

These options are available in VeriSim and apply to simulations of SystemVerilog components. They control how pulses are handled with respect to reject and error limits.

Full inertial pulse handling is when the reject and error limit are both equal to the delay of the trailing edge of the pulse. This is the default operating mode.

Full transport pulse handling is when the reject and error limit are both equal to 0 and all transitions/pulses are transferred to the output.

Specify spec params $PATHPULSE (used with -pathpulse option) along with other command line options allow the modification of reject and error limits. Please see the section Delay Model for Module Paths.

By default the pulses handling mode will be full inertial with the default error and reject limit equal to the trailing edge delay unless overridden with -pulse_e or -pulse_r. In this mode the $PATHPULSE specparams in specify blocks will be ignored.

Modifying Pulse Style Globally

Option	Description
`-pulse_e-onevent`	Indicate pulse errors on event. When a pulse (analyzed on the output) is recognized to have pulse width >= the reject limit and < error limit the output will be set to x starting at the leading edge of the output pulse (ie the delayed time of the leading edge).
`-pulse_e-ondetect`	Indicate pulse errors on detection. When a pulse (analyzed on the output) is recognized to have pulse width >= the reject limit and < error limit the output will be set to x starting immediately upon detection of the pulse (ie when the input changes causing the trailing edge of the pulse)

These options are available in VeriSim and apply to simulations of SystemVerilog components.

Pulse style invocation options (on-detect or on-event) are mutually exclusive and when specified they override the settings in the specify block.

- These options take precedence over the pulsestyle_ondetect and pulsestyle_onevent in all specify blocks.
- If there is no indication of pulse style in the specify block or the command line, the “on event” style will be used.

Handling Negative Pulses

Option	Description
`-noshowcancelled`	Do not show negative pulses. When a negative pulse is detected at the output it is rejected without setting the output to the error state and a warning message is issued unless -pulse_e-no-cancelled-msg is specified.
`-showcancelled`	Show negative pulses as X. When a negative pulse is detected at the output, the output is set to x either immediately or on event depending on the setting of the pulse style. A warning message is issued unless -

Option	Description
	pulse_e-no-cancelled-msg is specified.

These options are available in VeriSim and apply to simulations of SystemVerilog components.

Input pulses resulting in a negative pulse at the output can either be automatically cancelled or shown to be in the error state with the above settings.

- These options take precedence over the showcancelled/noshowcancelled in all specify blocks.
- If there is no indication of showcancelled/noshowcancelled the specify block or the command line, the “noshowcancelled” style will be used.

Managing Pulse Messages

Option	Description
`-pulse_e-no-cancelled-msg`	Turn off warnings when negative pulses are cancelled. By default a warning message will be printed to stderr indicating a negative pulse has been recognized and cancelled. When this option is used those messages are suppressed.
`-pulse_e-no-warn-msg`	Turn off warnings when pulses cause error state. By default a warning message will be printed to stderr indicating when a pulse with width >= reject limit and < error limit is recognized and set to the error state. When this option is used those messages are suppressed.

These options are available in VeriSim and apply to simulations of SystemVerilog components.

Standard Delay Format (SDF) Annotation

Option	Description
`+/-nosdf`	Run-time option to disable $sdf_annotate calls.
`+multisource_int_delays`	Run-time option to turn on interconnect multi-source delays causing INTERCONNECT paths to behave like module paths (ie IOPATH) including pulse handling. If not specified, INTERCONNECT constructs will be handled like PORT connections and will be more efficient.
`-sdf-verbose`	Run-time option to log all SDF annotation errors. Without this option only the first 10 SDF errors of the same type will be logged.

These options are available in VeriSim and apply to simulations of SystemVerilog components when SDF annotations are provided.

SDF annotation is on by default and will log errors to the log file specified in the $sdf_annotate call. If no log file is given, it will use file name sdf.log.

Controlling Timing Checks

Option	Description
`+/-notimingchecks`	Compile-time option to not include timing checks. The timing checks are not evaluated and are not available for SDF annotation. Delayed signals provided to $setuphold or $recrem are driven with 0 delay to the base signal.
`+/-nonegtchk`	Compile-time option to set negative limits to 0 and disable the NTC algorithm's calculation of delayed signals. Delayed signals provided to $setuphold or $recrem are driven with 0 delay to the base signal.
`+/-ntcnotchk`	Compile-time option to disable timing checks, howeverthe NTC algorithm will calculate the delays. Those timing checks that could contribute to the delay calculation remain enabled and are available for SDF annotation. Delayed signals provided to $setuphold or $recrem are driven with the NTC calculated delay to the base signal.
`+/-nonotifier`	Run-time option to disable notifier signal toggling by timing checks.
`+/-no_tchk_msg`	Run-time option to suppress timing violation messages. Note that the timing checks are still evaluated. Delayed signals provided to $setuphold or $recrem are driven with the NTC calculated delay to the base signal.
`+/-tchk_msg_start time`	Run-time option to start reporting violations no earlier than `time`. Note that the timing checks are always evaluated prior to the `time`, but violation reports are suppressed. In the absence of this option, the behaviour corresponds to `-tchk_msg_start 1`. `time` is a mandatory argument. It consists of a number and an optional time unit, e.g. "100ms". If no time unit is given, then the global time precision is assumed. Design teams may want to consider suppressing the violation reports till the device reset.
`-timingcheck-specs file`	Enable/Disable timing check at a specific scope using a specification file. See Setting Options by Scope Using Specification File for file details and examples.

These options are available in VeriSim and apply to simulations of SystemVerilog components.

By default, the timing checks are enabled and may be overridden with SDF annotation. The Negative Timing Check (NTC) algorithm to calculate delays is enabled as well.

User may improve efficiency by specifying either -notimingchecks or -nonegtchk option.

Delay Mode Controls

Option	Description
`+delay_mode_zero`	Compile time option to turn off distributed delays (associated with nets, gates, switches, UDPs, and continuous assignments). It also disables and removes module path delays and timing checks.
`+delay_mode_unit`	Compile time option to overwrite nonzero distributed delays by 1 precision time unit. It also disables and removes module path delays and timing checks.
`+delay_mode_path`	Compile time option to overwrite all distributed delays by 0. Only module path delays are in effect.
`+delay_mode_distributed`	Compile time option to disable and remove all path delays. Only distributed delays are in effect.

These options are available in VeriSim and apply to simulations of SystemVerilog components.

The user may specify the delay mode through the above command line options and/or compiler directives of the same name. Note that the above command line options are mutually exclusive. The command line option acts as if it is the first compiler directive in the topmost source file. A subsequent delay mode directive supersedes the current one.

When running with SDF, consider the effect of setting the delay mode. If the constructs are removed as indicated above, they will not be available for SDF to override. The +nosdf option can be used to disable SDF if needed.

Combinational Glitch Removal

Option	Description
`-opt-comb-glitch`	Enable combinational glitch mitigation

These options are available in VeriSim and apply to simulations of SystemVerilog components.

RTL code often uses a coding style intended to avoid inadvertent latch inference:

```
always @* begin
    x = 0;
    y = 0;
    if (a) x = 1;
    if (b) y = 1;
end
```

In the event that `b` rises while `a` is already high, a glitch will be created on `x` as it is set to 0 and then immediately set to 1. Strictly according to the SV-LRM this glitch must be produced, as other processes may be sensitive to it. However, it is possible to code simulation logic loops by creating two blocks sensitive to each other's glitches. This style may not result in an actual combinational loop during synthesis, and it may not cause a simulation loop on other simulators, depending on how the tools optimize and schedule the processes. For the benefit of users migrating from other tools with legacy code that causes a simulation loop due to this coding style, the option `-opt-comb-glitch` will enable a transform that suppresses some of these glitches. The resulting simulation will not be SV-LRM compliant, and may break conformant code, but it may also allow non-conforming legacy code to simulate properly.

SV-LRM Reference: 4.9.3 second paragraph, in the case where no delay is specified.

Other Run-Time Options

Option	Description
`-run-until time`	Specify end `time`. The end time consists of a number and optional time unit, e.g. "100ms". If no time unit is given, then the simulation time precision is assumed.
`-heartbeat N`	Specify heartbeat log message interval.
`-linebuf`	Force line-buffered output (default).
`-no-linebuf`	Do not force line-buffered output. Potential speedup for verbose simulations.
`-sv_seed n`	Specify `n` as the SystemVerilog random seed. `n` may be an arbitrarily large hexadecimal number, or 'random' to select a random seed. (default seed=1, if not used)
`-reseed t=n`	Specify time `t` to reseed the SystemVerilog randomization with seed `n`.
`-override-finish-completion n`	Specify an override value to use for all `$finish` (and `$stop` and `$fatal`) calls to control the diagnostic information produced. Only 0, 1 or 2 is permitted.

These options are available in VeriSim and apply to the simulation.

The `-heartbeat` option, if given, causes a message to be printed to simulation output each time the specified interval elapses. It provides a good way to track performance of the simulation and to ensure that a simulation is not "hung" when it gets to a point where the testbench does not output anything.

By default, most simulation output goes to the log file as well as `stdout`, and certain error messages go to the log file as well as `stderr`. If `stdout` and `stderr` are connected to a Linux terminal window then any buffering on those channels is flushed at the end of each line of output. However, if `stdout` is redirected to a file, then the buffering is flushed:

- at the end of each line of output, whenever `-linebuf` is in effect. If both `stdout` and `stderr` are redirected to the same file, then their relative order is preserved. Note that enabling `-linebuf` may impact the performance, in particular for verbose simulations.
- only when the internal buffer gets full, whenever `-no-linebuf` is in effect. If both `stdout` and `stderr` are redirected to the same file, then output from one may be arbitrarily reordered with respect to the other due to the differences in buffering. This is standard Linux behavior. Note that enabling `-no-linebuf` may improve the performance, in particular for verbose simulations.

Setting Options by Scope Using Specification File

Some features allow options to be set differently for different parts of the design by providing a specification file. The specification files for each feature will be of the same structure, however the details will be feature specific. The file will be read at either runtime or compile time depending on the feature, will have one option specification per line, and will be processed in order.

The features supporting specification files include:

- wave dump scope: `-wave-scope-specs file`
- timing checks: `-timingcheck-specs file`
- toggle, block, and/or expression coverage: `-code-cov-scope-specs file`
- design object visibility `+acc` related options: `-acc-specs file`

General Specification File Structure

Each line will be structured as a list of fields separated by spaces.

```
<spec_keyword>  {depth:}<scope>  <options>
```

Fields:

`<spec_keyword>` Specification level keyword (not all are supported by all features):

- module (affects all instances of this type of module)
- instance (affects only the specified instance and not its descendants)
- path (affects the path and its descendants upto given depth if specified)

- signal (affects the specific signal)

`{depth:}<scope>` Hierarchical path in the design

- Optionally depth can be indicated for `path` type lines with the following format. If a depth is provided then the option will be applied to the path and "depth" levels below that path. If depth is omitted the option is applied to the full depth under the path.
- scope is a string indicating the specific design component to which the options are applied.

`<options>` List of one or more feature specific options

Comment characters supported: # or // Commented out lines will be ignored

See feature specific examples below.

Specific Examples by Feature

Wave Dump Scope

Supported spec_keywords: instance, path, signal

Depth with the path keyword is supported

Options: +

- if + the scope is added to the waves
- - is not supported at this time

Run time option for example:

```
-wave-scope-specs wave_scope_file
```

With example wave_scope_file:

```
# dump all of rung0
path stimulus.t_0.rung0 +

// dump rung1 only at level of rung1
instance stimulus.t_0.rung1 +

// dump rung2 and next level
path 2:stimulus.t_0.rung2 +
path stimulus.t_0.rung2.to1 +

# dump rung3 only
path 1:stimulus.t_0.rung3 +

// a couple of signals from rung 4
```

```
signal stimulus.t_0.rung4.init +
signal stimulus.t_0.rung4.tq +
```

Timing Check

Supported spec_keywords: instance, path

Depth with the path keyword is supported

Options: [+|-]
[*setup*|hold|*setuphold*|removal|*recovery*|recrem|*skew*|timeskew|*fullskew*|width|*period*|nochange]*

- if + and one or more timecheck types are specified those timecheck(s) are enabled
- if + and no timecheck types are specified then all timechecks specified in the design at the given scope level are enabled
- if - and one or more timecheck types are specified those timecheck(s) are disabled
- if - and no timecheck types are specified then all timechecks specified in the design at the given scope level are disabled

Run time option for example:

```
-timingcheck-specs time_spec_file
```

With example time_spec_file:

```
# turn off rung0 and descendants to0 and to1
path stimulus.t_0.rung0 -$setup$hold

# turn off all timing specs for rung1.to1 only
instance stimulus.t_0.rung1.to1 -

// turn off rung2 and descendants then turn to1 back on
path stimulus.t_0.rung2 -$setup$hold
path stimulus.t_0.rung2.to1 +$setup

# should turn off rung3 and all descendants
path 0:stimulus.t_0.rung3 -$setup$hold

// should turn off rung4 at level 1 only which would not include descen
dants, but there are none at rung4
path 1:stimulus.t_0.rung4 -$setup$hold
```

Code Coverage Scope

The code coverage scope spec file is used for the code coverage features that are enabled and support scoping specification.

Supported spec_keywords: instance, path, signal

Depth with the path keyword is supported.

Options: +[b|t|e], -[b|e]

- if + the scope is added to the code coverage features that are turned on
- if `+b` the scope is added to block coverage
- if `+t` the scope is added to toggle coverage
- if `+e` the scope is added to the expression coverage
- A scope can be added to multiple code coverage features at a time. E.g. '+te' or '+et' will add the scope to both toggle and expression coverage.
- if - the scope is removed from the code coverage features that are turned on and support removal
- if `-b` the scope is removed from block coverage
- if `-e` the scope is removed from expression coverage
- A scope can be removed from multiple code coverage features at a time. E.g. `-eb` or `-be` will remove the scope from both block and expression coverage
- Toggle coverage does not support `-`.

While the same file will be used for all coverage types, the keyword signal is only applicable to toggle coverage and will be ignored for block and expression coverages.

For toggle coverage, -code-cov-scope-scpec option only applies during runtime.

For block and expression coverage the '+' option, and the '+' option only, gets used during compile time to instrument coverage collection. The -code-cov-scope-specs argument can optionally be used at runtime to allow '+' and '-" entries to refine which scopes instrumented at compile time have block or expression coverage stored.

Compile and run time option for example:

```
-code-cov all -code-cov-scope-specs code_cov_scope_file
```

With example code_cov_scope_file:

```
# Instrument block and expression coverage for dut and all levels of hi
erarchy underneath it,
# and collect toggle coverage for all nets and variables within dut wit
h callback permission
path top.testbench.dut +
# Collect toggle coverage for nets and variables with callback permissi
on in instance A
instance top.testbench.A +t

# Collect toggle coverage from single net.  This still requires that th
e net be given
# callback permission at compile time.  Signal scopes do not apply to e
xpression or block coverage.
signal top.testbench.clk +

# But do not collect block nor expression coverage for a particular ins
tance (takes effect during
# runtime, not during compile instrumentation)
instance top.testbench.dut.a.b.c -be
```

Design Object Visibility

Supported spec_keywords: module, path, signal

Depth with the path keyword is not supported

Options: +[rwcbf]+

- + followed by one or more of r, w, c, b, f turns on the functionality as described with the +acc command line option
- - is not supported

Compile time option for example:

```
-acc-specs acc_spec_file
```

With example acc_spec_file:

```
# apply +acc+rb to my_var
path top.dut.my_var +rb

# if the general +acc cmdline option included f this removes f
top.dut.my_var -f
```

Language Conformance Options

SystemVerilog is not rigorously defined, and it has evolved over the years. Unfortunately, a good many testbenches, including the reference implementation of UVM itself, will not compile on a strictly conforming SystemVerilog compiler. For this reason VeriSim provides switches to relax its interpretation of the SV-LRM.

Option	Description

Option	Description
`-shared-unit-scope`	Use single compilation-unit scope shared by all compilation units
`-separate-unit-scopes`	Give each compilation unit its own compilation-unit scope
`-all-class-spec`	Create default specializations for all parameterized classes
`-all-pkgs`	Generate code for all packages, even those not referenced
`-allow-int-enum-assign`	Allow assignment from integral expression to enum without a cast
`-allow-string-int-assign`	Allow assignment between strings and integers without a cast
`-implicit-bitstream-assign`	Allow assignment between any bitstream types without a cast
`-int-time-literal`	Treat time literals as integers
`-allow-ext-vif`	Allow implicitly specialized virtual interfaces, with restrictions
`-allow-self-vif`	Allow use of bare interface identifier as self virtual interface reference
`-allow-fwd-pkg`	Allow references to items in packages declared later in the code

Treatment of Compilation Unit Scope

Each file referenced on the `verisim` or `dvlcom` command line, together with everything the file transitively includes is treated as a potential compilation unit. The “compilation unit scope” is a scope that surrounds the compilation unit. Clause 3.12.1 of the SV-LRM requires that tools support both of the following options:

- A single compilation unit scope that encloses all files in the design. This option is selected with `-shared-unit-scope` and is the default.
- A separate scope for each potential compilation unit. This option is selected with `-separate-unit-scopes`.

Treatment of Unreferenced Packages and Classes

By default, VeriSim will compile the code for a package into the final image only if the package is referenced by the rest of the design in some way - either through use of an identifier using `pkg_name::id_name` syntax, or through a package import.

However, it is possible to define a package having a class that has a static initializer that calls some method:

```
package P;
  class C;
    static bit foo = other_pkg::other_class#(C)::do_something();
  endclass
endpackage
```

If nothing else references package P then the package will not be compiled by default. However, some testbenches may expect P to be compiled, and at run time, the static initializer calls do_something() which may result in creation of an instance of C. UVM tests work this way: when you write `uvm_component_utils(test) you are creating a static initializer that registers the class with the UVM factory, from where it can be instantiated if selected on the command line. In order to get this style to work, the -all-pkgs option is provided, which will force all packages to be compiled.

Along similar lines:

```
class C#(type T = int);
  static bit foo = other_pkg::other_class#(C)::do_something();
endclass
```

By default, VeriSim will not create a default specialization for a parameterized class. Instead, it will create specializations only for what is actually referenced. Again, any static initializers in parameterized classes that are not instantiated will not get run. The option -all-class-spec will override this behavior and force the creation of the default specialization.

Cast Permissiveness

The SV-LRM prohibits assignment of an integer to an enum without an explicit cast. Nevertheless, it appears that other tools do permit this, and legacy testbenches or third-party verification IP code may rely on this. The -allow-int-enum-assign switch will suppress this check, making enums almost useless (you have the enum methods, but no type safety) but allowing broken code to compile.

Similarly, other tools seem to permit arbitrary assignment between integers and SystemVerilog strings without an explicit bitstream cast. We believe this to be extremely dangerous, as code errors will not be caught. More generally, some tools permit any bitstream type to be assigned to any other bitstream type without an explicit cast! The -allow-int-string-assign will permit cast-free assignments between SystemVerilog strings and integers; -implicit-bitstream-assign permits any bitstream cast without an explicit cast.

Integral Time Literals

The SV-LRM specifies that a time literal such as 1us is a realtime (floating point) value, scaled to the time unit currently in effect for the containing design element. Consequence: time literals cannot be used in constraints, as constraints must involve integral values only (at least for bidirectional solving). At least one tool seems to support real-valued constraints, and therefore allows time literals in constraints.

The `-int-time-literal` switch causes time literals to be compiled as integral values, thereby permitting use in constraints. However, the user must ensure that the final scaled value of any such literal does not have a fractional portion, as the fraction will be lost.

Implicitly Specialized Virtual Interfaces

The SV-LRM prohibits any interface having interface ports, or hierarchical references that leave the scope of the interface, from being used as a virtual interface. VeriSim depends on this restriction to allow optimum code generation.

For best code generation, VeriSim compiles each specialization of a module or interface separately. For example, `foo#(4)` is a completely separate compiled entity from `foo#(8)`. If the parameter is used to select the width of a data object then separate compilation permits optimal code for each to be generated. This is straightforward, as the parameterization is explicit.

However, there are other conditions that cause specialized compilation. We refer to these as "implicit specialization" because no explicit parameters are involved.

Interface ports

```
interface foo(interface ii);
initial $display(ii.x);
endinterface

module top;
other_if oo();
other_if2 oo2();
foo u1(oo), u2(oo2);
```

This example shows an interface `foo` having a generic interface port. There are two instantiations of `foo`: one using `other_if` and the other using `other_if2`. Clearly, if `other_if` and `other_if2` are two different interfaces, one cannot efficiently use the same code for both instances of `foo`.

External Hierarchical References

```
interface foo;
initial $display(upward.x);
endinterface
```

Another cause of implicit specialization: upward hierarchical references. Clearly, if there were two instances of `foo` somewhere in the hierarchy, then `upward.x` may refer to two completely different objects. Again, it is not reasonable to have one body of compiled code for both cases.

Virtual Interfaces

In SystemVerilog, a virtual interface handle is a variable that can point to any compatible instance of the interface.

Variables and nets may be accessed. Tasks and functions may be called. However, an underlying assumption is that all instances of the interface that the handle can possibly point to are equivalent, compiled in exactly the same way. This is not true for implicitly specialized interfaces, justifying the SV-LRM restriction.

However, it turns out that other simulators do seem to allow this. The `-allow-ext-vif` switch can be used to recover some compatibility in this case. When VeriSim detects implicitly-specialized interfaces being used as a virtual interface, it will generate alternate (but less efficient) code for task/function calls to allow these calls to work. Direct access to any other object is still prohibited.

Virtual Interface Self-Reference

Some tools allow the use of a virtual interface identifier as a self-reference:

```
class registry#(type T = int);
virtual void set(T item);
...
endclass

interface foo;
initial registry#(virtual foo)::set(foo); // Register myself
...
endinterface
```

This treatment breaks standard Verilog syntax in the case where an implicit wire would have the same name as its containing interface:

```
interface suba(wire clk);
endinterface

interface subb(wire clk);
endinterface

interface clk;
    suba u1(clk);
    subb u2(clk);
endinterface
```

A classic SystemVerilog compiler would need to bind the references to `clk` to an implicit wire. Treating these references as interface self-references would break this function.

VeriSim supports interface self-references if `-allow-self-vif` is given. This achieves compatibility with vendor extensions for testbenches that require it, but breaks implicit wires. The default is to preserve compatibility with implicit wires.

Seed Argument

The SV-LRM requires that the `seed` argument to `$random()` be an integer variable. VeriSim permits the passing of a constant. However, behavior may differ compared to other simulators.

Static / Automatic Class Methods

The SV-LRM requires that all class methods have automatic lifetime. However, some user code appears to rely on support for static methods.

One point of clarification: the keyword `static` is overloaded: it can be used to indicate a storage lifetime, and it can also be used to indicate a method that can be invoked without an instance variable (i.e. no value of `this`). The order of declaration matters:

```
class C;
    // case (1): instance method whose arguments and local variables
    // have static lifetime
    task static foo();

    // case (2): non-instance method whose variables
    // have automatic lifetime
    static task foo();
endclass
```

The SV-LRM prohibits case (1). VeriSim permits it. Note that `pure` or `virtual` methods must have automatic lifetime.

Constraint Solver Options

The following apply to solving of SystemVerilog constraints.

Name	Description
`-cs-use-bdd`	Use BDD solver
`-cs-use-sat`	Use SAT solver
`-no-cs-range`	Disable range analysis during constraint solving
`-cs-array-max n`	Sets `n` as the max value for the randomized array.size (default 1000000)
`-cs-randc-max n`	Set `n` as the max number of bits in randc variables (default 16)
`-try-all-soft-`	Try all soft constraints enabled up front.

Name	Description
`first`	

General Approaches

The constraint solver iteratively executes two steps to solve a constraint set:

- Algebraic simplification of expressions involving only state variables; expansion of conditional and `foreach` constraints.
- Bidirectional constraint solving: synthesizing the constraint set into a boolean logic representation and using a boolean logic solver to find the solution.

There are two boolean logic solvers available:

The BDD solver uses reduced-order binary decision diagrams (BDD) to build a graph representing *all* possible solutions to the constraint problem. A random walk is then taken to select a solution. This approach has the advantage that each possible solution is equally likely to be selected. However, the BDDs are time-consuming to construct, and run time may not be acceptable for all but the most trivial problems.

The SAT solver recodes the constraint set into conjunctive normal form (CNF) and then uses a modified boolean satisfiability solver (SAT) to come to a solution. This approach scales to much larger problems. However, individual solutions do not necessarily have an equal probability of being selected.

The SAT solver is currently the default. However, the choice of solver can be forced: `-cs-use-bdd` to force use of the BDD solver, and `-cs-use-sat` to force use of the SAT solver.

Range Analysis

Range analysis is an optimization used to reduce the amount of work the logic synthesis engine must do. It attempts to infer valid ranges for each random variable, and given that, determines whether individual bits of the random variable must be constant, or equal to some other bit. e.g.

```
rand int a, b;
rand bit x;
constraint Q {
    a inside { [0:100] };
    b inside { [0:100] };
    x -> (a * b < 1000);
}
```

Without range analysis, `a` and `b` would be considered 32-bit integers by Verilog rules, so the multiplication would be a full 32x32 multiply. However, range analysis is able to infer that:

- a can be recoded as {25'b0,a[6:0]}
- b can be recoded as {25'b0,b[6:0]}
- Since the condition x is more specific than any conditions that led to the inferences on a and b, the above recodes are valid for the multiplication.

As a result, the synthesis engine will build out a 7x7 multiply which will result in a much smaller BDD or CNF constraint set.

However, we have seen constraint sets where little gain is to be had from range analysis, and enough different conditionals were used that range analysis actually costs more time than it saves! Typically this happens with constraints of the form:

```
rand bit [7:0] insn_name;
rand bit [7:0] opcode;
constraint Q {
  insn_name == INSN_LD_IMM -> opcode == 8'hA9;
  insn_name == INSN_RET    -> opcode == 8'h60;
  insn_name == INSN_CALL   -> opcode == 8'h20;
  // etc.for 100+ opcodes
}
```

Each implication antecedent creates a new condition regime for range analysis to deal with, which slows down efforts to determine which set of inferences can be made at any given point.

For such pathological cases the -no-cs-range switch can be used to disable range analysis.

Bounds on Otherwise Unbounded Behavior

The -cs-array-max option is used to provide a default constraint for code such as the following:

```
rand int q[$];
rand int x;

constraint CC { q.size() == x; }

std::randomize(q);
```

Absent any other constraint, the size of the queue could be randomized to any arbitrary 32-bit value. Since each queue element occupies 8 bytes (4 for the value, 1 for the rand_mode bit per element, and 3 padding as necessitated by x86_64 ABI rules) you could easily have an array of up to 2 billion elements taking up 16GB of memory - assuming you have enough memory. Instead, the constraint solver imposes an arbitrary maximum size of a million elements; the -cs-array-max option allows the limit to be overridden.

The -cs-randc-max option is used to control behavior of code such as:

```
randc int x;
std::randomize(x);
```

Internally, `randc` works like dealing cards: a deck of cards is procured, shuffled, and a single card is dealt on each call to `randomize`. Once the deck is exhausted, it is shuffled anew.

There is one card for each possible value of the variable. For a 32-bit integer, we need 4 billion cards, and the shoe that holds this deck is 16GB in size. Again, doable (but slow!) if you have the memory. The option is used to limit the size of the deck in cases where code is being silly. The default value of 16 allows UVM to run properly.

Controlling Soft Constraints

Given a series of N soft constraints, the constraint solver enables each soft constraint one by one. If a solve is possible with a soft constraint left in, it will be left enabled, otherwise it will be disabled. N soft constraints requires N+1 iterations of the constraint solving algorithm to try all possibilities. This is inefficient if the soft constraints are meant to guide the solver and are not expected to cause a failure. For the latter scenario, the `-try-all-soft-first` option will first try a solve with all soft constraints enabled. If the solve succeeds, then no more work need be done. Otherwise, the soft constraints are enabled one by one in priority order as before.

Coverage Options

The following applies to SystemVerilog components only, with the exception of line coverage, which applies to SystemVerilog and VHDL components.

Controlling Functional Coverage

Name	Description
`-no-fcov`	Disable functional coverage collection.
`-write-sql`	Write coverage results to `sqlite3` database. This option is no longer necessary in VeriSim since the default output is the `sqlite3` database.
`-limit-enum-bins`	Limit number of automatic bins created for enumerated types.
`-wildcard-limit n`	Set limit on number of ranges created when expanding wildcard coverage bins (default: 1000)
`-fcov-save-empty-bins`	Records the empty bins that are artifacts of an ignore directive in the coverage database.

The `-no-fcov` option disables functional coverage collection at run time. When given, no functional coverage is written to the output `sqlite3` database (normally "verisim.db") when simulation ends. Also, covergroups sensitive to an event will not be activated during the run, which should save some run time. However, the covergroups and coverpoints are partially created, so that user code can naively call coverage methods without immediately failing on a null reference. Care should be taken not to rely on coverage computation or side-effects from covering expressions (e.g. function calls) which may alter a run without coverage.

Functional coverage is collected by default. We believe that all of the supported SV-LRM-defined language features ought to be enabled and working without requiring special command line switches.

The coverage option `auto_bin_max` is specified in the SV-LRM to limit the number of automatic bins created for a coverpoint, but is specifically not applicable to bins of enumeration type. The option `-limit-enum-bins` has been added to override the SV-LRM compliant behavior, and to subject bins of enumerated type to the same limit.

By default, wildcard bins are processed by a dedicated 4-state wildcard engine that uses bit masking to determine if a sample input matches a wildcard. However, under certain circumstances, a wildcard bin must be expanded into an equivalent set of range bins. e.g. `{ 4'b11xx }` can be expanded into `{ [12:15] }`. This expansion may occur under the following conditions:

- The coverpoint requires multiple bins, so different matches may go into different bins.
- There is a multiple-bin `default`, which requires determination of which values do not hit any other bin.

However, some wildcard definitions are not practical to expand.
e.g. `{ 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx0 }`. This expands into all two-billion even numbers, and will likely not complete at run time due to memory exhaust. The `-wildcard-limit` option is used to set a limit on wildcard bin expansion. If the limit is encountered, a warning will be printed and the expansion will be incomplete.

Controlling Code Coverage

VeriSim supports three types of code coverage:

- Line/block coverage: indicates which statements in the code base have been executed.
- Toggle coverage: indicates which integer-valued nets/variables have changed state.
- Focused expression coverage: indicates which conditions that lead to a boolean-valued expression have been encountered. The focused expression model is used.

Line/block and expression coverage requires that the code base be instrumented at compile time. Toggle coverage requires the option at runtime.

Option	Description

Option	Description
`-code-cov type[:type...]`	Instrument or enable for code coverage type = b(lock), e(xpression), t(oggle) or a(ll)
`-expr-cov-row-limit n`	Set row limit for focused expression coverage
`-code-cov-scope-specs`	Limit code coverage of enabled types to a specific scope using a specification file.

Line/block Coverage

Line coverage requires instrumentation at compile time. Performance is reduced when line coverage is enabled.

The database written at the end of the simulation contains sufficient information to produce either a line coverage or block coverage report.

Toggle Coverage

Toggle coverage requires that VPI callback permission (`+acc+b`) be set on any net or variable whose toggles are to be captured. Toggle coverage itself is a run-time option, much like waveform dump.

The toggle coverage model counts transitions from `0` to `1` and `1` to `0`. Transitions to/from `z` or `x` are not counted.

Expression Coverage

Expression coverage requires instrumentation at compile time. Performance is reduced when expression coverage is enabled.

VeriSim uses a focused expression coverage model. For each boolean expression being covered, the minimal set of input conditions required to exercise all paths through the expression is computed. It may not be possible to cover all paths through an expression if one input is used more than once in an expression, and used in conflicting ways that impede either control or observability.

The result of analyzing each expression for focused expression coverage is a truth table. Every time an input changes, the runtime must search all rows in the truth table and increment the counter corresponding to the matching row. Large truth tables both take up a lot of memory and consume a lot of run time doing the search. The `-expr-cov-row-limit` option can be used to limit which expressions are covered; expressions requiring larger truth tables will not be covered. Typically these expressions are used in datapaths, and not control logic.

Input Filename Examples

Each input filename is decomposed into one or more extensions, each starting with the `.` character. Both the primary and secondary extensions, if they exist, can be used to indicate information about the file contents.

- The primary extension can be used to indicate that a file is compressed and to identify the type of compression used.
- In the absence of compression, VeriSim uses the primary extension to decide which language specification to apply to the file.
- When compression is present, VeriSim uses the secondary extension to decide which language specification to apply.
- By default, VeriSim interprets the extension `.v` as Verilog and `.sv` as SystemVerilog. All other extensions are unknown and the file will be interpreted as Verilog.
- To customize the list of recognized extensions, use the `-sysvlog-ext` and `-vlog-ext` command line options. For more information see the Common Options section. Including the `.` character in each extension is optional.

DVlcom uses the same methods of determining Verilog and SystemVerilog when parsing files.

DVhcom is only capable of parsing VHDL files so there is no need to distinguish the extensions.

Filename	Options	Assumed Language	Compression
`test.sv`		SystemVerilog	
`test.sv.gz`		SystemVerilog	gzip
`test.sv.bz2`		SystemVerilog	bzip2
`test.sv.tmp.gz`		Verilog (default)	gzip
`test.SV`		Verilog (default)	
`test.SV`	`-sysvlog-ext .SV`	SystemVerilog	
`test.SV.gz`	`-sysvlog-ext .SV`	SystemVerilog	gzip
`test`		Verilog (default)	
`test.svh` `test.svp`	`-sysvlog-ext svh:svp`	SystemVerilog	
`test.vh` `test.vp`	`-vlog-ext vh:vp`	Verilog	
`test.vh.gz` `test.vp`	`-vlog-ext vh:vp`	Verilog	gzip or none

Compile Once, Run Multiple Times

When invoked with no special options, VeriSim will compile a SystemVerilog program and then immediately execute it. The SystemVerilog code is first compiled into a shared library, called the "image". The image is then loaded into memory and executed in conjuction with a run-time event scheduler.

For VHDL VeriSim can only combine pre-compiled libraries to generate the image.

It is often useful to compile a single image, and then use the image to run multiple tests. This can be done using the following options on VeriSim:

Option	Description
`-genimage name`	Generate a compiled image for later execution; do not run anything.
`-image name`	Run the previously compiled image.
`-work path`	Set path where VeriSim will create a directory with temporary files and images.

If a single image is used to run multiple tests, then different simulation outcomes can be achieved by doing one or more of the following:

- Use command-line plusarg options, and `$value$plusargs` to affect control flow. The UVM `+UVM_TESTNAME=...` is a good example of this.
- Use different random seeds, set with `-sv_seed`.
- Use a PLI or DPI library, which can be named using the `-sv_lib` or `-pli_lib` option. Alternatively, use `LD_LIBRARY_PATH` to control how a library resolves.

Selectively Enabling Top-Level Modules

Modern verification methodologies such as UVM allow the user to compile all tests cases into a single image, and then select a test at run time. An older methodology is to compile the design and testbench along with a testcase module, and then run it. A fresh compile is required to run a different test case, which can be time consuming.

As an alternative, VeriSim allows multiple testcase modules to be compiled alongside a common testbench and design, using a single compile. At run time, one or more of these testcase modules can be enabled.

Option	Description
`-compile-top name(s)`	Compile one or more modules as a run-time selectable top-level instance.
`-run-top name(s)`	Enable top-level instances previously compiled.

These options work in conjunction with `-top`. If either `-top` or `-compile-top` is given, then:

- All modules listed after `-top` are unconditionally marked as top-level instances, and will always appear in the design.
- All modules listed after `-compile-top` are marked as run-time selectable top-level instances. These are compiled, but will appear in the design only if enabled.

If neither `-top` nor `-compile-top` is given then top-level modules are determined as required in the SV-LRM.

Anything compiled with `-compile-top` must still adhere to Verilog or VHDL rules:

- Modules must not have the same name as modules compiled with `-top`.
- Hierarchical references from hierarchy under a `-top` module to hierarchy under a `-compile-top` module are permitted, but will abort at run time unless the target hierarchy is enabled.

At run time, any module compiled with `-compile-top` can be “turned on” with `-run-top`.

Using Libraries and Configurations

Options

Option	Description
`-lib <name>`	Set default library for analyzed items
`-libmap <name>`	Specify library mapping file
`-L <name>`	For elaboration phase only. Specifies compiled libraries to look for cells in. Library search order defined by order -L options appear on command line

Organizing the Design Into Libraries

VeriSim uses the “libmap” functionality described in section 33.3.1 of the SV-LRM.

The library map, specified with `-libmap` (SystemVerilog only) contains rules that assign each analyzed module/cell to a library based on its location in the filesystem. If no rule matches, then the analyzed module is placed into the library indicated by the `-lib` command line option. In the absence of any rule or `-lib`, the work library will be used.

The `-incdir` directive in the lib map is ignored, as the SV-LRM does not specify what it does. If a module appears in an included file, the top-level source file (that did the include) is used as the reference point for looking it up in the lib map.

Some simulation tools process the HDL in a two-step process: first, the HDL source code is analyzed to create one or more "design libraries". Secondly, an elaboration step creates the final simulation image from these design libraries. To ease the transition for users of these tools, the VeriSim simulation system now supports this model. The DVlcom (for Verilog and SystemVerilog) and DVhcom (for VHDL) tools are first used to create the design libraries, after which VeriSim can be invoked.

The pre-processor and analysis related options defined for VeriSim also apply to DVlcom and DVhcom. DVlcom also support the same `-libmap` outlined above. The `-help` option from either tool will give the full list.

For example, the following will create a library A with the contents of files cell1.sv, cell2.sv and cell3.sv, and create library B with the contents of cell3b.sv.

```
dvlcom a/cell1.sv a/cell2.sv -lib A -l first.log +define+TEST_LIMIT=50
dvlcom a/cell3.sv -lib A -shared-unit-scopes -l second.log
dvlcom b/cell3b.sv -lib B -timescale 1ns/1ns
```

The same approach would apply to DVhcom for VHDL.

Using Configurations and Libraries

The top-level instance of a library-aware design can be specified using `-top libname.cellname`.

If the top level is a configuration, then it can also be specified using `-top configname`.

In the special case of a configuration having the same name as a cell in some library, one may specify `-top configname:config`.

The elaboration-time option `-L libname` can be used to specify libraries and their search order to look for cells in.

For example, if libraries tb, A and B exist in the verisim work directory, the following will elaborate and generate the design image for the top level module tb, assuming all cells required exist in the three libraries. Libraries are searched in the order tb,A,B,work.

```
verisim -top tb.top -L tb -L A -L B -genimage image
```

Legacy Behavior

If either `-lib` or `-libmap` is specified on the command line, then VeriSim operates in "library-aware" mode. In this mode, if a module is added to a library and has the same name as a previously analyzed module, then the most recently analyzed module will supersede the original.

If neither `-lib` nor `-libmap` is specified on the command line, then VerSim operates in "legacy" mode. In this mode, subsequent definitions of a module are ignored; the first definition wins. This is different from library-aware behavior.

You may use `-v` and/or `-y` option in "library-aware" mode. However, instead of using `-v` or `-y` option, it is recommended to analyze all files on the command line, use the lib map and/or DVlcom/DVhcom to organize them into libraries, and then specify configurations or -L to select cells to be instantiated in the design.

Stages Of A Simulation

Running a simulation in VeriSim is done in three distinct stages

Analyze (**Compile**): This is the process in which each design file is parsed and interpreted based on the particular language syntax. The resulting design components are saved for elaboration. Any syntax-related or other problems with the files are reported here.

Elaborate: Based on the desired top-level component(s) of a design, all other required components are taken and resolved into a final design image which is saved to disk. Any resolution issues (ie. missing components, incompatible references, etc..) are reported here.

Run: The final image is loaded by VeriSim, initialized and executed. The simulation will proceed until completion or a problem occurs. Non-static warnings and errors which can only be detected during the execution of the simulation are reported here.

The VeriSim tools allows users various control through the three stages if needed.

Running A Verilog (System Verilog) Simulation

The VeriSim tools allow control through the three stages of a simulation.

Single Invocation

To run a VeriSim simulation from a single command line can be done as follows

```
verisim <design files> ...
```

When executed as above, VeriSim will proceed through all three stages: processing all design files (**Compliation**), produce an image (**Elaborate**), and executing it until completion (**Run**).

The advantage here is the simplicity of the command.

Compile Once, Run Many

For larger designs the creation of the image may take some time, and if the design is intended to be run several times (ie. with different inputs or seed value) then the image can be created separately. See also Compile Once, Run Multiple Times.

The following command will parse all the design files (**Analyze**) and produce the image (**Elaborate**), but not run it.

```
verisim -genimage image <design files> ...
```

The following command will run the image (**Run**).

```
verisim -image image ...
```

The advantage of this flow is the time and effort saved in creating an image only once for multiple tests.

Building A Library

For some projects a single collection of files can be used in different top-level designs, depending how they are incorporated together.

For such a case it is possible to use the DVlcom tool to parse all the files and add the resulting design components to a library that can later used to create different types of images.

The following command will parse the given design files and populate the library (**Analyze**).

Note: The design components are only saved to a library on disk when using DVlcom, if processed by VeriSim it is kept in memory only for the existing process.

```
dvlcom <design file> ...
```

DVlcom can be run on all files together or each file independently - depending on what command line options are necessary for each file. Design components can be added to the same library or distributed accross multiple libraries. All libraries needed for a given simulation must be located/linked into a single VeriSim workspace. For linking a library refer to Managing Libraries.

When a file which is being analyzed contains an external reference, the reference must be found within one of the local libraries. If that reference is to a pre-compiled library, it must already be linked (see Managing Libraries). Lowest level design components must be analyzed first. The higher level components follow. This ordering is important so that the all references are properly resolved.

The default library is `work`, but can be replaced with any preferable name.

The following command will elaborate a design based on the given top-level name (**Elaborate**) and run the simulation (**Run**).

```
verisim -top <library>.<module name> ...
```

If desired the building of the image and the running of the simulation can also be separated. Compile Once, Run Multiple Times.

```
verisim -genimage image -top <library>.<module name> ...
```

And

```
verisim -image image ...
```

The advantage to this flow is that ability to parse each design file only once in order to build a variety of designs for simulation.

For management and inspection of the design component library you can use the DLib tool as follows.

To list all libraries

```
dlib ls
```

To list design components within a library

```
dlib ls -lib <library name>
```

For more information please refer to Managing Libraries.

Running A VHDL Simulation (Beta)

The VeriSim tools allow control through the three stages of a simulation.

A necessary part of any VHDL simulation in VeriSim is to provide the timescale during the **Elaboration** stage. For the purposes of these examples assume a timescale of `1ns/1ps`. For more information on `-timescale` please refer to Common Options.

Provided IEEE Standard Libraries

If your designs will use any of the IEEE standards for VHDL, they have been provided pre-compiled for VeriSim and made available in the path stored in the `STD_LIBS` environment variable.

The included libraries are:

- ieee87
- ieee93
- ieee08

To include them in your flow you will need to use DLib to link them to your workspace (see Managing Libraries).

```
dlib map -lib ieee ${STD_LIBS}/<library>
```

Building A Library

Due to the nature of VHDL, the **Analyze** stage MUST be performed as a separate stage. To do this the DVhcom tool can be used to parse all the files and add the resulting design components to a library that can later be used to create different types of images.

The following command will parse the given design files and populate the library (**Analyze**).

```
dvhcom <design files> ...
```

DVhcom can be run on all files together or each file independantly - depending on what command line options are necessary for each file. Design components can be added to the same library or distributed accross multiple libraries. All libraries needed for a given simulation must be located/linked into a single VeriSim workspace. For linking a library refer to Managing Libraries.

When a file which is being analyzed contains an external reference, the reference must be found within one of the local libraries. If that reference is to a pre-compiled library, it must already be linked (see previous section). Lowest level design components must be analyzed first. The higher level components follow. This ordering is important so that the all references are properly resolved.

The default library is `work`, but can be replaced with any preferable name.

The following command will elaborate a design based on a given top-level name (**Elaborate**) and run the simulation (**Run**).

```
verisim -timescale 1ns/1ps -top <library>.<module name> ...
```

If desired the building of the image and the running of the simulation can also be separated. See also Compile Once, Run Multiple Times.

```
verisim -genimage image -top <library>.<module name> ...
```

And

```
verisim -timescale 1ns/1ps -image image ...
```

The advantage of this flow is the ability to parse each design file only once in order to build a variety of designs for simulation.

For management and inspection of the design component library you can use the DLib tool as follows.

To list all libraries

```
dlib ls
```

To list design components within a library

```
dlib ls -lib <library name>
```

For more information please refer to Managing Libraries

Mixed VHDL/Verilog Designs

This feature should be considered alpha quality.

Instantiating Verilog Inside VHDL

You must declare and instantiate a component:

```
// leaf.v
module leaf #(parameter W=8) (input clk, input [W-1:0] d, output reg q);
...
endmodule

-- top.vhdl
component leaf
generic(
    W: integer
);
port(
    clk: in std_logic;
    d: in std_logic_vector(W-1 downto 0);
    q: out std_logic
);
end component

u1: leaf
    generic map(
        W => 4
    )
    port map(
        clk => clk,
        d => d,
        q => q
    );
```

You may encounter a 3-rd party IP that doesn't adhere to the above requirement. As a workaround, try to analyze its Verilog with `-gen-proto` option.

To compile: you may pre-analyze the Verilog with DVLcom:

```
dvlcom leaf.v
dvhcom top.vhdl
verisim -top work.top
```

Alternatively, analyze all Verilog at the time of elaboration:

```
dvhcom top.vhdl
verisim leaf.v -top work.top
```

You cannot instantiate a Verilog object directly as a VHDL entity. You may reference a Verilog object as a VHDL entity in a component configuration or configuration specification. However, incremental binding (both port map and generic map) are not supported.

Instantiating VHDL Inside Verilog

You may refer to a VHDL item using the full library path as a Verilog extended name:

```
-- leaf.vhdl
entity leaf is
generic(
    W : integer
);
port(
    clk: in std_logic;
    d: in std_logic_vector(W-1 downto 0);
    q: out std_logic
);
end leaf;
```

```
// top.sv
\work.leaf #(4) dut(
    .clk(clk),
    .d(d),
    .q(q));
```

The VHDL items must be analyzed with DVhcom, after which the Verilog can be compiled and elaborated:

```
dvhcom leaf.vhdl
verisim top.sv -lib work
```

Binding Verilog Items Into a VHDL Design

A Verilog item can be bound into a VHDL design. The syntax is identical to binding a Verilog item into a Verilog design.

Supported Type Conversions

The following data type conversions are supported for generic/parameter mapping and port connections:

Category	Verilog	VHDL	Notes
Integral	Any integral/bit/reg type	Any integral or enumerati	

Category	Verilog	VHDL on type	Notes
Floating	real/shortreal/realtime	Any floating type	
Single-bit	reg/bit	bit/std_logic/std_ulogic	
Vector	packed bit[:], reg [:]	bit_vector/std_logic_vector/std_ulogic_vector/unsigned	Widths must match
Signed	bit signed [:], reg signed [:]	signed	Widths must match
Stringlike	reg [:]	string	Bounds of Verilog type must match that of string for port connection
String	string	string	Supported for generic/parameter mapping only
Array	unpacked array	1D array	Number of elements must match; element type must be interoperable
Record	packed/unpacked struct	record	Members in declaration order must be interoperable

A VHDL string may be mapped to an untyped Verilog parameter, in which case the `reg[n*8-1:0]` representation will be used on the Verilog side.

Managing Libraries

The `dlib` utility is used to perform various maintenance tasks on libraries created with `dvlcom` or `dvhcom`.

Library Repository Versioning

Each numbered release of the VeriSim simulation tool set maintains a separate repository of items in your work area. More than one version may exist at any given time. This is useful if you need to switch between tool versions, e.g. to troubleshoot a problem.

However, this may be confusing, as the contents of the same library work area may appear different to different versions of the tools. To minimize this problem, the working revision is displayed in reports and messages generated by `dlib`.

It is not possible to examine a work area from version A of the toolset with version B of `dlib`, if A and B are different. Rather, you must use version A of `dlib`.

Listing Contents of Libraries and Work Area

Name	Description
`dlib ls`	Lists all libraries in work area.
`dlib -lib name ls`	Lists contents of a library.

This command is used to display the contents of a work area, or a library in the work area.

Without `-lib`, the names of all libraries are displayed, along with their mapping to a local or remote work area.

With `-lib`, the contents of a library are displayed. The following items are displayed for each object in the library:

- Name of the item
- Type of the item (entity, module, etc.)
- Time of last modification

Details about encrypted design elements are not listed. However, the report will indicate whether encrypted design elements are present.

Output Format

By default the output from the `ls` command is formatted into text-based tables.

The option of `-json` is provided to output information as JSON objects that can be easily parsed by enternal programs.

Mapping to Remote Work Areas

Name	Description
`dlib map -lib name path`	Creates new mapping.

It is possible to have a library of read-only components located in a common work area. `dlib -map` creates the mapping. The `path` argument must refer to a `verisim_work` directory, other than the local work area, and which must already exist.

Removing a Library or Mapping

Name	Description
`dlib rm -lib name`	Remove library or mapping.
`dlib rm -lib libname item`	Remove item from library.

The first form of the `dlib rm` command removes an entire library or mapping. If the library is mapped to a remote work area, only the mapping is removed, leaving the contents of the remote work area unaffected.

If the library is local, then the entire library is removed.

The second form of the command removes an item from a library. If the item is a SystemVerilog package/configuration that has the same name as another design element, then the `-config` or `-package` options can be used to disambiguate.

Secondary elements are also removed.

Dumping/Opening Waveforms

VeriSim supports VCD waveform dump.

Option	Description
`+acc[+rwcbfs]`	Generate support for waveform dump and VPI. This option must be given at compile time. Adding `+acc` defaults to `+acc+rwb`, for more information please review the secion “Using DPI and PLI”
`-acc-specs file`	Compile time option enabling one or more of [rwcbf] at a specific scope using a specification file. See Setting Options by Scope Using Specification File for details.
`-waves file.vcd`	Specifies file for VCD waveform dump. This option must be given at runtime.
`-waves file.vcd.gz`	Specifies file for VCD waveform dump while also compressing to GZIP format. This option must be

Option	Description
	given at runtime.
`-dump-start time`	Specifies time to start dump. Default: time 0.
`-dump-end time`	Specifies time to stop dump. Default: end of simulation.
`-dump-agg`	Enables dumping of aggregates (arrays and structs).
`-wave-scope-specs file`	Control waves to dump by scope using a specification file. See Setting Options by Scope Using Specification File for file details and examples.

For the `-dump-start` and `-dump-end` options, the time value may optionally have a time unit attached. If no time unit is specified, then the default time resolution of the simulation is assumed. Valid time units are: `fs`, `ps`, `ns`, `us`, `ms` and `s`. No spaces should appear between any digits and the time unit.

VeriSim also supports the standard `$dumpon/$dumpoff/$dumpvars` calls to dump specific time regions, or to limit the dump to specific signals.

The choice of database format is made depending on the dump file name. In all cases the standard dump system tasks are used.

VCD Dump

VCD dump is a supported target for transparent compression: if you dump to "file.vcd.gz" then VeriSim will automatically compress the VCD file using `gzip`. GTKWave can read such compressed VCD files directly.

VCD was invented around 1985, and does not support any advanced constructs. For this reason, VeriSim supports SystemVerilog and VHDL within the VCD format as best it can:

- Interfaces, program blocks, packages, etc. are dumped as if they were modules.
- By default, only integral, real values and events can be dumped. Enums and packed structs are dumped as their integral values. If `-dump-agg` is given, then elements of packed/unpacked arrays and structs are dumped as individual VCD signals.

Standard VCD supports the 4-value logic set `0`, `1`, `X`, and `Z`. Some VCD readers are able to read a non-standard file that can represent the additional VHDL 9-state values `L`, `H`, `W`, `U` and `-`. The `-dump-vcd-9s` option will cause VeriSim to write out such an enhanced VCD file. Without the option, the additional VHDL 9-state values are mapped to the standard values.

Examples:

- To dump waves in VeriSim do the following:

```
verisim +acc -waves <file.vcd>
```

- To view waves, you may use any waveform viewer that supports the above formats. For example, to view waves in GTKWave (http://gtkwave.sourceforge.net/), do the following:

```
gtkwave <file.vcd>
```

Using Verification Frameworks: UVM

UVM

VeriSim ships with UVM-1.1b, UVM-1.1d and UVM-1.2. All packages required some customization, in part because UVM actually does not compile properly if none of the major vendor's tools are used. Presumably none of the UVM maintainers had any way of testing this.

By default, VeriSim is configured to use UVM-1.2. If you have a testbench that requires UVM-1.1 then you must set the $UVM_HOME environment variable as follows:

```
export UVM_HOME=$DSIM_HOME/uvm-1.1b
```

VeriSim does not auto-compile UVM by default. You need to manually put the UVM code in your include file path, and compile the UVM package. This is typically done as follows:

```
verisim +incdir+$UVM_HOME/src $UVM_HOME/src/uvm_pkg.sv ...
```

Also take care to include `uvm_macros.svh` from any compilation unit that requires it.

To run, you need to load the UVM DPI library. The library has already been compiled; you merely need to load it as follows:

```
verisim -image image -sv_lib $UVM_HOME/src/dpi/libuvm_dpi.so +UVM_TESTN
AME=...
```

Additional Verification frameworks

For information about OVM and VMM, contact support@primarius-tech.com.

Using the DPI and PLI

The following only applies to SystemVerilog support.

VeriSim uses shared libraries for the DPI and PLI.

DPI

The following options pertain to the DPI:

Option	Description
`-sv_lib name`	Loads the named shared library at runtime. The shared library extension for your platform (e.g. ".so") is optional.
`-dpiheader ...`	Generate C/C++ DPI header file for exported items.

`-dpiheader` is a compile-time option. It is given the name of an include file to write out which contains prototypes for any exported SystemVerilog tasks/functions, as well as definitions for any structs that have C compatible layout.

`-sv_lib` is a runtime option only. This option may be given multiple times, to load more than one DPI library.

DPI Import

If SystemVerilog code declares a DPI import function, then the resulting image will reference the function, which ought to be resolved by loading a library (see above). Resolution failures are detected at the point where a DPI function is called to allow images to run without the DPI library, as long as they don't try to call the DPI functions.

DPI Export

If SystemVerilog code declares a DPI export function, then the resulting image will define the corresponding C function. This function will link properly with any DPI code provided, even if the call into the DPI export is made from a DPI import function. (Not all simulators support this smoothly!)

PLI

PLI support is evolving. The current support is sufficient to support licensing calls for specific partners, and fetch/deposit for UVM register model backdoor operations. Other applications may work, but this is not guaranteed. If your application does not work, use `-trace-vpi` to help find out why and contact Primarius support for assistance.

VeriSim aims to support the VPI interface. As of SystemVerilog, the tf and acc interfaces are deprecated. New code should not be using these. VeriSim supports a very limited set of tf routines, and does not support acc at all.

The following options pertain to the PLI:

Option	Description
`-pli_lib name`	Loads the named shared library at compile time and

Option	Description
	run time. The shared library extension for your platform (e.g. ".so") is optional. verisim will look in the shared library for the `vlog_startup_routines` array and will call the routines within the array. **Note**: The routines listed in the array should be declared as static to avoid conflicting with other bootstrap routines.
`+acc+[rwcbfs]`	Instruments the compiled code for VPI access. Using this option incurs a runtime penalty.
`-acc-specs file`	Allows fine-grained specification of VPI accessibility.

Unlike DPI, PLI libraries must be loaded in both at compile time and run time. The set of system calls defined by a PLI library is given in a table in the library, which also contains pointers to functions used at compile time to validate arguments (`checktf,sizetf`).

Specifying VPI Object Accessibility

Each variable/net in the design has a VPI accessibility, which consists of one or more of the following permissions:

Name	Description
r	The object's value can be read, or attributes can be queried.
w	The object's value can be written, but not forced.
c	The object's connectivity can be extracted (`vpiLowConn`, `vpiHighConn`)
b	Callbacks can be set on the object. This permission is required to allow waveform dump of the object.
f	The object can be forced.
s	(Whole design only) Enables inclusion of the PLI simulation regions (Pre/PostNBA, Pre/PostReNBA used for `cbReadWriteSynch` or `cbReadOnlySynch`). Enabling these regions will result in a slight loss of performance.

The `+acc+[rwcbfs]` option is used to specify the default accessibility for the whole design. Default is no access. A value of `+acc` is equivalent to `+acc+rwb`.

For finer-grain control, one may specify the accessibility of any net in an external file, using the `-acc-specs` option. The file is of a common format described in Setting Options by Scope Using Specification File. Each line in the `-acc-specs` file is either a module line, a path line, or a net line.

The key word `module` indicates a module line of the form

```
module name +mode
```

where `name` is the name of a module (not an instance) and `mode` is one or more of the permissions `rwcbf`. The module line specifies the default permissions of all variables/nets in the module.

The key word `path` indicates a path line of the form

```
signal path +mode
```

where `path` is indicates a scope starting point and `mode` is one or more of the permissions `rwcbf`. A net(s)/variable(s) in the path's scope will have the mode applied.

The keyword `signal` indicates a net line of the form

```
signal name +mode
```

where `name` is the name of a net/variable, and `mode` is one or more of the permissions `rwcbf`.

The `#` or `//` strings may be used to introduce comments.

Impact of Specifying Permissions

The VPI permissions allow VeriSim to optimize the design and choose implementations without impacting VPI requirements. Absent any VPI access requirements, VeriSim's optimizers may transform the design in ways that would result in incorrect behavior.

Read Permission

Consider the code:

```
reg [3:0] foo = 10;
always @(x) y = {foo,x};
```

If we knew that there was no requirement to read `foo`, then we can change the code to:

```
always @(x) y = {4'd10,x};
```

However, if there were a VPI reader, then after the optimization, the VPI code would fail due to optimization removing the declaration of `foo`.

Write Permission

Consider the code:

```
assign a = b & c;
assign d = a | e;
```

If we knew that nothing outside the code could possibly write to a, then we can change the code to:

```
always @(b or c or e) begin
  a = b & c;
  d = a | e;
end
```

Since two processes have been reduced to one, we can amortize the overhead of block scheduling. Assuming that the computations are cheap (e.g. the variables are single-bit `reg`) this is a big savings. However, this transform fails if VPI writes to `a`: since the transformed block is no longer sensitive to `a`, a write to `a` will never provoke an update of `d`. Adding `+w` permission to `a` will disable this optimization and result in correct behavior, but lower performance.

More importantly, write permission is *required* if `vpi_put_value()` will be called with a delay, so that the variable can be instrumented with the data structures required to implement the delay mechanism.

Callback permission

When `+b` permission is specified for a net/variable, then every time the variable is updated, the new value is compared to the previous value, and the value change callback is invoked if there is a change. The code to compare and execute callback is compiled into every expression that may update the variable, and the compare logic may be as costly as the update to the variable itself. Although VeriSim also has to do comparisons on variables for which other processes are sensitive, VeriSim tries to generate the comparison code only when necessary.

When the `+b` permission is specified for a scope containing a concurrent or immediate assertion, then every time an assertion evaluation starts or ends, VeriSim checks for registered callbacks to invoke. These checks add extra overhead to assertion evaluation, even if no callbacks are registered.

Force permission

When a variable is forced, the compiled code tracks whether or not the variable is currently forced. The force state of a variable is checked prior to each write, so that the write is “not done” when the variable is forced. Similar to callbacks, this adds considerable overhead to each write. Normally, VeriSim will instrument variables for forcing only if a Verilog `force` statement could potentially reference the variable - this property can be statically computed at compile time. However, with `+f`, VeriSim must instrument every single variable having this permission, which again is quite costly.

The situation is worse for nets. Individual bits of a net can be forced, so a write to a multi-bit net may result in some bits (the unforced bits) changing value, and other bits not changing. Net forcing is actually done using an interpreted engine, which also handles `inout` ports, strength logic modelling, tran solving and net delays.

Again, this treatment is usually reserved only for those nets that can be determined to be forced at compile time, but `+f` will force this treatment for every affected net.

Performance loss with a global `+acc+f` is so bad that it is strongly recommended to use an `-acc-specs` file to apply force permission only to those nets that specifically require it.

Tracing/debugging DPI/PLI/VPI

Name	Description
`-debug-vpi`	Log all failed/invalid VPI calls.
`-trace-vpi`	Log all VPI calls, successful or otherwise. Also log actions associated with PLI library loading.
`-trace-dpi`	Log all actions associated with DPI library loading.

Building Shared Libraries

This is platform-specific.

Linux x86, 32-bit

To compile:

```
gcc -c -o file.o file.c -I$DSIM_HOME/include
```

To create shared library:

```
gcc -shared -o mylib.so file1.o file2.o ...
```

Linux x86, 64-bit

To compile:

```
gcc -c -fPIC -o file.o file.c -I$DSIM_HOME/include
```

To create shared library:

```
gcc -shared -o mylib.so file1.o file2.o ...
```

Using VHPI

Currently there is no support for VHPI.

Using Compressed Files

VeriSim supports “transparent” compression: in certain contexts it will compress/decompress files on the fly.

The following types of compression are supported:

Suffix	Compression program
`.gz`	gzip
`.bz2`	bzip2
`.xz`	xz
`.Z`	compress

The compression/decompression is performed by launching the compression program in a child process. On a multi-core machine, the compression or decompression happens in parallel with model execution.

This requires that the compression program be installed. Due to patent issues, `compress` is typically not installed on many systems. The patents have expired, but `compress` is still rarely found since it has the worst performance of all the compressors.

Rules for Reading Files

If the filename being opened has a primary extension denoting compression, then the file is opened and decompressed.

If the filename being opened does not have a primary extension denoting compression and does not exist, but a file of the same name, plus an applicable extension exists, then that file will be opened and decompressed. e.g. if `gates.vg` does not exist, but `gates.vg.gz` does, then the latter will be opened and decompressed.

If the primary extension denotes a compression type, the secondary extension can be used to identify the intended language of the file. If no compression is specified, then the primary extension can be used to identify the intended language. Although the intended language can be explicitly specified using the Common Options.

See Input Filename Examples to see different combinations of extensions to indicate file type as well as compression types.

Rules for Writing Files

If the filename being opened has a primary extension denoting compression, then the file is opened and compressed.

If the filename being opened does not have a primary extension denoting compression, then the file is opened and written uncompressed. The existence of file(s) with a similar name does not alter behavior.

Caveats

If a file being opened for read does not exist, but there are multiple compressed versions using different compressors, then it is not defined which one will be opened.

Files opened using `$fopen` or `$open` using transparent compression will not support `$fseek` - you cannot seek on a compressed file, at least not easily.

Supported Contexts

Transparent compression is supported for the following operations:

- Compile: reading in source files: either off the command line, or using include, or during library (`-v`/`-y`) scans
- VCD waveform dump. Note that GTKWave can read in `*.vcd.gz` files directly.
- `$readmem`/`$writemem`
- System calls (`$open`/`$fopen`)
- Reading SDF files.

Lexical Conventions

Treatment of Non-ASCII Characters

VHDL is defined to use the 8-bit character set ISO8859-1. Verilog and SystemVerilog do not specify the character set at all, although “ASCII” is mentioned, as well as “printable ASCII” in the range 33-126 decimal for use in escaped identifiers.

Strictly speaking, if 7-bit ASCII is to be assumed, then characters outside that range must be rejected, even in comments. However, we have seen code with non-ASCII characters (e.g. the copyright symbol) in comments.

VeriSim accepts any 8-bit character sequence inside a comment. However, VeriSim treats the input as a sequence of 8-bit characters, one character per byte. It has no knowledge of UTF-8, UTF-16, byte order marks (BOM), etc. Therefore, only encodings that cannot spuriously produce a comment termination are reliable. UTF-8 has this property.

VeriSim accepts only printable 7-bit ASCII characters in escaped identifiers.

Verilog Design and Verification Building Blocks

Compilation Unit Scope

For purposes of parsing and initial semantic analysis, VeriSim treats each file listed on the command line as a separate compilation unit, which also includes anything included from the file. If a top-level construct is unterminated at the end of a top-level file then a parse error will be reported.

Clause SV-LRM 3.12.1 requires that tools support both possible models for compilation-unit scope: one scope for the entire design, and one scope per compilation unit. Previous versions of VeriSim supported only the first model.

Starting with 20201123 relase, both models are supported, selected by the `-shared-unit-scope` or `-separate-unit-scopes` respectively.

Given that the treatment of compilation-unit scope can vary from tool to tool, we recommend that compilation-unit scope not be used. All declarations outside design elements (modules, interfaces, etc.) should be put in packages.

Name Spaces

The **definitions name space** contains definitions for modules, interfaces, program blocks and primitives.

The **module name space** contains definitions of items that are encountered inside a module. In particular, it contains typedefs.

Generally speaking, two declarations cannot have the same name if they are to appear in the same name space. However, two declarations can have the same name if they appear in different name spaces. For example, it is legal to name a module the same as a variable. References to the module in a module instantiation are not ambiguous - such references are always to the definitions name space, whereas references to variables are always to the module and block name spaces.

There is an exception: it is possible to give an interface the same name as a typedef. If this name is used in a declaration of an interface port (or array thereof), it is ambiguous as to whether the interface or typedef is intended:

```
interface foo;
endinterface

typedef reg [1:0] foo;

module mymod(foo f);
...
endmodule
```

In this case, VeriSim will favor the module name space (or compilation-unit scope) and resolve the typedef first. This also means that VeriSim may not recognize interface names in contexts where they are illegal - the compiler will simply not check the definitions name space in such cases.

Time Value Rounding

VeriSim will warn if the value of a time literal is given with greater precision than the time precision in effect where it appears.

Scheduling Semantics

During development the following code was found to be problematic:

```
reg reset;

initial begin
  reset = 1'b0;
  #100;
  reset = 1'b1;
  ...
end

always @(posedge clk or negedge reset) begin
  ...
end
```

The code in question expected to see a reset at time zero, before the first clock edge.

Prior to the event simulation starting, the initial value of `reset` is 1'bx. Two processes are then scheduled at time zero:

- The initial block will set `reset` to 1'b0" at time zero.
- The always block will execute its event blocking statement, and block on either `clk` or `reset`.

It is not defined which happens first. If the initial block runs first, then the always block will not see a reset at time zero. On the other hand, if the always block runs first, then it will.

VeriSim now identifies always blocks that immediately block waiting for events, and schedules them to run at time zero before initial blocks and always blocks for which this determination cannot be made. For maximum portability, code like this should be avoided.

Delay Model for Module Paths

SV-LRM section 30.2 defines distributed delays and module delays as follows and this document adheres to these definitions:

1. *Distributed delays*, which specify the time it takes events to propagate through gates and nets inside the module (see 28.16)

2. *Module path delays*, which describe the time it takes an event at a source (input port or inout port) to propagate to a destination (output port or inout port)

This section covers Module Path Delays, followed by a section on Operational Modes which can be targeted with an appropriate setting of command line options.

Basic Module Path Delay Model

Given a module with a specify block which defines one or more module paths to an output, this section discusses the delay module used to determine the transitions on that output.

In the basic case, assuming no invocation options are given, the reject limit = error limit = delay of trailing edge. In this case the following approach is taken:

1. The previously scheduled value of the output is tracked throughout. During initialization it is set to the initial value of the output for variables or x for nets.
2. When a new value for the output is scheduled (eg by: continuous assignment, gate output, udp output) the transition type (one of 12 possibilities {0,1,x,z} ? {0,1,x,z}) is determined by comparing the previously scheduled value and the newly scheduled value.
3. Based on the rules set out in SV-LRM 1800-2012 section 30.5 the active path is selected and the delay is calculated.
4. The transition time of the input of the active path, plus the delay is the intended transition time for the output, ie the schedule time.
5. The output schedule is then analyzed and adjusted for the new value of the output.
    1. If there is another entry pending in the schedule and the new output?s schedule time is prior to that we have a negative pulse, see the Negative Pulse Handling section.
    2. If the output?s schedule time is at or after the pending item the pulse width is calculated as the difference of the scheduled times.
        1. If the pulse width < reject limit the pulse is rejected. The Pending transition is removed from the schedule and if the new value of the output != the starting value of the previously pending transition a new transition is added from the starting value of the pending transition to the new output value at the output?s schedule time.
        2. If the pulse width >= reject limit a new entry is made in the output schedule at the output?s schedule time.

Fine Tuning Reject and Error Limits

The default reject and error limit can be modified in various ways including (See command line option documentation for details):

- -pulse_e, -pulse_r percent

    - percent modifiers applied to the transition delay to get the error and reject limits respectively
- -pathpulse with $PATHPULSE specparams in the specify blocks
    - enables the $PATHPULSE settings in the specify blocks, if -pathpulse is not specified $PATHPULSE specparams are ignored.
    - these take precedence over the default limits even when pulse_e and pulse_r are specified.
- +transport_path_delays - changes the default reject and error limit to 0
    - Note that using transport delay allows small pulses to propagate and will impact simulator performance negatively particularly in large combinational logic.

Taken together with the appropriate precedence these various settings prepare the reject and error limits that are used to fine tune the pulse filtering.

The pulse width calculated in 5.2 of the Basic Module Path Delay Model is handled slightly differently in the presence of rejectLimit != error limit:

- if pulse width < reject limit it is handled as described in 5.2.1
- if rejectLimit <= pulse width < error limit then the output schedule is adjusted to add a transition to x and back to the final new output value.
    - if pulse_e-ondetect is specified the transition to x is actioned immediately, the previously pending transition is removed from the schedule and a new transition from x to the new output value is scheduled at the output?s schedule time.
    - Otherwise (by default or if pulse_e-onevent is specified) the previously pending transition is adjusted to be a transition to x and a new transition from x to the new output value is scheduled at the output?s schedule time.

Negative Pulse Handling

Regardless of how the pulse limits are set, negative pulses are detected as described in step 5.1 of the Basic Module Path Delay Model.

The in the absence of pulse style settings within the specify block and command line option -showcancelled, the negative pulse is rejected.

Otherwise the pulse style showcancelled is in effect and the output schedule will be adjusted with a transition to x and from x to the new output value.

- If pulse_e-ondetect is specified the transition to x is actioned immediately, the previously pending transition is removed from the schedule and a new transition from x to the new output value is scheduled at the time of the previously scheduled transition.

- Otherwise (by default or if pulse_e-onevent is specified) a new transition to x is made at the output trailing edge?s schedule time, the previously pending transition is removed from the schedule and a new transition from x to the new output value is scheduled at the time of the previously scheduled transition.

Operational Modes

This section discusses how the Basic Module Path Delay Model can be used with various options to achieve various operational modes. Please note these are not global modes and only apply the delays controlled with module paths within specify blocks.

Pure inertial (default mode)

This is the default mode with neither -pathpulse nor +transport_path_delay options set.

The $PATHPULSE settings in the specify blocks are ignored and the error and reject limits are set equal to the delay associated with the trailing edge of a pulse. Since the error limit == reject limit the only time the result will go to x as a result of pulse handling is when a negative pulse occurs with showcancelled enabled.

Hybrid Delay Model

This mode is enabled by one or more of the following options:

- -pathpulse option which will enable the $PATHPULSE settings.
- -pulse_e/-pulse_r percent options adjusting the default error and reject limits away from the defaults.

Note if +transport_path_delay is set the default error and reject limit will be 0 instead of the trailing edge?s transition delay.

The error and reject limits are set based on precedence rules and the settings of $PATHPULSE -pulse_e and -pulse_r values. If the error limit is not provided but the reject limit is, then the error limit will be set equal to the reject limit. When neither is specified the delay of the trailing edge will be used for both limits.

Pure Transport Mode (all pulses transmitted)

To enter this mode add the +transport_path_delay and not the -pathpulse options nor the pulse_e/pulse_r options.

The $PATHPULSE settings in the specify blocks are ignored and the error and reject limits are set equal to 0. Since the error limit == reject limit == 0 the only time the result will go to x as a result of pulse handling is with showcancelled enabled.

Data Types

Representation of Reals

1800-2012 specifies that `real` and `shortreal` are the same as C language `double` and `float` respectively. In a footnote, it “clarifies” that these types are represented using IEEE-754. These statements are potentially contradictory, as there is computer hardware for which C compilers exist, but does not support IEEE-754.

Fortunately, all of the platforms on which VeriSim exists or is contemplated *do* support IEEE-754 in hardware, and have the C ABI specify that IEEE-754 is indeed used.

chandle Data Type

The internal representation of a `chandle` is equivalent to a raw (void) pointer on all platforms.

IEEE 1735 Encryption In Verilog

`dvlencrypt` is an encryption tool developed by Primarius to protect Verilog or SystemVerilog code using IEEE1735 encryption. The encryption strength is similar to that provided by SSL, and is considered to be the strongest publicly available.

Usage:

```
dvlencrypt <input_plain.sv> -o <output_encrypted.sv>
```

- **IMPORTANT:** Always hand-inspect the output to ensure that all sections intended to be encrypted are actually encrypted. If you fail to specify the parameters for a section properly, it may not be encrypted.
- **Requirements:**
  - OpenSSL must be installed
  - Every input file requires an encryption pragma protect pattern marking up the secret sections.
  - Every instance within the file that needs to be encrypted, requires the encryption pragma protect pattern of each of the secret sections.

Example

Take this SV code marked with the required pragma protect patterns for the secret sections.

input_plain.sv:

```
module top;
`pragma protect begin
    initial $display("this is secret stuff 1");
```

```
`pragma protect end

initial $display("this is not secret stuff");

`pragma protect begin
    initial $display("this is secret stuff 2");
`pragma protect end
endmodule
```

Run `dvlencrypt`:

```
dvlencrypt input_plain.sv -o output_encrypted.sv
```

Output from `dvlencrypt` is shown below.

output_encrypted.sv:

```
module top;

`pragma protect begin_protected
`pragma protect version=1
`pragma protect encrypt_agent="dvlencrypt"
`pragma protect encrypt_agent_info="Primarius Technologies Co., Ltd."
`pragma protect data_method="aes256-cbc"
`pragma protect key_keyowner="Primarius Technologies Co., Ltd."
`pragma protect key_keyname="VeriSim"
`pragma protect key_method="rsa"
`pragma protect key_block
PrLI34U6UgDhQ99aT2gg1HmJt70gd1KX2skLHYK6RbbzBkzXiLyl4LPS4QnieFRv
gnQ4vczCaAMD80J8vOmnhF79dlu3bS17HYTutAQ3QuOI2QpX/CTQjvYAtE8+yzbO
x1HSGo2FAQfTzmBgUHR98jUj7DKZw78o0669mCXa7Sbt7/0WOKo+vatG/xAyhxhJ
TIBECfgAYBJtJQy89/D4TcktSHFu0lUobgd6100N+qkG+Ac3+hlV5pNz/CezOYNR
icsoBdMVbm6x2uBrm2hb5zxZjPDpsHlLgCibZuxTaMH87j2kBto2JujtnqACRW1I
Up98TujTbRuq7nLpApNUIQ==
`pragma protect data_block
YMhTB1BsL7t1WYE9lg240jls9DpGtgkAAWcnGmIrerJZOaYbWwv5IX2K6GC8GLxo
ZiOy+5hD0HXOV2lUdBCgsQ==
`pragma protect end_protected

initial $display("this is not secret stuff");

`pragma protect begin_protected
`pragma protect version=1
`pragma protect encrypt_agent="dvlencrypt"
`pragma protect encrypt_agent_info="Primarius Technologies Co., Ltd."
`pragma protect data_method="aes256-cbc"
`pragma protect key_keyowner="Primarius Technologies Co., Ltd."
`pragma protect key_keyname="VeriSim"
`pragma protect key_method="rsa"
`pragma protect key_block
h2kkOBw22xuxebhjy4m0t8eKR5tUMZ+jKIJ/MPmUR6LyXamMQEhn0K9Za4OT5L+b
```

```
nqlsq/LaDqVn/QQwNEU1fj2F9Sfv2qh2UqCehBjB6e1EQmOxRq/wpXcf8UjXec98
ciZ9ApIyk9J7u+ry4L4oyrkWSPM3MwziOMFLnD0K7Me8kKrPFmSVaB4AtBhVbppD
n27gb9TXKHiJpUnSMzwPuCSyA3NPZzKNx0W3DS2Zfb1yYB172xxxZnRGnUvmGdtx
ehLyyxg0pBRvLxvosTeKHBftR4VIImDgrIoP9igD0e2OwMGN1IK+MES60knkK/PN
xtYNr62w2pVD1M6xvqOBsA==
`pragma protect data_block
DJ/RDdH/kAkPAvbtUKV16pN7g0zpyInx45VovXssPey7NjoalXdtgEQOT20cmlQF
WyoYBL7nUkP8I5WYMZ0mcQ==
`pragma protect end_protected
endmodule
```

LRM Extensions

All Language Reference Manual (LRM) references are with respect to IEEE Std 1800™-2012(Revision of IEEE Std 1800-2009).

Functional Coverage

Cross coverage between crosses

A cross coverage may involve another cross, a. k. a. cross of crosses. As an extension to the LRM, it is supported with limitation: bin definitions are not permitted in a cross of crosses.

Force/Release

LRM section 10.6.2 discusses force and release procedural statements and indicates some limitations. The following limitatations are lifted as an LRM extension. Force/release of:

- individually indexed bits or part selects of a variable
- individually indexed singular elements of an unpacked array (variable or net)
- entire unpacked arrays

Please note force/release of singular elements of unpacked structs is still *unsupported*.

$deposit(signal, value)

This system call is similar to a force statement: it sets the specified signal to a certain value, but unlike force does not need to be released. The value is used to simulate the signal until a new value is assigned. The assignment takes effect immediately with the highest precedence, it will override a procedural force or assign.

The value is released when another update of the signal occurs through regular processing of the simulation.

The signal can be a variable/net, or an expression on a variable/net, such as an array reference or slice, indicating where to put the value. The value's should match the signal expression's type.

Parameters of dynamic array type

LRM section 6.20.1 discusses parameter declaration syntax and indicates some limitations. The following limitatations are lifted as an LRM extension.

- *unsized_dimension*, i.e. `[]` may follow after *parameter_identifier*. For instance:

```
localparam integer DYNARR[] = '{ 1, 2, 3 };
```

Virtual Interface Self-Reference

The name of a virtual interface can now be used to refer to the current instance of the interface, much in the same way that `this` refers to the current instance of an object. To enable this extension provide the `-allow-self-vif` option to VeriSim.

```
package P;
    function void register_if(virtual foo_if vif);
    ...
    endfunction
endpackage

interface foo_if;
import P::*;
initial register_if(foo_if); // registers current instance
endinterface
```

Notes on VHDL Integral Arithmetic

The LRM for 2008 and earlier puts no constraints on the precision of `integer` or `universal_integer`, other than the width must be at least 32 bits. The 2019 LRM mandates 64 bits of precision.

For maximum legacy code compatibility, we use 32-bit integers for VHDL2008 and earlier, and 64-bit integers for VHDL2019. This treatment applies only to integral types; physical types use 64-bit representation regardless of language version selected.

VHDL requires integral overflow checking. e.g. LRM2008 5.2.3.1: "The same arithmetic operators are predefined for all integer types (see 9.2). It is an error if the execution of such an operation (in particular, an implicit conversion) cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the integer type)."

Integer overflow checking incurs some overhead, and many VHDL implementations do not perform overflow checking for this reason. VeriSim does by default; this can be disabled with the `-vhdl-no-ov-chk` analysis and elaboration time option.

NOTE: Overflow checking is different from bounds checking. e.g.:

```
variable a,b,c: integer;

b := 65536;
c := 65536;
a := b * c;
```

In this example, assuming 32-bit integers, the arithmetic result of multiplying 65536 by 65536 is `2**32`. However, this is congruent to zero modulo `2**32`. A range check will not fail here, as the wrapped value zero is within the bounds of an integer. An overflow check *will* fail here, as the represented result is not equal to the infinitely precise mathematical result.

IEEE 1735 Encryption In VHDL

There is currently no support for creating encrypted VHDL sources. However, support is included for decryption. VHDL code can be encrypted using an alternate IEEE1735 compatible encryptor and the VeriSim public key.

FAQ

Assertions

Assertions at the end of simulation

Q. How many assertions are checked at the end of simulation time?

Evaluation attempts may still be in progress at the end of simulation time. The number of such evaluation attempts is reported at the bottom of the log file. For instance:

```
=N: 12 SVA evaluations in progress at end of simulation,
     7 considered passed, 5 considered failed.
```

indicates that 12 evaluations attempts were in progress at the end of simulation time. Given that no subsequent sampling tick exist, the evaluation engine was forced to make pass or fail decision: 7 (out of 12) evaluation attempts have passed and 5 (out of 12) evaluation attempts have failed.

Q. Why is the action block of my assertion not executed at the end of simulation time?

After the scheduler event queue is exhausted or terminated by the `$finish` task, a verdict for the ongoing evaluation attempts is determined. At this point, the action block of an assertion statement cannot be scheduled. Instead:

- if the assertion fails, the fail statement is superseded by an `$error`. The `(end of simulation)` is appended to the end of the `$error` message to indicate the need to supersede the fail statement.
- if the assertion passes, the pass statement is superseded by a null statement `;`

Debug

Call stack

Q. How to display the call stack?

Call the `$stacktrace` system task. The stack is displayed as it is seen from the point of calling `$stacktrace`.

Known Issues

Version: 20220822.0.0

All Language Reference Manual references are noted with the following abbreviations:

- **SV-LRM** : SystemVerilog Language Reference Manual IEEE Std 1800™-2017(Revision of IEEE Std 1800-2012)
- **VHDL-LRM** : VHDL Language Reference Manual IEEE Std 1076™-2000

VHDL

Beta Version: Software and results are provided **as is** and should be used for testing purposes only

VHDL-LRM extensions:

- IEEE Std 1076™-2008
- IEEE Std 1076™-2000

SystemVerilog

Limitations with forced variables and nets

A force/release statement on singular elements (selected with a constant) of unpacked arrays is supported. However, a force/release statement on unpacked structures, unions, or elements thereof is not supported. The LHS expression of the assignment in the procedural force/release statement:

shall reference a non-random static variable or a net with a built-in net type:

- or shall be a constant indexing/field select thereof,
- or shall be a concatenation thereof

shall not involve a select operation in the LHS expansion where:

- the indexing is not constant
- the prefix type is a user-defined nettype
- the prefix type is an unpacked structure
- the prefix type is an unpacked union
- the prefix type is a dynamically sized array

shall have a singular data type:

- except entire unpacked arrays, these may be forced/released

If a forced statement (active or inactive) is applied to a net/variable or part thereof, the following accesses to that net/variable are prohibited:

- Non-singular assignment such as:
  - assignment to an unpacked array slice
- Mutating array methods (e.g. sort, shuffle)

- Randomize
- Pass any of the following through a ref port
  - the said net/variable as a whole
  - a part-select of the said net/variable, even if the part-select itself is not forced
  - a bit-select of the said net/variable, even if the bit-select itself is not forced

SV-LRM Reference: 10.6.2 The force and release procedural statements

SystemVerilog Assertions

Description:

Use of an unsupported SVA element will be flagged as a compile-time error.

The following properties are not currently supported: `accept_on`, `reject_on`, `sync_accept_on` and `sync_reject_on`. All other property and sequence operators are supported.

Additional limitations on assertions are:

1. Only variables, nets, expressions and event expressions can be passed as arguments to named sequences and properties. Arguments or local variables of type sequence or property are not yet supported.

2. Recursive properties containing sample value functions, sequence methods or their own clocking event are not yet supported.

3. Assertion control system tasks (section 20.12 in the SV-LRM) for ON, OFF and KILL are supported, but those related to assert action tasks are not. User may use a compile time option `-vacuous` to control whether or not to execute an assertion's pass action on vacuous success.

4. VPI access to SVA elements (section 37) and Assertion API (section 39) are only partially supported:

   –

   a. VPI access to assertion related objects (sections 37.47 to 37.54) is limited to concurrent and immediate assert, assume and cover statements. Access to property inst, sequence inst, property declaration, sequence declaration, property expression and sequence expression is not supported.

   –

   b. Placing assertion system callbacks is supported only for cbAssertionSysInitialized reason.

   –

   c. Placing assertion callbacks is supported only for reasons cbAssertionStart, cbAssertionSuccess, cbAssertionVacuousSuccess and cbAssertionFailure.

5. Code coverage control and API (section 40) for assertions is not supported. Coverage information for concurrent assertions and immediate cover statements are always written to the coverage database.

6. A multiclocked sequence or property concatenating two possibly overlapping clock ticks may have a race condition if the ticks happen in different re-entries to the observe region during the same time step.

7. An initialization assignment to local variables is not supported for properties with multiple semantic leading clocks, though an equivalent property is supported as long as the local variable initialization is done outside the local variable declaration.

8. The evaluation of inout arguments to system tasks used in deferred assertion action blocks are not evaluated until the deferred time. All other input arguments, and inout arguments to non-system tasks and functions are evaluated immediately.

User may specify `-no-sva` command option to ignore any concurrent assertion statement, expect property statement, property declaration, and sequence declaration.

SV-LRM Reference: 16 Assertions

Specify Blocks

Zero Delay Glitch with two Specify Blocks

Description:

Zero delay glitches may occur on a net under the influence of two specify blocks: the first specify block is inside the module instantiated in another module which contains the second specify block.

In the higher level module, a transition may be scheduled on the net in the same time slot that a zero delay change is happening in the lower level module. Due to nondeterminism in the evaluation order of these two events, the net resolved value may experience a zero delay glitch.

SV-LRM Reference: 30 Specify Blocks

General SV-LRM Support

Description:

There are several parts of the SV-LRM which are not yet supported in VeriSim as indicated below.

SV-LRM Reference:

SV-LRM Section	Notes
5 Lexical Conventions	Attributes are ignored
7.3 Unions	Tagged unions not supported, untagged (packed and unpacked) are supported
9.4.3 Level-sensitive event control	wait/wait fork supported, wait_order is not
10.11 Net aliasing	Feature supported but not all semantic checks are implemented.
11.9 Tagged union expressions and member access	Not supported
11.11 Operator overloading	Will never be supported; removed in 1800-2017
11.13 Let construct	passing event expressions not supported
12.4 Conditional if-else statement	Supported, with exception of matches
12.5 Case statement	Supported, with exception of matches.
12.6 Pattern matching conditional statements	Not supported
16.9.4 Global clocking past	Not supported

SV-LRM Section	Notes
and future sampled value functions	
17 Checkers	Not supported
18.5 Constraint blocks	Supported. Two constraint solvers are supported: a BDD solver which produces correctly distributed results, but which does not scale to large problems, and a SAT solver which handles constraint sets too large for BDD, but which does not necessarily provide properly distributed results.
18.17 Random sequence generation	Not supported
19.3 Defining the coverage model: covergroup	Sample @@begin/end not supported
20.12 Assertion control system tasks	Assert controls ON, OFF and KILL are implemented for concurrent, deferred, and immediate assertions. All others are not supported
20.13 Sample value system functions	Global clocking past and future sample value functions not supported
20.14 Coverage system functions	Support functional coverage only
20.17 Programmable logic array (PLA) functions	Not supported
22.10 `celldefine and `endcelldefine	ignored
22.11 `pragma	Not supported, except for IEEE 1735 protection
23.4 Nested modules	Not supported
23.5 Extern modules	Not supported
25.6 Interfaces and specify blocks	Interface modports used as source or destination in a module path are not supported
25.7 Tasks/Functions	extern/forkjoin not supported, basic task/function/import/export is supported
31.8 Vector signals in timing checks	Vectors are supported but the option to create multiple single bit timing checks is not supported
32.9 Loading timing data	Supported, with the exception of a

SV-LRM Section	Notes
from SDF	providing config file
33 Configuring the contents of a design	Not supported: parameter overrides
34 Protected envelopes	IEEE 1735 V1 is supported, V2 is not entirely. Contact Primarius for key if interested.
36 Programming language interface (PLI/VPI)	Rudimentary support for certain PLI libraries available
37 VPI object model diagrams	Rudimentary support for subset required for UVM register model backdoor. This support should be considered experimental.
38 VPI routine definitions	Rudimentary support for subset required for UVM register model backdoor. This support should be considered experimental.
39 Assertion API	Partial support
40 Code coverage control and API	Not Supported
41 Data read API	Not Supported

VHDL 2008 support

VHDL2008 LRM Section	Notes
5.3.2 Array types	Element constraints partially supported
5.3.3 Record types	Element constraints not supported
5.6 Protected types	Operator overloading for methods not supported
6.3 Subtype declarations	Element constraints and record element resolution not supported. Array element resolution is

VHDL2008 LRM Section	Notes
	supported.
6.5 Interface declarations	Interface type/subprogram/package declarations not supported for entities.
6.11 PSL	Not supported.
9.3.6 Type conversions	Conversion between floating array and integer array not supported.
9.4.3 Static expressions	Entity attributes are not treated as globally static.
10.5 Signal assignment statement	force/release not supported.
11.6 Concurrent signal assignment	force/release not supported.
16.2 Predefined attributes	attributes new to 2008 not supported.
17-23 VHPI	not supported.

VHDL 2019 support

The conditional analysis directives are supported. No other feature new to 2019 is supported.

Legalese: Licenses and Attributions

VeriSim uses several open-source packages whose licenses may impose requirements on VeriSim itself. It also makes use of some libries provided as packages by the operating system.

Operating System Provided Packages

The following packaged are needed by VeriSim and are provided by the operating system:

- ca-certificates
- binutils - required to provide the system linker `ld` for VeriSim to link the final image
- libedit2
- sqlite3

Open-source Packages

The opensource packages are:

LLVM

LLVM is a general-purpose compiler backend optimization and code generation framework. VeriSim uses it to generate native x86_64 code.

LLVM is licensed under the University of Illinois/NCSA Open Source License which requires that the following be reproduced in the documentation:

However, LLVM is in the process of changing its license terms to be those of the Apache 2 license. Clause 4a) of this license requires that anyone who redistributes the software must provide users with a copy of the license, to wit:

```
e
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modificati
ons,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or Obje
ct
      form, that is based on (or derived from) the Work and for which t
he
      editorial revisions, annotations, elaborations, or other modifica
tions
      represent, as a whole, an original work of authorship. For the pu
rposes
      of this License, Derivative Works shall not include works that re
main
      separable from, or merely link (or bind by name) to the interface
s of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, including
      the original version of the Work and any modifications or additio
ns
      to that Work or Derivative Works thereof, that is intentionally
      submitted to Licensor for inclusion in the Work by the copyright
owner
      or by an individual or Legal Entity authorized to submit on behal
f of
      the copyright owner. For the purposes of this definition, "submit
ted"
      means any form of electronic, verbal, or written communication se
nt
      to the Licensor or its representatives, including but not limited
 to
```

Boehm-GC

The Boehm-Demers-Weiser garbage collector is a conservative mark-and-sweep collector that is currently used for runtime garbage collection. It is released under the following license terms:

Colorado University Decision Diagrams (CUDD)

The Colorado University Decision Diagram (CUDD) package is used to build and traverse reduced-order binary decision diagrams as a solving strategy in the constraint solver. Its license requires that the following text be reproduced in the documentation:

libunwind

An independent implementation of `libunwind`, a utility that traverses call stacks is used to generate backtraces upon receipt of a fatal signal or invocation of

`$stacktrace`. Its license requires that the following text be reproduced in the documentation:

Minisat

Minisat is a small SAT solver, which is used as one of the solvers in the constraint solver. It is released under the “MIT license”, their version of which reads as follows:

Minijson

Minijson is a small C++ JSON reader/writer package. It is released under the following license:

libatomic_ops

This package provides semi-portable access to hardware-provided atomic memory update operations on a number of architectures.

A few library routines are covered by the GNU General Public License. These are put into a separate library, libatomic_ops_gpl.a and VeriSim **does not** include or distribute these.

The low-level part of the library used by VeriSim is mostly covered by the following license:

A few files in the sysdeps directory were inherited in part from the Boehm-Demers-Weiser conservative garbage collector, and are covered by its license, which is similar in spirit:

openssl

The OpenSSL toolkit stays under a double license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts.

libunwind

The libunwind library is for unwinding a call stack. It's distribution license is below: