Verilator 4.200

https://verilator.org

2021-03-12

Contents

1 NAME

Verilator - Translate and simulate SystemVerilog code using C++/SystemC

2 SYNOPSIS

```
verilator --help
verilator --version
verilator --cc [options] [source_files.v]... [opt_c_files.cpp/c/cc/a/o/so]
verilator --sc [options] [source_files.v]... [opt_c_files.cpp/c/cc/a/o/so]
verilator --lint-only -Wall [source_files.v]...
```

3 DESCRIPTION

The "Verilator" package converts all synthesizable, and many behavioral, Verilog and SystemVerilog designs into a C++ or SystemC model that after compiling can be executed. Verilator is not a traditional simulator, but a compiler.

Verilator is typically used as follows:

1. The `verilator` executable is invoked with parameters similar to GCC, Cadence Verilog-XL/NC-Verilog, or Synopsys VCS. `verilator` reads the specified user's SystemVerilog code, lints it, optionally adds coverage and waveform tracing support, and compiles the design into a source level C++ or SystemC "model". The resulting model's C++ or SystemC code is output as .cpp and .h files. This is referred to as "verilating" and the process is "to verilate"; the output is a "verilated" model.

2. For simulation, a small user written C++ wrapper file is required, the "wrapper". This wrapper defines the C++ function 'main()' which instantiates the Verilated model as a C++/SystemC object.

3. The user main wrapper, the files created by Verilator, a "runtime library" provided by Verilator, and if applicable the SystemC libraries are then compiled using a C++ compiler to create a simulation executable.

4. The resulting executable will perform the actual simulation, during "simulation runtime".

To get started, jump down to the §7 section.

4 ARGUMENT SUMMARY

This is a short summary of the arguments to the "verilator" executable. See §5 for the detailed descriptions of these arguments.

```
{file.v}                    Verilog package, module and top module filenames
{file.c/cc/cpp}             Optional C++ files to compile in
{file.a/o/so}               Optional C++ files to link in
```

```
 +1364-1995ext+<ext>        Use Verilog 1995 with file extension <ext>
 +1364-2001ext+<ext>        Use Verilog 2001 with file extension <ext>
 +1364-2005ext+<ext>        Use Verilog 2005 with file extension <ext>
 +1800-2005ext+<ext>        Use SystemVerilog 2005 with file extension <ext>
 +1800-2009ext+<ext>        Use SystemVerilog 2009 with file extension <ext>
 +1800-2012ext+<ext>        Use SystemVerilog 2012 with file extension <ext>
 +1800-2017ext+<ext>        Use SystemVerilog 2017 with file extension <ext>
--assert                    Enable all assertions
--autoflush                 Flush streams after all $displays
--bbox-sys                  Blackbox unknown $system calls
--bbox-unsup                Blackbox unsupported language features
--bin <filename>            Override Verilator binary
--build                     Build model executable/library after Verilation
 -CFLAGS <flags>            C++ compiler flags for makefile
--cc                        Create C++ output
--cdc                       Clock domain crossing analysis
--clk <signal-name>         Mark specified signal as clock
--make <build-tool>         Generate scripts for specified build tool
--compiler <compiler-name>  Tune for specified C++ compiler
--converge-limit <loops>    Tune convergence settle time
--coverage                  Enable all coverage
--coverage-line             Enable line coverage
--coverage-toggle           Enable toggle coverage
--coverage-user             Enable SVL user coverage
--coverage-underscore       Enable coverage of _signals
 -D<var>[=<value>]          Set preprocessor define
--debug                     Enable debugging
--debug-check               Enable debugging assertions
--no-debug-leak             Disable leaking memory in --debug mode
--debugi <level>            Enable debugging at a specified level
--debugi-<srcfile> <level>  Enable debugging a source file at a level
--default-language <lang>   Default language to parse
 +define+<var>=<value>      Set preprocessor define
--dpi-hdr-only              Only produce the DPI header file
--dump-defines              Show preprocessor defines with -E
--dump-tree                 Enable dumping .tree files
--dump-treei <level>        Enable dumping .tree files at a level
--dump-treei-<srcfile> <level>  Enable dumping .tree file at a source file at a level
--dump-tree-addrids         Use short identifiers instead of addresses
 -E                         Preprocess, but do not compile
--error-limit <value>       Abort after this number of errors
--exe                       Link to create executable
 -F <file>                  Parse options from a file, relatively
 -f <file>                  Parse options from a file
 -FI <file>                 Force include of a file
--flatten                   Force inlining of all modules, tasks and functions
 -G<name>=<value>           Overwrite top-level parameter
--gdb                       Run Verilator under GDB interactively
--gdbbt                     Run Verilator under GDB for backtrace
--generate-key              Create random key for --protect-key
--getenv <var>              Get environment variable with defaults
--help                      Display this help
--hierarchical              Enable hierarchical Verilation
 -I<dir>                    Directory to search for includes
```

```
 -j <jobs>                  Parallelism for --build
--gate-stmts <value>        Tune gate optimizer depth
--if-depth <value>          Tune IFDEPTH warning
 +incdir+<dir>              Directory to search for includes
--inhibit-sim               Create function to turn off sim
--inline-mult <value>       Tune module inlining
 -LDFLAGS <flags>           Linker pre-object flags for makefile
--l2-name <value>           Verilog scope name of the top module
--language <lang>           Default language standard to parse
 +libext+<ext>+[ext]...     Extensions for finding modules
--lint-only                 Lint, but do not make output
 -MAKEFLAGS <flags>         Options to make during --build
--max-num-width <value>     Maximum number width (default: 64K)
--MMD                       Create .d dependency files
--MP                        Create phony dependency targets
--Mdir <directory>          Name of output object directory
--mod-prefix <topname>      Name to prepend to lower classes
--no-clk <signal-name>      Prevent marking specified signal as clock
--no-decoration             Disable comments and symbol decorations
--no-pins64                 Don't use vluint64_t's for 33-64 bit sigs
--no-skip-identical         Disable skipping identical output
 +notimingchecks            Ignored
 -O0                        Disable optimizations
 -O3                        High performance optimizations
 -O<optimization-letter>    Selectable optimizations
 -o <executable>            Name of final executable
--no-order-clock-delay      Disable ordering clock enable assignments
--no-verilate               Skip verilation and just compile previously Verilated code.
--output-split <statements>          Split .cpp files into pieces
--output-split-cfuncs <statements>   Split model functions
--output-split-ctrace <statements>   Split tracing functions
 -P                         Disable line numbers and blanks with -E
--pins-bv <bits>            Specify types for top level ports
--pins-sc-uint              Specify types for top level ports
--pins-sc-biguint           Specify types for top level ports
--pins-uint8                Specify types for top level ports
--pipe-filter <command>     Filter all input through a script
--pp-comments               Show preprocessor comments with -E
--prefix <topname>          Name of top level class
--prof-cfuncs               Name functions for profiling
--prof-threads              Enable generating gantt chart data for threads
--protect-key <key>         Key for symbol protection
--protect-ids               Hash identifier names for obscurity
--protect-lib <name>        Create a DPI protected library
--private                   Debugging; see docs
--public                    Debugging; see docs
--public-flat-rw            Mark all variables, etc as public_flat_rw
 -pvalue+<name>=<value>     Overwrite toplevel parameter
--quiet-exit                Don't print the command on failure
--relative-includes         Resolve includes relative to current file
--no-relative-cfuncs        Disallow 'this->' in generated functions
--report-unoptflat          Extra diagnostics for UNOPTFLAT
--rr                        Run Verilator and record with rr
--savable                   Enable model save-restore
```

```
--sc                        Create SystemC output
--stats                     Create statistics file
--stats-vars                Provide statistics on variables
 -sv                        Enable SystemVerilog parsing
 +systemverilogext+<ext>    Synonym for +1800-2017ext+<ext>
--threads <threads>         Enable multithreading
--threads-dpi <mode>        Enable multithreaded DPI
--threads-max-mtasks <mtasks>  Tune maximum mtask partitioning
--timescale <timescale>     Sets default timescale
--timescale-override <timescale>  Overrides all timescales
--top <topname>             Alias of --top-module
--top-module <topname>      Name of top level input module
--trace                     Enable waveform creation
--trace-coverage            Enable tracing of coverage
--trace-depth <levels>      Depth of tracing
--trace-fst                 Enable FST waveform creation
--trace-max-array <depth>   Maximum bit width for tracing
--trace-max-width <width>   Maximum array depth for tracing
--trace-params              Enable tracing of parameters
--trace-structs             Enable tracing structure names
--trace-threads <threads>   Enable waveform creation on separate threads
--trace-underscore          Enable tracing of _signals
 -U<var>                    Undefine preprocessor define
--unroll-count <loops>      Tune maximum loop iterations
--unroll-stmts <stmts>      Tune maximum loop body size
--unused-regexp <regexp>    Tune UNUSED lint signals
 -V                         Verbose version and config
 -v <filename>              Verilog library
 +verilog1995ext+<ext>      Synonym for +1364-1995ext+<ext>
 +verilog2001ext+<ext>      Synonym for +1364-2001ext+<ext>
--version                   Displays program version and exits
--vpi                       Enable VPI compiles
--waiver-output <filename>  Create a waiver file based on the linter warnings
 -Wall                      Enable all style warnings
 -Werror-<message>          Convert warnings to errors
 -Wfuture-<message>         Disable unknown message warnings
 -Wno-<message>             Disable warning
 -Wno-context               Disable source context on warnings
 -Wno-fatal                 Disable fatal exit on warnings
 -Wno-lint                  Disable all lint warnings
 -Wno-style                 Disable all style warnings
 -Wpedantic                 Warn on compliance-test issues
--x-assign <mode>           Assign non-initial Xs to this value
--x-initial <mode>          Assign initial Xs to this value
--x-initial-edge            Enable initial X->0 and X->1 edge triggers
--xml-only                  Create XML parser output
--xml-output                XML output filename
 -y <dir>                   Directory to search for modules
```

This is a short summary of the simulation runtime arguments, i.e. for the final Verilated simulation runtime models. See §6 for the detailed description of these arguments.

```
 +verilator+debug                Enable debugging
```

```
+verilator+debugi+<value>         Enable debugging at a level
+verilator+help                   Display help
+verilator+prof+threads+file+<filename>  Set profile filename
+verilator+prof+threads+start+<value>    Set profile starting point
+verilator+prof+threads+window+<value>   Set profile duration
+verilator+rand+reset+<value>     Set random reset technique
+verilator+seed+<value>           Set random seed
+verilator+noassert               Disable assert checking
+verilator+V                      Verbose version and config
+verilator+version                Show version and exit
```

5 VERILATION ARGUMENTS

The following are the arguments that may be passed to the "verilator" executable.

{file.v}

Specifies the Verilog file containing the top module to be Verilated.

{file.c/.cc/.cpp/.cxx}

Used with --exe to specify optional C++ files to be linked in with the Verilog code. The file path should either be absolute, or relative to where the make will be executed from, or add to your makefile's VPATH the appropriate directory to find the file.

See also the -CFLAGS and -LDFLAGS options, which are useful when the C++ files need special compiler flags.

{file.a/.o/.so}

Specifies optional object or library files to be linked in with the Verilog code, as a shorthand for -LDFLAGS "<file>". The file path should either be absolute, or relative to where the make will be executed from, or add to your makefile's VPATH the appropriate directory to find the file.

If any files are specified in this way, Verilator will include a make rule that uses these files when linking the *module* executable. This generally is only useful when used with the --exe option.

+1364-1995ext+*ext*

+1364-2001ext+*ext*

+1364-2005ext+*ext*

+1800-2005ext+*ext*

+1800-2009ext+*ext*

+1800-2012ext+*ext*

+1800-2017ext+*ext*

Specifies the language standard to be used with a specific filename extension, *ext*.

For compatibility with other simulators, see also the synonyms `+verilog1995ext+`*ext*, `+verilog2001ext+`*ext*, and `+systemverilogext+`*ext*.

For any source file, the language specified by these options takes precedence over any language specified by the `--default-language` or `--language` options.

These options take effect in the order they are encountered. Thus the following would use Verilog 1995 for `a.v` and Verilog 2001 for `b.v`.

```
verilator ... +1364-1995ext+v a.v +1364-2001ext+v b.v
```

These flags are only recommended for legacy mixed language designs, as the preferable option is to edit the code to repair new keywords, or add appropriate `‘begin_keywords`.

Note `‘begin_keywords` is a SystemVerilog construct, which specifies *only* the set of keywords to be recognized. This also controls some error messages that vary between language standards. Note at present Verilator tends to be overly permissive, e.g. it will accept many grammar and other semantic extensions which might not be legal when set to an older standard.

--assert

Enable all assertions.

--autoflush

After every $display or $fdisplay, flush the output stream. This ensures that messages will appear immediately but may reduce performance. For best performance call "fflush(stdout)" occasionally in the C++ main loop. Defaults to off, which will buffer output as provided by the normal C/C++ standard library IO.

--bbox-sys

Black box any unknown $system task or function calls. System tasks will simply become no-operations, and system functions will be replaced with unsized zero. Arguments to such functions will be parsed, but not otherwise checked. This prevents errors when linting in the presence of company specific PLI calls.

Using this argument will likely cause incorrect simulation.

--bbox-unsup

Black box some unsupported language features, currently UDP tables, the cmos and tran gate primitives, deassign statements, and mixed edge errors. This may enable linting the rest of the design even when unsupported constructs are present.

Using this argument will likely cause incorrect simulation.

--bin ***filename***

Rarely needed. Override the default filename for Verilator itself. When a dependency (.d) file is created, this filename will become a source dependency, such that a change in this binary will have make rebuild the output files.

--build

After generating the SystemC/C++ code, Verilator will invoke the toolchain to build the model library (and executable when `--exe` is also used). Verilator manages the build itself, and for this --build requires GNU Make to be available on the platform.

-CFLAGS ***flags***

Add specified C compiler flag to the generated makefiles. For multiple flags either pass them as a single argument with space separators quoted in the shell (`-CFLAGS "-a -b"`), or use multiple -CFLAGS arguments (`-CFLAGS -a -CFLAGS -b`).

When make is run on the generated makefile these will be passed to the C++ compiler (g++/clang++/msvc++).

--cc

Specifies C++ without SystemC output mode; see also --sc.

--cdc

Permanently experimental. Perform some clock domain crossing checks and issue related warnings (CDCRSTLOGIC) and then exit; if warnings other than CDC warnings are needed make a second run with --lint-only. Additional warning information is also written to the file {prefix}__cdc.txt.

Currently only checks some items that other CDC tools missed; if you have interest in adding more traditional CDC checks, please contact the authors.

--clk ***signal-name***

Sometimes it is quite difficult for Verilator to distinguish clock signals from other data signals. Occasionally the clock signals can end up in the checking list of signals which determines if further evaluation is needed. This will heavily degrade the performance of a Verilated model.

With --clk <signal-name>, user can specified root clock into the model, then Verilator will mark the signal as clocker and propagate the clocker attribute automatically to other signals derived from that. In this way, Verilator will try to avoid taking the clocker signal into checking list.

Note signal-name is specified by the RTL hierarchy path. For example, v.foo.bar. If the signal is the input to top-module, the directly the signal name. If you find it difficult to find the exact name, try to use `/*verilator clocker*/` in RTL file to mark the signal directly.

If clock signals are assigned to vectors and then later used individually, Verilator will attempt to decompose the vector and connect the single-bit clock signals directly. This should be transparent to the user.

--make ***build-tool***

Generates a script for the specified build tool.

Supported values are `gmake` for GNU Make and `cmake` for CMake. Both can be specified together. If no build tool is specified, gmake is assumed. The executable of gmake can be configured via environment variable "MAKE".

When using --build Verilator takes over the responsibility of building the model library/executable. For this reason --make cannot be specified when using --build.

--compiler ***compiler-name***

Enables workarounds for the specified C++ compiler, either `clang`, `gcc`, or `msvc`. Currently this does not change any performance tuning flags, but it may in the future.

clang

Tune for clang. This may reduce execution speed as it enables several workarounds to avoid silly hard-coded limits in clang. This includes breaking deep structures as for msvc as described below.

gcc

Tune for GNU C++, although generated code should work on almost any compliant C++ compiler. Currently the default.

msvc

Tune for Microsoft Visual C++. This may reduce execution speed as it enables several workarounds to avoid silly hard-coded limits in MSVC++. This includes breaking deeply nested parenthesized expressions into sub-expressions to avoid error C1009, and breaking deep blocks into functions to avoid error C1061.

--converge-limit ***loops***

Rarely needed. Specifies the maximum number of runtime iterations before creating a model failed to converge error. Defaults to 100.

--coverage

Enables all forms of coverage, alias for "--coverage-line --coverage-toggle --coverage-user".

--coverage-line

Specifies basic block line coverage analysis code should be inserted.

Coverage analysis adds statements at each code flow change point (e.g. at branches). At each such branch a unique counter is incremented. At the end of a test, the counters along with the filename and line number corresponding to each counter are written into logs/coverage.dat.

Verilator automatically disables coverage of branches that have a $stop in them, as it is assumed $stop branches contain an error check that should not occur. A /*verilator coverage_block_off*/ comment will perform a similar function on any code in that block or below, or /*verilator coverage_on/coverage_off*/ will disable coverage around lines of code.

Note Verilator may over-count combinatorial (non-clocked) blocks when those blocks receive signals which have had the UNOPTFLAT warning disabled; for most accurate results do not disable this warning when using coverage.

--coverage-toggle

Specifies signal toggle coverage analysis code should be inserted.

Every bit of every signal in a module has a counter inserted. The counter will increment on every edge change of the corresponding bit.

Signals that are part of tasks or begin/end blocks are considered local variables and are not covered. Signals that begin with underscores, are integers, or are very wide (>256 bits total storage across all dimensions) are also not covered.

Hierarchy is compressed, such that if a module is instantiated multiple times, coverage will be summed for that bit across ALL instantiations of that module with the same parameter set. A module instantiated with different parameter values is considered a different module, and will get counted separately.

Verilator makes a minimally-intelligent decision about what clock domain the signal goes to, and only looks for edges in that clock domain. This means that edges may be ignored if it is known that the edge could never be seen by the receiving logic. This algorithm may improve in the future. The net result is coverage may be lower than what would be seen by looking at traces, but the coverage is a more accurate representation of the quality of stimulus into the design.

There may be edges counted near time zero while the model stabilizes. It's a good practice to zero all coverage just before releasing reset to prevent counting such behavior.

A /*verilator coverage_off/on */ comment pair can be used around signals that do not need toggle analysis, such as RAMs and register files.

--coverage-underscore

Enable coverage of signals that start with an underscore. Normally, these signals are not covered. See also --trace-underscore.

--coverage-user

Enables user inserted functional coverage. Currently, all functional coverage points are specified using SVA which must be separately enabled with --assert.

For example, the following statement will add a coverage point, with the comment "DefaultClock":

```
DefaultClock: cover property (@(posedge clk) cyc==3);
```

-D*var*=*value*

Defines the given preprocessor symbol. Similar to +define, but does not allow multiple definitions with a single option using plus signs. +define is fairly standard across Verilog tools while -D is similar to GCC.

--debug

Select the debug executable of Verilator (if available), and enable more internal assertions (equivalent to `--debug-check`), debugging messages (equivalent to `--debugi 3`), and intermediate form dump files (equivalent to `--dump-treei 3`).

--debug-check

Rarely needed. Enable internal debugging assertion checks, without changing debug verbosity. Enabled automatically when --debug specified.

--no-debug-leak

In --debug mode, by default Verilator intentionally leaks AstNode instances instead of freeing them, so that each node pointer is unique in the resulting tree files and dot files.

This option disables the leak. This may avoid out-of-memory errors when Verilating large models in --debug mode.

Outside of --debug mode, AstNode instances should never be leaked and this option has no effect.

--debugi ***level***

--debugi-*srcfile level*

Rarely needed - for developer use. Set internal debugging level globally to the specified debug level (1-10) or set the specified Verilator source file to the specified level (e.g. `--debugi-V3Width 9`). Higher levels produce more detailed messages.

--default-language ***value***

Select the language to be used by default when first processing each Verilog file. The language value must be "1364-1995", "1364-2001", "1364-2005", "1800-2005", "1800-2009", "1800-2012" or "1800-2017".

Any language associated with a particular file extension (see the various +*lang*ext+ options) will be used in preference to the language specified by --default-language.

The --default-language flag is only recommended for legacy code using the same language in all source files, as the preferable option is to edit the code to repair new keywords, or add appropriate `‘begin_keywords`. For legacy mixed language designs, the various +*lang*ext+ options should be used.

If no language is specified, either by this flag or +*lang*ext+ options, then the latest SystemVerilog language (IEEE 1800-2017) is used.

+define+*var*=*value*

+define+*var*=*value*+*var2*=*value2*...

Defines the given preprocessor symbol, or multiple symbols if separated by plus signs. Similar to -D; +define is fairly standard across Verilog tools while -D is similar to GCC.

--dpi-hdr-only

Only generate the DPI header file. This option has no effect on the name or location of the emitted DPI header file, it is output in `--Mdir` as it would be without this option.

--dump-defines

With -E, suppress normal output, and instead print a list of all defines existing at the end of preprocessing the input files. Similar to GCC "-dM" option. This also gives you a way of finding out what is predefined in Verilator using the command:

```
touch foo.v ; verilator -E --dump-defines foo.v
```

--dump-tree

Rarely needed. Enable writing .tree debug files with dumping level 3, which dumps the standard critical stages. For details on the format see the Verilator Internals manual. --dump-tree is enabled automatically with --debug, so "--debug --no-dump-tree" may be useful if the dump files are large and not desired.

--dump-treei ***level***

--dump-treei-*srcfile* *level*

Rarely needed - for developer use. Set internal tree dumping level globally to a specific dumping level or set the specified Verilator source file to the specified tree dumping level (e.g. `--dump-treei-V3Order 9`). Level 0 disables dumps and is equivalent to "--no-dump-tree". Level 9 enables dumping of every stage.

--dump-tree-addrids

Rarely needed - for developer use. Replace AST node addresses with short identifiers in tree dumps to enhance readability. Each unique pointer value is mapped to a unique identifier, but note that this is not necessarily unique per node instance as an address might get reused by a newly allocated node after a node with the same address has been dumped then freed.

-E

Preprocess the source code, but do not compile, as with 'gcc -E'. Output is written to standard out. Beware of enabling debugging messages, as they will also go to standard out.

--error-limit ***value***

After this number of errors are encountered during Verilator run, exit. Warnings are not counted in this limit. Defaults to 50.

Does not affect simulation runtime errors, for those see +verilator+error+limit.

--exe

Generate an executable. You will also need to pass additional .cpp files on the command line that implement the main loop for your simulation.

-F ***file***

Read the specified file, and act as if all text inside it was specified as command line parameters. Any relative paths are relative to the directory containing the specified file. See also -f. Note -F is fairly standard across Verilog tools.

-f ***file***

Read the specified file, and act as if all text inside it was specified as command line parameters. Any relative paths are relative to the current directory. See also -F. Note -f is fairly standard across Verilog tools.

The file may contain // comments which are ignored to the end of the line. Any $VAR, $(VAR), or ${VAR} will be replaced with the specified environment variable.

-FI ***file***

Force include of the specified C++ header file. All generated C++ files will insert a #include of the specified file before any other includes. The specified file might be used to contain define prototypes of custom VL_VPRINTF functions, and may need to include verilatedos.h as this file is included before any other standard includes.

--flatten

Force flattening of the design's hierarchy, with all modules, tasks and functions inlined. Typically used with `--xml-only`. Note flattening large designs may require significant CPU time, memory and storage.

-G*name*=*value*

Overwrites the given parameter of the toplevel module. The value is limited to basic data literals:

Verilog integer literals

The standard Verilog integer literals are supported, so values like 32'h8, 2'b00, 4 etc. are allowed. Care must be taken that the single quote (I') is properly escaped in an interactive shell, e.g., as -GWIDTH=8\'hx.

C integer literals

It is also possible to use C integer notation, including hexadecimal (0x..), octal (0..) or binary (0b..) notation.

Double literals

Double literals must be one of the following styles: - contains a dot (.) (e.g. 1.23) - contains an exponent (e/E) (e.g. 12e3) - contains p/P for hexadecimal floating point in C99 (e.g. 0x123.ABCp1)

Strings

Strings must be in double quotes (""). They must be escaped properly on the command line, e.g. as -GSTR="\"My String\"" or -GSTR='"My String"'.

--gate-stmts *value*

Rarely needed. Set the maximum number of statements that may be present in an equation for the gate substitution optimization to inline that equation.

--gdb

Run Verilator underneath an interactive GDB (or VERILATOR_GDB environment variable value) session. See also --gdbbt.

--gdbbt

If --debug is specified, run Verilator underneath a GDB process and print a backtrace on exit, then exit GDB immediately. Without --debug or if GDB doesn't seem to work, this flag is ignored. Intended for easy creation of backtraces by users; otherwise see the --gdb flag.

--generate-key

Generate a true-random key suitable for use with --protect-key, print it, and exit immediately.

--getenv *variable*

If the variable is declared in the environment, print it and exit immediately. Otherwise, if it's built into Verilator (e.g. VERILATOR_ROOT), print that and exit immediately. Otherwise, print a newline and exit immediately. This can be useful in makefiles. See also -V, and the various *.mk files.

--help

Displays this message and program version and exits.

--hierarchical

Enable hierarchical Verilation otherwise /*verilator hier_block*/ metacomment is ignored. See §18.

-I*dir*

See -y.

--if-depth *value*

Rarely needed. Set the depth at which the IFDEPTH warning will fire, defaults to 0 which disables this warning.

+incdir+*dir*

See -y.

--inhibit-sim

Rarely needed. Create a "inhibitSim(bool)" function to enable and disable evaluation. This allows an upper level testbench to disable modules that are not important in a given simulation, without needing to recompile or change the SystemC modules instantiated.

--inline-mult *value*

Tune the inlining of modules. The default value of 2000 specifies that up to 2000 new operations may be added to the model by inlining, if more than this number of operations would result, the module is not inlined. Larger values, or a value < 1 will inline everything, will lead to longer compile times, but potentially faster simulation speed. This setting is ignored for very small modules; they will always be inlined, if allowed.

-j <value>

Specify the level of parallelism for --build. <value> must be a positive integer specifying the maximum number of parallel build jobs, or can be omitted. When <value> is omitted, the build will not try to limit the number of parallel build jobs but attempt to execute all independent build steps in parallel.

-LDFLAGS *flags*

Add specified C linker flags to the generated makefiles. For multiple flags either pass them as a single argument with space separators quoted in the shell (`-LDFLAGS "-a -b"`), or use multiple -LDFLAGS arguments (`-LDFLAGS -a -LDFLAGS -b`).

When make is run on the generated makefile these will be passed to the C++ linker (ld) *after* the primary file being linked. This flag is called -LDFLAGS as that's the traditional name in simulators; it's would have been better called LDLIBS as that's the Makefile variable it controls. (In Make, LDFLAGS is before the first object, LDLIBS after. -L libraries need to be in the Make variable LDLIBS, not LDFLAGS.)

--l2-name *value*

Instead of using the module name when showing Verilog scope, use the name provided. This allows simplifying some Verilator-embedded modeling methodologies. Default is an l2-name matching the top module. The default before 3.884 was "--l2-name v"

For example, the program "module t; initial $display("%m"); endmodule" will show by default "t". With "--l2-name v" it will print "v".

--language *value*

A synonym for `--default-language`, for compatibility with other tools and earlier versions of Verilator.

+libext+*ext*+*ext*...

Specify the extensions that should be used for finding modules. If for example module x is referenced, look in *x.ext*. Note +libext+ is fairly standard across Verilog tools. Defaults to .v and .sv.

--lint-only

Check the files for lint violations only, do not create any other output.

You may also want the -Wall option to enable messages that are considered stylistic and not enabled by default.

If the design is not to be completely Verilated see also the --bbox-sys and --bbox-unsup options.

-MAKEFLAGS <string>

When using --build, add the specified flag to the invoked make command line. For multiple flags either pass them as a single argument with space separators quoted in the shell (e.g. `-MAKEFLAGS "-a -b"`), or use multiple -MAKEFLAGS arguments (e.g. `-MAKEFLAGS -l -MAKEFLAGS -k`). Use of this option should not be required for simple builds using the host toolchain.

--max-num-width *value*

Set the maximum number literal width (e.g. in 1024'd22 this it the 1024). Defaults to 64K.

--MMD =item --no-MMD

Enable/disable creation of .d dependency files, used for make dependency detection, similar to gcc -MMD option. By default this option is enabled for --cc or --sc modes.

--MP

When creating .d dependency files with --MMD, make phony targets. Similar to gcc -MP option.

--Mdir *directory*

Specifies the name of the Make object directory. All generated files will be placed in this directory. If not specified, "obj_dir" is used. The directory is created if it does not exist and the parent directories exist; otherwise manually create the Mdir before calling Verilator.

--mod-prefix *topname*

Specifies the name to prepend to all lower level classes. Defaults to the same as --prefix.

--no-clk *signal-name*

Prevent the specified signal from being marked as clock. See `--clk`.

--no-decoration

When creating output Verilated code, minimize comments, white space, symbol names and other decorative items, at the cost of greatly reduced readability. This may assist C++ compile times. This will not typically change the ultimate model's performance, but may in some cases.

--no-pins64

Backward compatible alias for "--pins-bv 33".

--no-relative-cfuncs

Disable 'this->' references in generated functions, and instead Verilator will generate absolute references starting from 'vlTOPp->'. This prevents V3Combine from merging functions from multiple instances of the same module, so it can grow the instruction stream.

This is a work around for old compilers. Don't set this if your C++ compiler supports __restrict__ properly, as GCC 4.5.x and newer do. For older compilers, test if this switch gives you better performance or not.

Compilers which don't honor __restrict__ will suspect that 'this->' references and 'vlTOPp->' references may alias, and may write slow code with extra loads and stores to handle the (imaginary) aliasing. Using only 'vlTOPp->' references allows these old compilers to produce tight code.

--no-skip-identical =item --skip-identical

Rarely needed. Disables or enables skipping execution of Verilator if all source files are identical, and all output files exist with newer dates. By default this option is enabled for --cc or --sc modes only.

+notimingchecks

Ignored for compatibility with other simulators.

-O0

Disables optimization of the model.

-O3

Enables slow optimizations for the code Verilator itself generates (as opposed to "-CFLAGS -O3" which effects the C compiler's optimization. -O3 may improve simulation performance at the cost of compile time. This currently sets --inline-mult -1.

-O*optimization-letter*

Rarely needed. Enables or disables a specific optimizations, with the optimization selected based on the letter passed. A lowercase letter disables an optimization, an upper case letter enables it. This is intended for debugging use only; see the source code for version-dependent mappings of optimizations to -O letters.

-o *executable*

Specify the name for the final executable built if using --exe. Defaults to the --prefix if not specified.

--no-order-clock-delay

Rarely needed. Disables a bug fix for ordering of clock enables with delayed assignments. This flag should only be used when suggested by the developers.

--output-split *statements*

Enables splitting the output .cpp files into multiple outputs. When a C++ file exceeds the specified number of operations, a new file will be created at the next function boundary. In addition, if the total output code size exceeds the specified value, VM_PARALLEL_BUILDS will be set to 1 by default in the generated make files, making parallel compilation possible. Using --output-split should have only a trivial impact on model performance. But can greatly improve C++ compilation speed. The use of *ccache* (set for you if present at configure time) is also more effective with this option.

This option is on by default with a value of 20000. To disable, pass with a value of 0.

--output-split-cfuncs *statements*

Enables splitting functions in the output .cpp files into multiple functions. When a generated function exceeds the specified number of operations, a new function will be created. With --output-split, this will enable the C++ compiler to compile faster, at a small loss in performance that gets worse with decreasing split values. Note that this option is stronger than --output-split in the sense that --output-split will not split inside a function.

Defaults to the value of --output-split, unless explicitly specified.

--output-split-ctrace *statements*

Similar to --output-split-cfuncs, enables splitting trace functions in the output .cpp files into multiple functions.

Defaults to the value of --output-split, unless explicitly specified.

-P

With -E, disable generation of 'line markers and blank lines, similar to GCC -P flag.

--pins64

Backward compatible alias for "--pins-bv 65". Note that's a 65, not a 64.

--pins-bv *width*

Specifies SystemC inputs/outputs of greater than or equal to *width* bits wide should use sc_bv's instead of uint32/vluint64_t's. The default is "--pins-bv 65", and the value must be less than or equal to 65. Versions before Verilator 3.671 defaulted to "--pins-bv 33". The more sc_bv is used, the worse for performance. Use the "/*verilator sc_bv*/" attribute to select specific ports to be sc_bv.

--pins-sc-uint

Specifies SystemC inputs/outputs of greater than 2 bits wide should use sc_uint between 2 and 64. When combined with the "--pins-sc-biguint" combination, it results in sc_uint being used between 2 and 64 and sc_biguint being used between 65 and 512.

--pins-sc-biguint

Specifies SystemC inputs/outputs of greater than 65 bits wide should use sc_biguint between 65 and 512, and sc_bv from 513 upwards. When combined with the "--pins-sc-uint" combination, it results in sc_uint being used between 2 and 64 and sc_biguint being used between 65 and 512.

--pins-uint8

Specifies SystemC inputs/outputs that are smaller than the --pins-bv setting and 8 bits or less should use uint8_t instead of uint32_t. Likewise pins of width 9-16 will use uint16_t instead of uint32_t.

--pipe-filter ***command***

Rarely needed. Verilator will spawn the specified command as a subprocess pipe, to allow the command to perform custom edits on the Verilog code before it reaches Verilator.

Before reading each Verilog file, Verilator will pass the file name to the subprocess' stdin with 'read "<filename>"'. The filter may then read the file and perform any filtering it desires, and feeds the new file contents back to Verilator on stdout by first emitting a line defining the length in bytes of the filtered output 'Content-Length: <bytes>', followed by the new filtered contents. Output to stderr from the filter feeds through to Verilator's stdout and if the filter exits with non-zero status Verilator terminates. See the t/t_pipe_filter test for an example.

To debug the output of the filter, try using the -E option to see preprocessed output.

--pp-comments

With -E, show comments in preprocessor output.

--prefix ***topname***

Specifies the name of the top level class and makefile. Defaults to V prepended to the name of the --top switch, or V prepended to the first Verilog filename passed on the command line.

--prof-cfuncs

Modify the created C++ functions to support profiling. The functions will be minimized to contain one "basic" statement, generally a single always block or wire statement. (Note this will slow down the executable by ~5%.) Furthermore, the function name will be suffixed with the basename of the Verilog module and line number the statement came from. This allows gprof or oprofile reports to be correlated with the original Verilog source statements. See also verilator_profcfunc.

--prof-threads

Enable gantt chart data collection for threaded builds.

Verilator will record the start and end time of each macro-task across a number of calls to eval. (What is a macro-task? See the Verilator internals document.)

When profiling is enabled, the simulation runtime will emit a blurb of profiling data in non-human-friendly form. The `verilator_gantt` script will transform this into a nicer visual format and produce some related statistics.

--protect-key ***key***

Specifies the private key for --protect-ids. For best security this key should be 16 or more random bytes, a reasonable secure choice is the output of `verilator --generate-key`. Typically, a key would be created by the user once for a given protected design library, then every Verilator run for subsequent versions of that library would be passed the same --protect-key. Thus, if the input Verilog is similar between library versions (Verilator runs), the Verilated code will likewise be mostly similar.

If --protect-key is not specified and a key is needed, Verilator will generate a new key for every Verilator run. As the key is not saved, this is best for security, but means every Verilator run will give vastly different output even for identical input, perhaps harming compile times (and certainly thrashing any *ccache*).

--protect-ids

Hash any private identifiers (variable, module, and assertion block names that are not on the top level) into hashed random-looking identifiers, resulting after compilation in protected library binaries that expose less design information. This hashing uses the provided or default --protect-key, see important details there.

Verilator will also create a {prefix}__idmap.xml file which contains the mapping from the hashed identifiers back to the original identifiers. This idmap file is to be kept private, and is to assist mapping any simulation runtime design assertions, coverage, or trace information, which will report the hashed identifiers, back to the original design's identifier names.

Using DPI imports/exports is allowed and generally relatively safe in terms of information disclosed, which is limited to the DPI function prototyptes. Use of the VPI is not recommended as many design details may be exposed, and an INSECURE warning will be issued.

--protect-lib ***name***

Produces C++, Verilog wrappers and a Makefile which can in turn produce a DPI library which can be used by Verilator or other simulators along with the corresponding Verilog wrapper. The Makefile will build both a static and dynamic version of the library named lib*name*.a and lib*name*.so respectively. This is done because some simulators require a dynamic library, but the static library is arguably easier to use if possible. --protect-lib implies --protect-ids.

This allows for the secure delivery of sensitive IP without the need for encrypted RTL (i.e. IEEE P1735). See examples/make_protect_lib in the distribution for a demonstration of how to build and use the DPI library.

When using --protect-lib it is advised to also use `--timescale-override /1fs` to ensure the model has a time resolution that is always compatible with the time precision of the upper instantiating module.

--private

Opposite of --public. Is the default; this option exists for backwards compatibility.

--public

This is only for historical debug use. Using it may result in mis-simulation of generated clocks.

Declares all signals and modules public. This will turn off signal optimizations as if all signals had a /*verilator public*/ comments and inlining. This will also turn off inlining as if all modules had a /*verilator public_module*/, unless the module specifically enabled it with /*verilator inline_module*/.

--public-flat-rw

Declares all variables, ports and wires public as if they had /*verilator public_flat_rw @ (<variable's_source_process_e
comments. This will make them VPI accessible by their flat name, but not turn off module inlining. This is particularly useful in combination with --vpi. This may also in some rare cases result in mis-simulation of generated clocks. Instead of this global switch, marking only those signals that need public_flat_rw is typically significantly better performing.

-pvalue+*name*=*value*

Overwrites the given parameter(s) of the toplevel module. See -G for a detailed description.

--quiet-exit

When exiting due to an error, do not display the "Exiting due to Errors" nor "Command Failed" messages.

--relative-includes

When a file references an include file, resolve the filename relative to the path of the referencing file, instead of relative to the current directory.

--report-unoptflat

Extra diagnostics for UNOPTFLAT warnings. This includes for each loop, the 10 widest variables in the loop, and the 10 most fanned out variables in the loop. These are candidates for splitting into multiple variables to break the loop.

In addition produces a GraphViz DOT file of the entire strongly connected components within the source associated with each loop. This is produced irrespective of whether --dump-tree is set. Such graphs may help in analyzing the problem, but can be very large indeed.

Various commands exist for viewing and manipulating DOT files. For example the *dot* command can be used to convert a DOT file to a PDF for printing. For example:

```
dot -Tpdf -O Vt_unoptflat_simple_2_35_unoptflat.dot
```

will generate a PDF Vt_unoptflat_simple_2_35_unoptflat.dot.pdf from the DOT file.

As an alternative, the *xdot* command can be used to view DOT files interactively:

```
xdot Vt_unoptflat_simple_2_35_unoptflat.dot
```

--rr

Run Verilator and record with rr. See: rr-project.org.

--savable

Enable including save and restore functions in the generated model.

The user code must create a VerilatedSerialize or VerilatedDeserialze object then calling the << or >> operators on the generated model and any other data the process needs saved/restored. These functions are not thread safe, and are typically called only by a main thread.

For example:

```
void save_model(const char* filenamep) {
    VerilatedSave os;
    os.open(filenamep);
    os << main_time;  // user code must save the timestamp, etc
    os << *topp;
}
void restore_model(const char* filenamep) {
    VerilatedRestore os;
    os.open(filenamep);
    os >> main_time;
    os >> *topp;
}
```

--sc

Specifies SystemC output mode; see also --cc.

--stats

Creates a dump file with statistics on the design in {prefix}__stats.txt.

--stats-vars

Creates more detailed statistics, including a list of all the variables by size (plain --stats just gives a count). See --stats, which is implied by this.

--structs-packed

Converts all unpacked structures to packed structures and issues a UNPACKED warning. Currently this is the default and --no-structs-packed will not work. Specifying this option allows for forward compatibility when a future version of Verilator no longer always packs unpacked structures.

-sv

Specifies SystemVerilog language features should be enabled; equivalent to "--language 1800-2017". This option is selected by default, it exists for compatibility with other simulators.

+systemverilogext+*ext*

A synonym for `+1800-2017ext+`*ext*.

--threads *threads*

--no-threads

With --threads 0 or --no-threads, the default, the generated model is not thread safe. With --threads 1, the generated model is single threaded but may run in a multithreaded environment. With --threads N, where N >= 2, the model is generated to run multithreaded on up to N threads. See §19.

--threads-dpi all

--threads-dpi none

--threads-dpi pure

When using --threads, controls which DPI imported tasks and functions are considered thread safe.

With --threads-dpi all, enable Verilator to assume all DPI imports are threadsafe, and to use thread-local storage for communication with DPI, potentially improving performance. Any DPI libraries need appropriate mutexes to avoid undefined behavior.

With --threads-dpi none, Verilator assume DPI imports are not thread safe, and Verilator will serialize calls to DPI imports by default, potentially harming performance.

With --threads-dpi pure, the default, Verilator assumes DPI pure imports are threadsafe, but non-pure DPI imports are not.

--threads-max-mtasks *value*

Rarely needed. When using --threads, specify the number of mtasks the model is to be partitioned into. If unspecified, Verilator approximates a good value.

--timescale *timeunit*/*timeprecision*

Sets default timescale, timeunit and timeprecision for when ‘timescale does not occur in sources. Default is "1ps/1ps" (to match SystemC). This is overridden by `--timescale-override`.

--timescale-override *timeunit*/*timeprecision*

--timescale-override /*timeprecision*

Overrides all ‘timescales in sources. The timeunit may be left empty to specify only to override the timeprecision, e.g. "/1fs".

The time precision must be consistent with SystemC's sc_set_time_resolution, or the C++ code instantiating the Verilated module. As 1fs is the finest time precision it may be desirable to always use a precision of 1fs.

--top *topname*

--top-module *topname*

When the input Verilog contains more than one top level module, specifies the name of the Verilog module to become the top level module, and sets the default for --prefix if not explicitly specified. This is not needed with standard designs with only one top. See also the MULTITOP warning section.

--trace

Adds waveform tracing code to the model using VCD format. This overrides `--trace-fst`.

Verilator will generate additional {prefix}__Trace*.cpp files that will need to be compiled. In addition verilated_vcd_sc.cpp (for SystemC traces) or verilated_vcd_c.cpp (for both) must be compiled and linked in. If using the Verilator generated Makefiles, these files will be added to the source file lists for you. If you are not using the Verilator Makefiles, you will need to add these to your Makefile manually.

Having tracing compiled in may result in some small performance losses, even when tracing is not turned on during model execution.

See also `--trace-threads`.

--trace-coverage

With --trace and --coverage-*, enable tracing to include a traced signal for every --coverage-line or --coverage-user inserted coverage point, to assist in debugging coverage items. Note --coverage-toggle does not get additional signals added, as the original signals being toggle-analyzed are already visible.

The added signal will be a 32-bit value which will increment on each coverage occurrence. Due to this, this option may greatly increase trace file sizes and reduce simulation speed.

--trace-depth ***levels***

Specify the number of levels deep to enable tracing, for example --trace-level 1 to only see the top level's signals. Defaults to the entire model. Using a small number will decrease visibility, but greatly improve simulation performance and trace file size.

--trace-fst

Enable FST waveform tracing in the model. This overrides `--trace`. See also `--trace-threads`.

--trace-max-array ***depth***

Rarely needed. Specify the maximum array depth of a signal that may be traced. Defaults to 32, as tracing large arrays may greatly slow traced simulations.

--trace-max-width ***width***

Rarely needed. Specify the maximum bit width of a signal that may be traced. Defaults to 256, as tracing large vectors may greatly slow traced simulations.

--no-trace-params

Disable tracing of parameters.

--trace-structs

Enable tracing to show the name of packed structure, union, and packed array fields, rather than a single combined packed bus. Due to VCD file format constraints this may result in significantly slower trace times and larger trace files.

--trace-threads ***threads***

Enable waveform tracing using separate threads. This is typically faster in simulation runtime but uses more total compute. This option is independent of, and works with, both `--trace` and `--trace-fst`. Different trace formats can take advantage of more trace threads to varying degrees. Currently VCD tracing can utilize at most --trace-threads 1, and FST tracing can utilize at most --trace-threads 2. This overrides `--no-threads`.

--trace-underscore

Enable tracing of signals that start with an underscore. Normally, these signals are not output during tracing. See also --coverage-underscore.

-U*var*

Undefines the given preprocessor symbol.

--unroll-count *loops*

Rarely needed. Specifies the maximum number of loop iterations that may be unrolled. See also BLKLOOPINIT warning.

--unroll-stmts *statements*

Rarely needed. Specifies the maximum number of statements in a loop for that loop to be unrolled. See also BLKLOOPINIT warning.

--unused-regexp *regexp*

Rarely needed. Specifies a simple regexp with * and ? that if a signal name matches will suppress the UNUSED warning. Defaults to "*unused*". Setting it to "" disables matching.

-V

Shows the verbose version, including configuration information compiled into Verilator. (Similar to perl -V.) See also --getenv.

-v *filename*

Read the filename as a Verilog library. Any modules in the file may be used to resolve instances in the top level module, else ignored. Note -v is fairly standard across Verilog tools.

--no-verilate

When using --build, disable generation of C++/SystemC code, and execute only the build. This can be useful for rebuilding Verilated code produced by a previous invocation of Verilator.

+verilog1995ext+*ext*

+verilog2001ext+*ext*

Synonyms for `+1364-1995ext+`*ext* and `+1364-2001ext+`*ext* respectively

--version

Displays program version and exits.

--vpi

Enable use of VPI and linking against the verilated_vpi.cpp files.

--waiver-output <filename>

Generate a waiver file which contains all waiver statements to suppress the warnings emitted during this Verilator run. This in particular is useful as a starting point for solving linter warnings or suppressing them systematically.

The generated file is in the Verilator Configuration format, see §20, and can directly be consumed by Verilator. The standard file extension is .vlt.

-Wall

Enable all code style warnings, including code style warnings that are normally disabled by default. Equivalent to "-Wwarn-lint -Wwarn-style". Excludes some specialty warnings, i.e. IMPERFECTSCH.

-Werror-*message*

Promote the specified warning message into an error message. This is generally to discourage users from violating important site-wide rules, for example `-Werror-NOUNOPTFLAT`.

-Wfuture-*message*

Rarely needed. Suppress unknown Verilator comments or warning messages with the given message code. This is used to allow code written with pragmas for a later version of Verilator to run under a older version; add -Wfuture- arguments for each message code or comment that the new version supports which the older version does not support.

-Wno-*message*

Disable the specified warning/error message. This will override any lint_on directives in the source, i.e. the warning will still not be printed.

-Wno-context

Disable showing the suspected context of the warning message by quoting the source text at the suspected location. This can be used to appease tools which process the warning messages but may get confused by lines from the original source.

-Wno-fatal

When warnings are detected, print them, but do not terminate Verilator.

Having warning messages in builds is sloppy. It is strongly recommended you cleanup your code, use inline lint_off, or use -Wno-... flags rather than using this option.

-Wno-lint

Disable all lint related warning messages, and all style warnings. This is equivalent to "-Wno-ALWCOMBORDER -Wno-BSSPACE -Wno-CASEINCOMPLETE -Wno-CASEOVERLAP -Wno-CASEX -Wno-CASTCONST -Wno-CASEWITHX -Wno-CMPCONST -Wno-COLONPLUS -Wno-ENDLABEL -Wno-IMPLICIT -Wno-LITENDIAN -Wno-PINCONNECTEMPTY -Wno-PINMISSING -Wno-SYNCASYNCNET -Wno-UNDRIVEN -Wno-UNSIGNED -Wno-UNUSED -Wno-WIDTH" plus the list shown for Wno-style.

It is strongly recommended you cleanup your code rather than using this option, it is only intended to be use when running test-cases of code received from third parties.

-Wno-style

Disable all code style related warning messages (note by default they are already disabled). This is equivalent to "-Wno-DECLFILENAME -Wno-DEFPARAM -Wno-IMPORTSTAR -Wno-INCABSPATH -Wno-PINCONNECTEMPTY -Wno-PINNOCONNECT -Wno-SYNCASYNCNET -Wno-UNDRIVEN -Wno-UNUSED -Wno-VARHIDDEN".

-Wpedantic

Warn on any construct demanded by IEEE, and disable all Verilator extensions that may interfere with IEEE compliance to the standard defined with --default-language (etc). Similar to GCC's -Wpedantic. Rarely used, and intended only for strict compliance tests.

-Wwarn-*message*

Enables the specified warning message.

-Wwarn-lint

Enable all lint related warning messages (note by default they are already enabled), but do not affect style messages. This is equivalent to "-Wwarn-ALWCOMBORDER -Wwarn-BSSPACE -Wwarn-CASEINCOMPLETE -Wwarn-CASEOVERLAP -Wwarn-CASEX -Wwarn-CASTCONST -Wwarn-CASEWITHX -Wwarn-CMPCONST -Wwarn-COLONPLUS -Wwarn-ENDLABEL -Wwarn-IMPLICIT -Wwarn-LITENDIAN -Wwarn-PINMISSING -Wwarn-REALCVT -Wwarn-UNSIGNED -Wwarn-WIDTH".

-Wwarn-style

Enable all code style related warning messages. This is equivalent to "-Wwarn ASSIGNDLY -Wwarn-DECLFILENAME -Wwarn-DEFPARAM -Wwarn-INCABSPATH -Wwarn-PINNOCONNECT -Wwarn-SYNCASYNCNET -Wwarn-UNDRIVEN -Wwarn-UNUSED -Wwarn-VARHIDDEN".

--x-assign 0

--x-assign 1

--x-assign fast (default)

--x-assign unique

Controls the two-state value that is substituted when an explicit X value is encountered in the source. `--x-assign fast`, the default, converts all Xs to whatever is best for performance. `--x-assign 0` converts all Xs to 0s, and is also fast. `--x-assign 1` converts all Xs to 1s, this is nearly as fast as 0, but more likely to find reset bugs as active high logic will fire. Using `--x-assign unique` will result in all explicit Xs being replaced by a constant value determined at runtime. The value is determined by calling a function at initialization time. This enables randomization of Xs with different seeds on different executions. This method is the slowest, but safest for finding reset bugs.

If using --x-assign unique, you may want to seed your random number generator such that each regression run gets a different randomization sequence. The simplest is to use the +verilator+seed runtime option. Alternatively use the system's srand48() or for Windows srand() function to do this. You'll probably also want to print any seeds selected, and code to enable rerunning with that same seed so you can reproduce bugs.

Note. This option applies only to values which are explicitly written as X in the Verilog source code. Initial values of clocks are set to 0 unless --x-initial-edge is specified. Initial values of all other state holding variables are controlled with --x-initial.

--x-initial 0

--x-initial fast

--x-initial unique (default)

Controls the two-state value that is used to initialize variables that are not otherwise initialized.

`--x-initial 0`, initializes all otherwise uninitialized variables to zero.

`--x-initial unique`, the default, initializes variables using a function, which determines the value to use each initialization. This gives greatest flexibility and allows finding reset bugs. See §23.

`--x-initial fast`, is best for performance, and initializes all variables to a state Verilator determines is optimal. This may allow further code optimizations, but will likely hide any code bugs relating to missing resets.

Note. This option applies only to initial values of variables. Initial values of clocks are set to 0 unless --x-initial-edge is specified.

--x-initial-edge

Enables emulation of event driven simulators which generally trigger an edge on a transition from X to 1 (`posedge`) or X to 0 (`negedge`). Thus the following code, where `rst_n` is uninitialized would set `res_n` to `1'b1` when `rst_n` is first set to zero:

```
reg  res_n = 1'b0;

always @(negedge rst_n) begin
   if (rst_n == 1'b0) begin
      res_n <= 1'b1;
   end
end
```

In Verilator, by default, uninitialized clocks are given a value of zero, so the above `always` block would not trigger.

While it is not good practice, there are some designs that rely on X → 0 triggering a `negedge`, particularly in reset sequences. Using --x-initial-edge with Verilator will replicate this behavior. It will also ensure that X → 1 triggers a `posedge`.

Note. Some users have reported that using this option can affect convergence, and that it may be necessary to use --converge-limit to increase the number of convergence iterations. This may be another indication of problems with the modeled design that should be addressed.

--xml-only

Create XML output only, do not create any other output.

The XML format is intended to be used to leverage Verilator's parser and elaboration to feed to other downstream tools. Be aware that the XML format is still evolving; there will be some changes in future versions.

--xml-output ***filename***

Filename for XML output file. Using this option automatically sets --xml-only.

-y ***dir***

Add the directory to the list of directories that should be searched for include files or libraries. The three flags -y, +incdir and -I have similar effect; +incdir and +y are fairly standard across Verilog tools while -I is used by many C++ compilers.

Verilator defaults to the current directory ("-y .") and any specified --Mdir, though these default paths are used after any user specified directories. This allows '-y "$(pwd)"' to be used if absolute filenames are desired for error messages instead of relative filenames.

6 SIMULATION RUNTIME ARGUMENTS

The following are the arguments that may be passed to a Verilated executable, provided that executable calls Verilated::commandArgs().

All simulation runtime arguments begin with +verilator, so that the user's executable may skip over all +verilator arguments when parsing its command line.

+verilator+debug

Enable simulation runtime debugging. Equivalent to +verilator+debugi+4.

+verilator+debugi+*value*

Enable simulation runtime debugging at the provided level.

+verilator+error+limit+*value*

Set number of non-fatal errors (e.g. assertion failures) before exiting simulation runtime. Also affects number of $stop calls needed before exit. Defaults to 1.

+verilator+help

Display help and exit.

+verilator+prof+threads+file+*filename*

When a model was Verilated using --prof-threads, sets the simulation runtime filename to dump to. Defaults to "profile_threads.dat".

+verilator+prof+threads+start+*value*

When a model was Verilated using --prof-threads, the simulation runtime will wait until $time is at this value (expressed in units of the time precision), then start the profiling warmup, then capturing. Generally this should be set to some time that is well within the normal operation of the simulation, i.e. outside of reset. If 0, the dump is disabled. Defaults to 1.

+verilator+prof+threads+window+*value*

When a model was Verilated using --prof-threads, after $time reaches +verilator+prof+threads+start, Verilator will warm up the profiling for this number of eval() calls, then will capture the profiling of this number of eval() calls. Defaults to 2, which makes sense for a single-clock-domain module where it's typical to want to capture one posedge eval() and one negedge eval().

+verilator+rand+reset+*value*

When a model was Verilated using "-x-initial unique", sets the simulation runtime initialization technique. 0 = Reset to zeros. 1 = Reset to all-ones. 2 = Randomize. See §23.

+verilator+seed+*value*

For $random and "-x-initial unique", set the simulation runtime random seed value. If zero or not specified picks a value from the system random number generator.

+verilator+noassert

Disable assert checking per runtime argument. This is the same as calling "Verilated::assertOn(false)" in the model.

+verilator+V

Shows the verbose version, including configuration information.

+verilator+version

Displays program version and exits.

7 EXAMPLE C++ EXECUTION

We'll compile this example into C++. For an extended and commented version of what this C++ code is doing, see examples/make_tracing_c/sim_main.cpp.

```
mkdir test_our
cd test_our

cat >our.v <<'EOF'
  module our;
     initial begin $display("Hello World"); $finish; end
  endmodule
EOF

cat >sim_main.cpp <<'EOF'
  #include "Vour.h"
  #include "verilated.h"
  int main(int argc, char** argv, char** env) {
      VerilatedContext* contextp = new VerilatedContext;
```

```
        contextp->commandArgs(argc, argv);
        Vour* top = new Vour{contextp};
        while (!contextp->gotFinish()) { top->eval(); }
        delete top;
        delete contextp;
        return 0;
    }
EOF
```

Next we need Verilator installed. See the README in the source kit for various ways to install or point to Verilator binaries. In brief, if you installed Verilator using the package manager of your operating system, or did a "make install" to place Verilator into your default path, you do not need anything special in your environment, and should not have VERILATOR_ROOT set. However, if you installed Verilator from sources and want to run Verilator out of where you compiled Verilator, you need to point to the kit:

```
# See above; don't do this if using an OS-distributed Verilator
export VERILATOR_ROOT=/path/to/where/verilator/was/installed
export PATH=$VERILATOR_ROOT/bin:$PATH
```

Now we run Verilator on our little example.

```
verilator -Wall --cc our.v --exe --build sim_main.cpp
```

We can see the source code under the "obj_dir" directory. See the FILES section below for descriptions of some of the files that were created.

```
ls -l obj_dir
```

(Verilator included a default compile rule and link rule, since we used --exe and passed a .cpp file on the Verilator command line. Verilator also then used `make` to build a final executable, since we used --build. You can also write your own compile rules, and run make yourself as we'll show in the SYSTEMC section.

And now we run it:

```
obj_dir/Vour
```

And we get as output:

```
Hello World
- our.v:2: Verilog $finish
```

Really, you're better off writing a Makefile to do all this for you. Then, when your source changes it will automatically run all of these steps; to aid this Verilator can create a makefile dependency file. See the examples directory in the distribution.

8 EXAMPLE SYSTEMC EXECUTION

This is an example similar to the above, but using SystemC.

```
mkdir test_our_sc
cd test_our_sc

cat >our.v <<'EOF'
  module our (clk);
     input clk;  // Clock is required to get initial activation
     always @ (posedge clk)
        begin $display("Hello World"); $finish; end
  endmodule
EOF

cat >sc_main.cpp <<'EOF'
  #include "Vour.h"
  int sc_main(int argc, char** argv) {
      Verilated::commandArgs(argc, argv);
      sc_clock clk{"clk", 10, SC_NS, 0.5, 3, SC_NS, true};
      Vour* top = new Vour{"top"};
      top->clk(clk);
      while (!Verilated::gotFinish()) { sc_start(1, SC_NS); }
      delete top;
      return 0;
  }
EOF
```

See the README in the source kit for various ways to install or point to Verilator binaries. In brief, if you installed Verilator using the package manager of your operating system, or did a "make install" to place Verilator into your default path, you do not need anything special in your environment, and should not have VERILATOR_ROOT set. However, if you installed Verilator from sources and want to run Verilator out of where you compiled Verilator, you need to point to the kit:

```
# See above; don't do this if using an OS-distributed Verilator
export VERILATOR_ROOT=/path/to/where/verilator/was/installed
export PATH=$VERILATOR_ROOT/bin:$PATH
```

Now we run Verilator on our little example.

```
verilator -Wall --sc our.v
```

We then can compile it

```
make -j -C obj_dir -f Vour.mk Vour__ALL.a
make -j -C obj_dir -f Vour.mk ../sc_main.o verilated.o
```

And link with SystemC. Note your path to the libraries may vary, depending on the operating system.

```
cd obj_dir
export SYSTEMC_LIBDIR=/path/to/where/libsystemc.a/exists
export LD_LIBRARY_PATH=$SYSTEMC_LIBDIR:$LD_LIBRARY_PATH
# Might be needed if SystemC 2.3.0
export SYSTEMC_CXX_FLAGS=-pthread


g++ -L$SYSTEMC_LIBDIR ../sc_main.o Vour__ALL.a verilated.o \
          -o Vour -lsystemc
```

And now we run it

```
cd ..
obj_dir/Vour
```

And we get the same output as the C++ example:

```
Hello World
- our.v:2: Verilog $finish
```

Really, you're better off using a Makefile to do all this for you. Then, when your source changes it will automatically run all of these steps. See the examples directory in the distribution.

9 EVALUATION LOOP

When using SystemC, evaluation of the Verilated model is managed by the SystemC kernel, and for the most part can be ignored. When using C++, the user must call eval(), or eval_step() and eval_end_step().

1. When there is a single design instantiated at the C++ level that needs to evaluate within a given context, just call designp->eval().

2. When there are multiple designs instantiated at the C++ level that need to evaluate within a context, call first_designp->eval_step() then ->eval_step() on all other designs. Then call ->eval_end_step() on the first design then all other designs. If there is only a single design, you would call eval_step() then eval_end_step(); in fact eval() described above is just a wrapper which calls these two functions.

When eval() is called Verilator looks for changes in clock signals and evaluates related sequential always blocks, such as computing always_ff @ (posedge...) outputs. Then Verilator evaluates combinatorial logic.

Note combinatorial logic is not computed before sequential always blocks are computed (for speed reasons). Therefore it is best to set any non-clock inputs up with a separate eval() call before changing clocks.

Alternatively, if all always_ff statements use only the posedge of clocks, or all inputs go directly to always_ff statements, as is typical, then you can change non-clock inputs on the negative edge of the input clock, which will be faster as there will be fewer eval() calls.

For more information on evaluation, see docs/internals.rst in the distribution.

10 BENCHMARKING & OPTIMIZATION

For best performance, run Verilator with the "-O3 --x-assign fast --x-initial fast --noassert" flags. The -O3 flag will require longer time to run Verilator, and "--x-assign fast --x-initial fast" may increase the risk of reset bugs in trade for performance; see the above documentation for these flags.

If using Verilated multithreaded, use `numactl` to ensure you are using non-conflicting hardware resources. See §19.

Minor Verilog code changes can also give big wins. You should not have any UNOPTFLAT warnings from Verilator. Fixing these warnings can result in huge improvements; one user fixed their one UNOPTFLAT warning by making a simple change to a clock latch used to gate clocks and gained a 60% performance improvement.

Beyond that, the performance of a Verilated model depends mostly on your C++ compiler and size of your CPU's caches. Experience shows that large models are often limited by the size of the instruction cache, and as such reducing code size if possible can be beneficial.

The supplied $VERILATOR_ROOT/include/verilated.mk file uses the OPT, OPT_FAST, OPT_SLOW and OPT_GLOBAL variables to control optimization. You can set these when compiling the output of Verilator with Make, for example:

```
make OPT_FAST="-Os -march=native" -f Vour.mk Vour__ALL.a
```

OPT_FAST specifies optimization flags for those parts of the model that are on the fast path. This is mostly code that is executed every cycle. OPT_SLOW applies to slow-path code, which executes rarely, often only once at the beginning or end of simulation. Note that OPT_SLOW is ignored if VM_PARALLEL_BUILDS is not 1, in which case all generated code will be compiled in a single compilation unit using OPT_FAST. See also the `--output-split` option. The OPT_GLOBAL variable applies to common code in the runtime library used by Verilated models (shipped in $VERILATOR_ROOT/include). Additional C++ files passed on the verilator command line use OPT_FAST. The OPT variable applies to all compilation units in addition to the specific OPT_* variables described above.

You can also use the -CFLAGS and/or -LDFLAGS options on the verilator command line to pass flags directly to the compiler or linker.

The default values of the OPT_* variables are chosen to yield good simulation speed with reasonable C++ compilation times. To this end, OPT_FAST is set to "-Os" by default. Higher optimization such as "-O2" or "-O3" may help (though often they provide only a very small performance benefit), but compile times may be excessively large even with medium sized designs. Compilation times can be improved at the expense of simulation speed by reducing optimization, for example with OPT_FAST="-O0". Often good simulation speed can be achieved with OPT_FAST="-O1 -fstrict-aliasing" but with improved compilation times. Files controlled by OPT_SLOW have little effect on performance and therefore OPT_SLOW is empty by default (equivalent to "-O0") for improved compilation speed. In common use-cases there should be little benefit in changing OPT_SLOW. OPT_GLOBAL is set to "-Os" by default and there should rarely be a need to change it. As the runtime library is small in comparison to a lot of Verilated models, disabling optimization on the runtime library should not have a serious effect on overall compilation time, but may have detrimental effect on simulation speed, especially with tracing. In addition to the above, for best results use OPT="-march=native", the latest Clang compiler (about 10% faster than GCC), and link statically.

Generally the answer to which optimization level gives the best user experience depends on the use case and some experimentation can pay dividends. For a speedy debug cycle during development, especially on

large designs where C++ compilation speed can dominate, consider using lower optimization to get to an executable faster. For throughput oriented use cases, for example regressions, it is usually worth spending extra compilation time to reduce total CPU time.

If you will be running many simulations on a single model, you can investigate profile guided optimization. With GCC, using -fprofile-arcs, then -fbranch-probabilities will yield another 15% or so.

Modern compilers also support link-time optimization (LTO), which can help especially if you link in DPI code. To enable LTO on GCC, pass "-flto" in both compilation and link. Note LTO may cause excessive compile times on large designs.

Unfortunately, using the optimizer with SystemC files can result in compilation taking several minutes. (The SystemC libraries have many little inlined functions that drive the compiler nuts.)

If you are using your own makefiles, you may want to compile the Verilated code with -DVL_INLINE_OPT=inline. This will inline functions, however this requires that all cpp files be compiled in a single compiler run.

You may uncover further tuning possibilities by profiling the Verilog code. Use Verilator's --prof-cfuncs, then GCC's -g -pg. You can then run either oprofile or gprof to see where in the C++ code the time is spent. Run the gprof output through verilator_profcfunc and it will tell you what Verilog line numbers on which most of the time is being spent.

When done, please let the author know the results. We like to keep tabs on how Verilator compares, and may be able to suggest additional improvements.

11 FILES

All output files are placed in the output directory specified with the `--Mdir` option, or "obj_dir" if not specified.

Verilator creates the following files in the output directory:

For --make gmake, it creates:

```
{prefix}.mk                     // Make include file for compiling
{prefix}_classes.mk             // Make include file with class names
{prefix}_hier.mk                // Make file for hierarchy blocks
{prefix}_hierMkArgs.f           // Arguments for hierarchical Verilation.
```

For --make cmake, it creates:

```
{prefix}.cmake                  // CMake include script for compiling
{prefix}_hierCMakeArgs.f        // Arguments for hierarchical Verilation.
```

For -cc and -sc mode, it also creates:

```
{prefix}.cpp                    // Top level C++ file
{prefix}.h                      // Top level header
```

```
{prefix}__Slow{__n}.cpp             // Constructors and infrequent cold routines
{prefix}{__n}.cpp                   // Additional top C++ files (--output-split)
{prefix}{each_verilog_module}.cpp   // Lower level internal C++ files
{prefix}{each_verilog_module}.h     // Lower level internal header files
{prefix}{each_verilog_module}{__n}.cpp   // Additional lower C++ files (--output-split)
```

For --hierarchy mode, it creates: {prefix}__hierVer.d // Make dependencies of the top module in the hierarchical Verilation {prefix}__hier.dir/ // Directory to store .dot, .vpp, .tree of the top module in the hierarchical Verilation V{hier_block}/ // Directory to Verilate each hierarchy block

In certain debug and other modes, it also creates:

```
{prefix}.xml                        // XML tree information (--xml)
{prefix}__Dpi.cpp                   // DPI import and export wrappers
{prefix}__Dpi.h                     // DPI import and export declarations
{prefix}__Inlines.h                 // Inline support functions
{prefix}__Syms.cpp                  // Global symbol table C++
{prefix}__Syms.h                    // Global symbol table header
{prefix}__Trace__Slow{__n}.cpp      // Wave file generation code (--trace)
{prefix}__Trace{__n}.cpp            // Wave file generation code (--trace)
{prefix}__cdc.txt                   // Clock Domain Crossing checks (--cdc)
{prefix}__stats.txt                 // Statistics (--stats)
{prefix}__idmap.txt                 // Symbol demangling (--protect-ids)
```

It also creates internal files that can be mostly ignored:

```
{mod_prefix}_{each_verilog_module}{__n}.vpp  // Pre-processed verilog
{prefix}__ver.d                     // Make dependencies (-MMD)
{prefix}__verFiles.dat              // Timestamps for skip-identical
{prefix}{misc}.dot                  // Debugging graph files (--debug)
{prefix}{misc}.tree                 // Debugging files (--debug)
```

After running Make, the C++ compiler may produce the following:

```
verilated{misc}.d                   // Intermediate dependencies
verilated{misc}.o                   // Intermediate objects
{mod_prefix}{misc}.d                // Intermediate dependencies
{mod_prefix}{misc}.o                // Intermediate objects
{prefix}                            // Final executable (w/--exe argument)
{prefix}__ALL.a                     // Library of all Verilated objects
{prefix}__ALL.cpp                   // Include of all code for single compile
{prefix}{misc}.d                    // Intermediate dependencies
{prefix}{misc}.o                    // Intermediate objects
```

12 ENVIRONMENT

LD_LIBRARY_PATH

A generic Linux/OS variable specifying what directories have shared object (.so) files. This path should include SystemC and any other shared objects needed at simulation runtime.

MAKE

Names the executable of the make command invoked when using the --build option. Some operating systems may require "gmake" to this variable to launch GNU make. If this variable is not specified, "make" is used.

OBJCACHE

Optionally specifies a caching or distribution program to place in front of all runs of the C++ compiler. For example, "ccache". If using distcc or icecc/icecream, they would generally be run under cache; see the documentation for those programs. If OBJCACHE is not set, and at configure time ccache was present, ccache will be used as a default.

SYSTEMC

Deprecated. Used only if SYSTEMC_INCLUDE or SYSTEMC_LIBDIR is not set. If set, specifies the directory containing the SystemC distribution. If not specified, it will come from a default optionally specified at configure time (before Verilator was compiled).

SYSTEMC_ARCH

Deprecated. Used only if SYSTEMC_LIBDIR is not set. Specifies the architecture name used by the SystemC kit. This is the part after the dash in the lib-{...} directory name created by a 'make' in the SystemC distribution. If not set, Verilator will try to intuit the proper setting, or use the default optionally specified at configure time (before Verilator was compiled).

SYSTEMC_CXX_FLAGS

Specifies additional flags that are required to be passed to GCC when building the SystemC model. System 2.3.0 may need this set to "-pthread".

SYSTEMC_INCLUDE

If set, specifies the directory containing the systemc.h header file. If not specified, it will come from a default optionally specified at configure time (before Verilator was compiled), or computed from SYSTEMC/include.

SYSTEMC_LIBDIR

If set, specifies the directory containing the libsystemc.a library. If not specified, it will come from a default optionally specified at configure time (before Verilator was compiled), or computed from SYSTEMC/lib-SYSTEMC_ARCH.

VERILATOR_BIN

If set, specifies an alternative name of the `verilator` binary. May be used for debugging and selecting between multiple operating system builds.

VERILATOR_COVERAGE_BIN

If set, specifies an alternative name of the `verilator_coverage binary`. May be used for debugging and selecting between multiple operating system builds.

VERILATOR_GDB

If set, the command to run when using the --gdb option, such as "ddd". If not specified, it will use "gdb".

VERILATOR_ROOT

Specifies the directory containing the distribution kit. This is used to find the executable, Perl library, and include files. If not specified, it will come from a default optionally specified at configure time (before Verilator was compiled). It should not be specified if using a pre-compiled Verilator package as the hard-coded value should be correct.

13 CONNECTING TO C++

Verilator creates a *prefix*.h and *prefix*.cpp file for the top level module, together with additional .h and .cpp files for internals. See the examples directory in the kit for examples.

After the model is created, there will be a *prefix*.mk file that may be used with Make to produce a *prefix*__ALL.a file with all required objects in it. This is then linked with the user's C++ main loop to create the simulation executable.

The user must write the C++ main loop of the simulation. Here is a simple example:

```
#include <verilated.h>          // Defines common routines
#include <iostream>             // Need std::cout
#include "Vtop.h"               // From Verilating "top.v"


Vtop *top;                      // Instantiation of module


vluint64_t main_time = 0;       // Current simulation time
// This is a 64-bit integer to reduce wrap over issues and
// allow modulus.  This is in units of the timeprecision
// used in Verilog (or from --timescale-override)


double sc_time_stamp () {       // Called by $time in Verilog
    return main_time;           // converts to double, to match
                                // what SystemC does
}


int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);   // Remember args


    top = new Vtop;             // Create instance


    top->reset_l = 0;           // Set some inputs


    while (!Verilated::gotFinish()) {
        if (main_time > 10) {
            top->reset_l = 1;   // Deassert reset
        }
        if ((main_time % 10) == 1) {
            top->clk = 1;       // Toggle clock
        }
        if ((main_time % 10) == 6) {
            top->clk = 0;
        }
        top->eval();            // Evaluate model
        cout << top->out << endl;       // Read a output
        main_time++;            // Time passes...
    }
```

```
        top->final();               // Done simulating
        //    // (Though this example doesn't get here)
        delete top;
    }
```

Note signals are read and written as member variables of the model. You call the eval() method to evaluate the model. When the simulation is complete call the final() method to execute any SystemVerilog final blocks, and complete any assertions. See §9.

14 CONNECTING TO SYSTEMC

Verilator will convert the top level module to a SC_MODULE. This module will plug directly into a SystemC netlist.

The SC_MODULE gets the same pinout as the Verilog module, with the following type conversions: Pins of a single bit become bool. Pins 2-32 bits wide become uint32_t's. Pins 33-64 bits wide become sc_bv's or vluint64_t's depending on the --no-pins64 switch. Wider pins become sc_bv's. (Uints simulate the fastest so are used where possible.)

Lower modules are not pure SystemC code. This is a feature, as using the SystemC pin interconnect scheme everywhere would reduce performance by an order of magnitude.

15 DIRECT PROGRAMMING INTERFACE (DPI)

Verilator supports SystemVerilog Direct Programming Interface import and export statements. Only the SystemVerilog form ("DPI-C") is supported, not the original Synopsys-only DPI.

DPI Example

In the SYSTEMC example above, if you wanted to import C++ functions into Verilog, put in our.v:

```
import "DPI-C" function int add (input int a, input int b);

initial begin
   $display("%x + %x = %x", 1, 2, add(1,2));
endtask
```

Then after Verilating, Verilator will create a file Vour__Dpi.h with the prototype to call this function:

```
 extern int add (int a, int b);
```

From the sc_main.cpp file (or another .cpp file passed to the Verilator command line, or the link), you'd then:

```
#include "svdpi.h"
#include "Vour__Dpi.h"
int add(int a, int b) { return a+b; }
```

DPI System Task/Functions

Verilator extends the DPI format to allow using the same scheme to efficiently add system functions. Simply use a dollar-sign prefixed system function name for the import, but note it must be escaped.

```
export "DPI-C" function integer \$myRand;

initial $display("myRand=%d", $myRand());
```

Going the other direction, you can export Verilog tasks so they can be called from C++:

```
export "DPI-C" task publicSetBool;

task publicSetBool;
   input bit in_bool;
   var_bool = in_bool;
endtask
```

Then after Verilating, Verilator will create a file Vour__Dpi.h with the prototype to call this function:

```
extern void publicSetBool(svBit in_bool);
```

From the sc_main.cpp file, you'd then:

```
#include "Vour__Dpi.h"
publicSetBool(value);
```

Or, alternatively, call the function under the design class. This isn't DPI compatible but is easier to read and better supports multiple designs.

```
#include "Vour__Dpi.h"
Vour::publicSetBool(value);
// or top->publicSetBool(value);
```

Note that if the DPI task or function accesses any register or net within the RTL, it will require a scope to be set. This can be done using the standard functions within svdpi.h, after the module is instantiated, but before the task(s) and/or function(s) are called.

For example, if the top level module is instantiated with the name "dut" and the name references within tasks are all hierarchical (dotted) names with respect to that top level module, then the scope could be set with

```
#include "svdpi.h"
...
svSetScope(svGetScopeFromName("TOP.dut"));
```

(Remember that Verilator adds a "TOP" to the top of the module hierarchy.)

Scope can also be set from within a DPI imported C function that has been called from Verilog by querying the scope of that function. See the sections on DPI Context Functions and DPI Header Isolation below and the comments within the svdpi.h header for more information.

DPI Imports that access signals

If a DPI import accesses a signal through the VPI Verilator will not be able to know what variables are accessed and may schedule the code inappropriately. Ideally pass the values as inputs/outputs so the VPI is not required. Alternatively a workaround is to use a non-inlined task as a wrapper:

```
logic din;

// This DPI function will read "din"
import "DPI-C" context function void dpi_that_accesses_din();

always @ (...)
   dpi_din_args(din);

task dpi_din_args(input din);
   /* verilator no_inline_task */
   dpi_that_accesses_din();
endtask
```

DPI Display Functions

Verilator allows writing $display like functions using this syntax:

```
import "DPI-C" function void
      \$my_display(input string formatted /*verilator sformat*/ );
```

The /*verilator sformat*/ indicates that this function accepts a $display like format specifier followed by any number of arguments to satisfy the format.

DPI Context Functions

Verilator supports IEEE DPI Context Functions. Context imports pass the simulator context, including calling scope name, and filename and line number to the C code. For example, in Verilog:

```
import "DPI-C" context function int dpic_line();
initial $display("This is line %d, again, line %d\n", `line, dpic_line());
```

This will call C++ code which may then use the svGet* functions to read information, in this case the line number of the Verilog statement that invoked the dpic_line function:

```
int dpic_line() {
    // Get a scope:  svScope scope = svGetScope();

    const char* scopenamep = svGetNameFromScope(scope);
    assert(scopenamep);

    const char* filenamep = "";
    int lineno = 0;
    if (svGetCallerInfo(&filenamep, &lineno)) {
        printf("dpic_line called from scope %s on line %d\n",
           scopenamep, lineno);
        return lineno;
    } else {
        return 0;
    }
}
```

See the IEEE Standard for more information.

DPI Header Isolation

Verilator places the IEEE standard header files such as svdpi.h into a separate include directory, vltstd (VeriLaTor STandarD). When compiling most applications $VERILATOR_ROOT/include/vltstd would be in the include path along with the normal $VERILATOR_ROOT/include. However, when compiling Verilated models into other simulators which have their own svdpi.h and similar standard files with different contents, the vltstd directory should not be included to prevent picking up incompatible definitions.

Public Functions

Instead of DPI exporting, there's also Verilator public functions, which are slightly faster, but less compatible.

16 VERIFICATION PROCEDURAL INTERFACE (VPI)

Verilator supports a very limited subset of the VPI. This subset allows inspection, examination, value change callbacks, and depositing of values to public signals only.

VPI is enabled with the verilator --vpi switch.

To access signals via the VPI, Verilator must be told exactly which signals are to be accessed. This is done using the Verilator public pragmas documented below.

Verilator has an important difference from an event based simulator; signal values that are changed by the VPI will not immediately propagate their values, instead the top level header file's eval() method must be called. Normally this would be part of the normal evaluation (i.e. the next clock edge), not as part of the value change. This makes the performance of VPI routines extremely fast compared to event based simulators, but can confuse some test-benches that expect immediate propagation.

Note the VPI by its specified implementation will always be much slower than accessing the Verilator values by direct reference (structure->module->signame), as the VPI accessors perform lookup in functions at simulation runtime requiring at best hundreds of instructions, while the direct references are evaluated by the compiler and result in only a couple of instructions.

For signal callbacks to work the main loop of the program must call VerilatedVpi::callValueCbs().

VPI Example

In the below example, we have readme marked read-only, and writeme which if written from outside the model will have the same semantics as if it changed on the specified clock edge.

```
cat >our.v <<'EOF'
  module our (input clk);
     reg readme   /*verilator public_flat_rd*/;
     reg writeme  /*verilator public_flat_rw @(posedge clk) */;
     initial $finish;
  endmodule
EOF
```

There are many online tutorials and books on the VPI, but an example that accesses the above signal "readme" would be:

```
cat >sim_main.cpp <<'<<EOF'
  #include "Vour.h"
  #include "verilated.h"
  #include "verilated_vpi.h"  // Required to get definitions


  vluint64_t main_time = 0;   // See comments in first example
  double sc_time_stamp() { return main_time; }


  void read_and_check() {
      vpiHandle vh1 = vpi_handle_by_name((PLI_BYTE8*)"TOP.our.readme", NULL);
      if (!vh1) { vl_fatal(__FILE__, __LINE__, "sim_main", "No handle found"); }
      const char* name = vpi_get_str(vpiName, vh1);
      printf("Module name: %s\n", name);  // Prints "readme"


      s_vpi_value v;
      v.format = vpiIntVal;
```

```
        vpi_get_value(vh1, &v);
        printf("Value of v: %d\n", v.value.integer);  // Prints "readme"
    }

    int main(int argc, char** argv, char** env) {
        Verilated::commandArgs(argc, argv);
        Vour* top = new Vour;
        Verilated::internalsDump();  // See scopes to help debug
        while (!Verilated::gotFinish()) {
            top->eval();
            VerilatedVpi::callValueCbs();  // For signal callbacks
            read_and_check();
        }
        delete top;
        return 0;
    }
EOF
```

17 CROSS COMPILATION

Verilator supports cross-compiling Verilated code. This is generally used to run Verilator on a Linux system and produce C++ code that is then compiled on Windows.

Cross compilation involves up to three different OSes. The build system is where you configured and compiled Verilator, the host system where you run Verilator, and the target system where you compile the Verilated code and run the simulation.

Currently, Verilator requires the build and host system type to be the same, though the target system type may be different. To support this, ./configure and make Verilator on the build system. Then, run Verilator on the host system. Finally, the output of Verilator may be compiled on the different target system.

To support this, none of the files that Verilator produces will reference any configure generated build-system specific files, such as config.h (which is renamed in Verilator to config_build.h to reduce confusion.) The disadvantage of this approach is that include/verilatedos.h must self-detect the requirements of the target system, rather than using configure.

The target system may also require edits to the Makefiles, the simple Makefiles produced by Verilator presume the target system is the same type as the build system.

CMake

Verilator can be run using CMake, which takes care of both running Verilator and compiling the output. There is a CMake example in the examples/ directory. The following is a minimal CMakeLists.txt that would build the code listed in "EXAMPLE C++ EXECUTION":

```
project(cmake_example)
find_package(verilator HINTS $ENV{VERILATOR_ROOT})
add_executable(Vour sim_main.cpp)
verilate(Vour SOURCES our.v)
```

find_package will automatically find an installed copy of Verilator, or use a local build if VERILATOR_ROOT is set.

It is recommended to use CMake >= 3.12 and the Ninja generator, though other combinations should work. To build with CMake, change to the folder containing CMakeLists.txt and run:

```
mkdir build
cd build
cmake -GNinja ..
ninja
```

Or to build with your system default generator:

```
mkdir build
cd build
cmake ..
cmake --build .
```

If you're building the example you should have an executable to run:

```
./Vour
```

The package sets the CMake variables verilator_FOUND, VERILATOR_ROOT and VERILATOR_BIN to the appropriate values, and also creates a verilate() function. verilate() will automatically create custom commands to run Verilator and add the generated C++ sources to the target specified.

```
verilate(target SOURCES source ... [TOP_MODULE top] [PREFIX name]
         [TRACE] [TRACE_FST] [SYSTEMC] [COVERAGE]
         [INCLUDE_DIRS dir ...] [OPT_SLOW ...] [OPT_FAST ...]
         [OPT_GLOBAL ..] [DIRECTORY dir] [VERILATOR_ARGS ...])
```

Lowercase and ... should be replaced with arguments, the uppercase parts delimit the arguments and can be passed in any order, or left out entirely if optional.

verilate(target ...) can be called multiple times to add other verilog modules to an executable or library target.

When generating Verilated SystemC sources, you should also include the SystemC include directories and link to the SystemC libraries.

Verilator's CMake support provides a convenience function to automatically find and link to the SystemC library. It can be used as:

```
verilator_link_systemc(target)
```

where target is the name of your target.

The search paths can be configured by setting some variables:

- The variables SYSTEMC_INCLUDE and SYSTEMC_LIBDIR to give a direct path to the SystemC include an library path.

- SYSTEMC_ROOT to set the installation prefix of an installed SystemC library.

- SYSTEMC to set the installation prefix of an installed SystemC library (same as above).

- When using Accellera's SystemC with CMake support, a CMake target is available that simplifies the above steps. This will only work if the SystemC installation can be found by CMake. This can be configured by setting the CMAKE_PREFIX_PATH variable during CMake configuration.

Don't forget to set the same C++ standard for the Verilated sources as the SystemC library. This can be specified using the SYSTEMC_CXX_FLAGS environment variable.

target

Name of a target created by add_executable or add_library.

SOURCES

List of Verilog files to Verilate. Must have at least one file.

PREFIX

Optional. Sets the Verilator output prefix. Defaults to the name of the first source file with a "V" prepended. Must be unique in each call to verilate(), so this is necessary if you build a module multiple times with different parameters. Must be a valid C++ identifier, i.e. contains no white space and only characters A-Z, a-z, 0-9 or _.

TOP_MODULE

Optional. Sets the name of the top module. Defaults to the name of the first file in the SOURCES array.

TRACE

Optional. Enables VCD tracing if present, equivalent to "VERILATOR_ARGS --trace".

TRACE_FST

Optional. Enables FST tracing if present, equivalent to "VERILATOR_ARGS --trace-fst".

SYSTEMC

Optional. Enables SystemC mode, defaults to C++ if not specified.

COVERAGE

Optional. Enables coverage if present, equivalent to "VERILATOR_ARGS --coverage"

INCLUDE_DIRS

Optional. Sets directories that Verilator searches (same as -y).

OPT_SLOW

Optional. Set compiler flags for the slow path. You may want to reduce the optimization level to improve compile times with large designs.

OPT_FAST

Optional. Set compiler flags for the fast path.

OPT_GLOBAL

Optional. Set compiler flags for the common runtime library used by Verilated models.

DIRECTORY

Optional. Set the verilator output directory. It is preferable to use the default, which will avoid collisions with other files.

VERILATOR_ARGS

Optional. Extra arguments to Verilator. Do not specify --Mdir or --prefix here, use DIRECTORY or PREFIX.

18 HIERARCHICAL VERILATION

Large designs may take long (e.g. 10+ minutes) and huge memory (e.g. 100+GB) to Verilate. One workaround is hierarchical Verilation, it is to Verilate each moderate size of building blocks and finally combine the building blocks. The building block will be called "hierarchy block" later.

The current hierarchical Verilation is based on protect-lib. Each hierarchy block is Verilated to protect-lib. User modules of the hierarchy blocks will see a tiny wrapper generated by protect-lib instead of the actual design.

Usage

All user need to do is mark moderate size of module as hierarchy block and pass --hierarchical option to verilator command. There are two ways to mark a module:

```
a) Write /* verilator hier_block */ metacomment in HDL code
    See L</"LANGUAGE EXTENSIONS"> for more detail.
b) add hier_block line in the configuration file.
   See C<hier_block> in L</"CONFIGURATION FILES"> for example.
```

You don't have to take care of hierarchical blocks when compiling Verilated C++ code. You can compile as usual. make -C obj_dir -f Vtop_module_name.mk

See also "Overlapping Verilation and compilation" to get executable quickly.

Limitations

Because hierarchy blocks are Verilated to protect-lib, they have some limitations such as:

```
The block cannot be accessed using dot (.) from upper module.
Signals in the block cannot be traced.
Modport cannot be used at the hirarchical block boundary.
```

On the other hand, the following usage is supported.

```
- Nested hierarchy block. A hierarchy block may instantiate other
  hierarchy blocks.
- Parameterized hierarchy block. Parameters of a hierarchy block can be
  overridden using #(.param_name(value)) construct.
```

The simulation speed may not be as fast as flat Verilation, in which all modules are globally scheduled.

Overlapping Verilation and compilation

Verilator needs to run N + 2 times in hierarchical Verilation, where N is the number of hierarchy blocks. 1 of 2 is for the top module which refers wrappers of all other hierarchy blocks. The other 1 of 2 is the initial run that searches modules marked with /*verilator hier_block*/ metacomment and creates a plan and write in (prefix)_hier.mk. This initial run internally invokes other N + 1 runs, so you don't have to care about these N + 1 times of run.

If -j <jobs> option is specified, Verilation for hierarchy blocks runs in parallel.

If --build option is also specified, C++ compilation also runs as soon as a hierarchy block is Verilated. C++ compilation and Verilation for other hierarchy blocks run simultaneously.

19 MULTITHREADING

Verilator supports multithreaded simulation models.

With --no-threads, the default, the model is not thread safe, and any use of more than one thread calling into one or even different Verilated models may result in unpredictable behavior. This gives the highest single thread performance.

With --threads 1, the generated model is single threaded, however the support libraries are multithread safe. This allows different instantiations of model(s) to potentially each be run under a different thread. All threading is the responsibility of the user's C++ testbench.

With --threads N, where N is at least 2, the generated model will be designed to run in parallel on N threads. The thread calling eval() provides one of those threads, and the generated model will create and manage the other N-1 threads. It's the client's responsibility not to oversubscribe the available CPU cores. Under CPU oversubscription, the Verilated model should not livelock nor deadlock, however, you can expect performance to be far worse than it would be with proper ratio of threads and CPU cores.

The remainder of this section describe behavior with --threads 1 or --threads N (not --no-threads).

VL_THREADED is defined when compiling a threaded Verilated module, causing the Verilated support classes become threadsafe.

The thread used for constructing a model must be the same thread that calls eval() into the model, this is called the "eval thread". The thread used to perform certain global operations such as saving and tracing must be done by a "main thread". In most cases the eval thread and main thread are the same thread (i.e. the user's top C++ testbench runs on a single thread), but this is not required.

The --trace-threads options can be used to produce trace dumps using multiple threads. If --trace-threads

is set without --threads, then --trace-threads will imply --threads 1, i.e.: the support libraries will be thread safe.

With --trace-threads 0, trace dumps are produced on the main thread. This again gives the highest single thread performance.

With --trace-threads N, where N is at least 1, N additional threads will be created and managed by the trace files (e.g.: VerilatedVcdC or VerilatedFstC), to generate the trace dump. The main thread will be released to proceed with execution as soon as possible, though some blocking of the main thread is still necessary while capturing the trace. Different trace formats can utilize a various number of threads. See the --trace-threads option.

When running a multithreaded model, the default Linux task scheduler often works against the model, by assuming threads are short lived, and thus often schedules threads using multiple hyperthreads within the same physical core. For best performance use the `numactl` program to (when the threading count fits) select unique physical cores on the same socket. The same applies for --trace-threads as well.

As an example, if a model was Verilated with "--threads 4", we consult

```
egrep 'processor|physical id|core id' /proc/cpuinfo
```

To select cores 0, 1, 2, and 3 that are all located on the same socket (0) but different physical cores. (Also useful is "numactl --hardware", or `lscpu` but those doesn't show Hyperthreading cores.) Then we execute

```
numactl -m 0 -C 0,1,2,3 -- verilated_executable_name
```

This will limit memory to socket 0, and threads to cores 0, 1, 2, 3, (presumably on socket 0) optimizing performance. Of course this must be adjusted if you want another simulator using e.g. socket 1, or if you Verilated with a different number of threads. To see what CPUs are actually used, use --prof-threads.

Multithreaded Verilog and Library Support

$display/$stop/$finish are delayed until the end of an eval() call in order to maintain ordering between threads. This may result in additional tasks completing after the $stop or $finish.

If using --coverage, the coverage routines are fully thread safe.

If using --dpi, Verilator assumes pure DPI imports are thread safe, balancing performance versus safety. See --threads-dpi.

If using --savable, the save/restore classes are not multithreaded and must be called only by the eval thread.

If using --sc, the SystemC kernel is not thread safe, therefore the eval thread and main thread must be the same.

If using --trace, the tracing classes must be constructed and called from the main thread.

If using --vpi, since SystemVerilog VPI was not architected by IEEE to be multithreaded, Verilator requires all VPI calls are only made from the main thread.

20 CONFIGURATION FILES

In addition to the command line, warnings and other features may be controlled by configuration files, typically named with the .vlt extension. An example:

```
`verilator_config
lint_off -rule WIDTH
lint_off -rule CASEX  -file "silly_vendor_code.v"
```

This disables WIDTH warnings globally, and CASEX for a specific file.

Configuration files are fed through the normal Verilog preprocessor prior to parsing, so `ifdefs, `defines, and comments may be used as if it were normal Verilog code.

Note that file or line-specific configuration only applies to files read after the configuration file. It is therefore recommended to pass the configuration file to Verilator as the first file.

The grammar of configuration commands is as follows:

`verilator_config

Take remaining text and treat it as Verilator configuration commands.

coverage_on [-file "<filename>" [-lines <line> [- <line>]]]

coverage_off [-file "<filename>" [-lines <line> [- <line>]]]

Enable/disable coverage for the specified filename (or wildcard with '*' or '?', or all files if omitted) and range of line numbers (or all lines if omitted). Often used to ignore an entire module for coverage analysis purposes.

lint_on [-rule <message>] [-file "<filename>" [-lines <line> [- <line>]]]

lint_off [-rule <message>] [-file "<filename>" [-lines <line> [- <line>]]]

lint_off [-rule <message>] [-file "<filename>"] [-match "<string>"]

Enable/disables the specified lint warning, in the specified filename (or wildcard with '*' or '?', or all files if omitted) and range of line numbers (or all lines if omitted).

With lint_off using '*' will override any lint_on directives in the source, i.e. the warning will still not be printed.

If the -rule is omitted, all lint warnings (see list in -Wno-lint) are enabled/disabled. This will override all later lint warning enables for the specified region.

If -match is set the linter warnings are matched against this (wildcard) string and are waived in case they match and iff rule and file (with wildcard) also match.

In previous versions -rule was named -msg. The latter is deprecated, but still works with a deprecation info, it may be removed in future versions.

tracing_on [-file "<filename>" [-lines <line> [- <line>]]]

tracing_off [-file "<filename>" [-lines <line> [- <line>]]]

Enable/disable waveform tracing for all future signals declared in the specified filename (or wildcard with '*' or '?', or all files if omitted) and range of line numbers (or all lines if omitted).

For tracing_off, instances below any module in the files/ranges specified will also not be traced.

clock_enable -module "<modulename>" -var "<signame>"

Indicate the signal is used to gate a clock, and the user takes responsibility for insuring there are no races related to it.

Same as /*verilator clock_enable*/, see §22 for more information and an example.

clocker -module "<modulename>" [-task "<taskname>"] -var "<signame>"

clocker -module "<modulename>" [-function "<funcname>"] -var "<signame>"

no_clocker -module "<modulename>" [-task "<taskname>"] -var "<signame>"

no_clocker -module "<modulename>" [-function "<funcname>"] -var "<signame>"

Indicate the signal is used as clock or not. This information is used by Verilator to mark the signal as clocker and propagate the clocker attribute automatically to derived signals. See `--clk` for more information.

Same as /*verilator clocker*/, see §22 for more information.

coverage_block_off -module "<modulename>" -block "<blockname>"

coverage_block_off -file "<filename>" -line <lineno>

Specifies the entire begin/end block should be ignored for coverage analysis purposes. Can either be specified as a named block or as a filename and line number.

Same as /*verilator coverage_block_off*/, see §22 for more information.

full_case -file "<filename>" -lines <lineno>

parallel_case -file "<filename>" -lines <lineno>

Same as "//synopsys full_case" and "//synopsys parallel_case". When these synthesis directives are discovered, Verilator will either formally prove the directive to be true, or failing that, will insert the appropriate code to detect failing cases at simulation runtime and print an "Assertion failed" error message.

hier_block -module "<modulename>"

Specifies that the module is a unit of hierarchical Verilation. Note that the setting is ignored unless --hierachical option is specified. See §18 for more information.

inline -module "<modulename>"

Specifies the module may be inlined into any modules that use this module. Same as /*verilator inline_module*/, and see that under §22 for more information.

isolate_assignments -module "<modulename>" [-task "<taskname>"] -var "<signame>"

isolate_assignments -module "<modulename>" [-function "<funcname>"] -var "<signame>"

isolate_assignments -module "<modulename>" -function "<fname>"

Used to indicate the assignments to this signal in any blocks should be isolated into new blocks. When there is a large combinatorial block that is resulting in a UNOPTFLAT warning, attaching this to the signal causing a false loop may clear up the problem.

Same as /* verilator isolate_assignments */, see §22 for more information.

no_inline -module "<modulename>"

Specifies the module should not be inlined into any modules that use this module. Same as /*verilator no_inline_module*/, and see that under §22 for more information.

no_inline [-module "<modulename>"] -task "<taskname>"

no_inline [-module "<modulename>"] -function "<funcname>"

Specify the function or task should not be inlined into where it is used. This may reduce the size of the final executable when a task is used a very large number of times. For this flag to work, the task and tasks below it must be pure; they cannot reference any variables outside the task itself.

Same as /*verilator no_inline_task*/, see §22 for more information.

public [-module "<modulename>"] [-task/-function "<taskname>"] -var "<signame>"

public_flat [-module "<modulename>"] [-task/-function "<taskname>"] -var "<signame>"

public_flat_rd [-module "<modulename>"] [-task/-function "<taskname>"] -var "<signame>"

public_flat_rw [-module "<modulename>"] [-task/-function "<taskname>"] -var "<signame>" "@(edge)"

Sets the variable to be public. Same as /*verilator public*/ or /*verilator public_flat*/ etc, see those under §22 for more information.

sc_bv -module "<modulename>" [-task "<taskname>"] -var "<signame>"

sc_bv -module "<modulename>" [-function "<funcname>"] -var "<signame>"

Sets the port to be of sc_bv<*width*> type, instead of bool, vluint32_t or vluint64_t. Same as /*verilator sc_bv*/, see that under §22 for more information.

sformat [-module "<modulename>"] [-task "<taskname>"] -var "<signame>"

sformat [-module "<modulename>"] [-function "<funcname>"] -var "<signame>"

Must be applied to the final argument of type "input string" of a function or task to indicate the function or task should pass all remaining arguments through $sformatf. This allows creation of DPI functions with $display like behavior. See the test_regress/t/t_dpi_display.v file for an example.

Same as /*verilator sformat*/, see §22 for more information.

split_var [-module "<modulename>"] [-task "<taskname>"] -var "<varname>"

split_var [-module "<modulename>"] [-function "<funcname>"] -var "<varname>"

Break the variable into multiple pieces typically to resolve UNOPTFLAT performance issues. Typically the variables to attach this to are recommended by Verilator itself, see UNOPTFLAT.

Same as /*verilator split_var*/, see §22 for more information.

21 LANGUAGE STANDARD SUPPORT

Verilog 2001 (IEEE 1364-2001) Support

Verilator supports most Verilog 2001 language features. This includes signed numbers, "always @*", generate statements, multidimensional arrays, localparam, and C-style declarations inside port lists.

Verilog 2005 (IEEE 1364-2005) Support

Verilator supports most Verilog 2005 language features. This includes the ‘begin_keywords and ‘end_keywords compiler directives, $clog2, and the uwire keyword.

SystemVerilog 2005 (IEEE 1800-2005) Support

Verilator supports ==? and !=? operators, ++ and -- in some contexts, $bits, $countbits, $countones, $error, $fatal, $info, $isunknown, $onehot, $onehot0, $unit, $warning, always_comb, always_ff, always_latch, bit, byte, chandle, const, do-while, enum, export, final, import, int, interface, logic, longint, modport, package, program, shortint, struct, time, typedef, union, var, void, priority case/if, and unique case/if.

It also supports .name and .* interconnection.

Verilator partially supports concurrent assert and cover statements; see the enclosed coverage tests for the syntax which is allowed.

SystemVerilog 2012 (IEEE 1800-2012) Support

Verilator implements a full SystemVerilog 2012 preprocessor, including function call-like preprocessor defines, default define arguments, `__FILE__, `__LINE__ and `undefineall.

Verilator currently has some support for SystemVerilog synthesis constructs. As SystemVerilog features enter common usage they are added; please file a bug if a feature you need is missing.

SystemVerilog 2017 (IEEE 1800-2017) Support

Verilator supports the 2017 "for" loop constructs, and several minor cleanups made in 1800-2017.

Verilog AMS Support

Verilator implements a very small subset of Verilog AMS (Verilog Analog and Mixed-Signal Extensions) with the subset corresponding to those VMS keywords with near equivalents in the Verilog 2005 or SystemVerilog 2009 languages.

AMS parsing is enabled with "--language VAMS" or "--language 1800+VAMS".

At present Verilator implements ceil, exp, floor, ln, log, pow, sqrt, string, and wreal.

Synthesis Directive Assertion Support

With the --assert switch, Verilator reads any "//synopsys full_case" or "//synopsys parallel_case" directives. The same applies to any "//ambit synthesis", "//cadence" or "//pragma" directives of the same form.

When these synthesis directives are discovered, Verilator will either formally prove the directive to be true, or failing that, will insert the appropriate code to detect failing cases at simulation runtime and print an "Assertion failed" error message.

Verilator likewise also asserts any "unique" or "priority" SystemVerilog keywords on case statement, as well as "unique" on if statements. However, "priority if" is currently simply ignored.

22 LANGUAGE EXTENSIONS

The following additional constructs are the extensions Verilator supports on top of standard Verilog code. Using these features outside of comments or 'ifdef's may break other tools.

'__FILE__

The __FILE__ define expands to the current filename as a string, like C++'s __FILE__. This was incorporated into to the 1800-2009 standard (but supported by Verilator since 2006!)

'__LINE__

The __LINE__ define expands to the current filename as a string, like C++'s __LINE__. This was incorporated into to the 1800-2009 standard (but supported by Verilator since 2006!)

'error *string*

This will report an error when encountered, like C++'s #error.

$c(*string*, ...);

The string will be embedded directly in the output C++ code at the point where the surrounding Verilog code is compiled. It may either be a standalone statement (with a trailing ; in the string), or a function that returns up to a 32-bit number (without a trailing ;). This can be used to call C++ functions from your Verilog code.

String arguments will be put directly into the output C++ code. Expression arguments will have the code to evaluate the expression inserted. Thus to call a C++ function, $c("func(",a,")") will result in 'func(a)' in the output C++ code. For input arguments, rather than hard-coding variable names in the string $c("func(a)"), instead pass the variable as an expression $c("func(",a,")"). This will allow the call to work inside Verilog functions where the variable is flattened out, and also enable other optimizations.

If you will be reading or writing any Verilog variables inside the C++ functions, the Verilog signals must be declared with /*verilator public*/.

You may also append an arbitrary number to $c, generally the width of the output. [signal_32_bits = $c32("...");] This allows for compatibility with other simulators which require a differently named PLI function name for each different output width.

$display, $write, $fdisplay, $fwrite, $sformat, $swrite

Format arguments may use C fprintf sizes after the % escape. Per the Verilog standard, %x prints a number with the natural width, and %0x prints a number with minimum width. Verilator extends this so %5x prints 5 digits per the C standard (this is unspecified in Verilog, but was incorporated into the 1800-2009).

'coverage_block_off

Specifies the entire begin/end block should be ignored for coverage analysis. Must be inside a code block, e.g. within a begin/end pair. Same as /* verilator coverage_block_off */ and `coverage_block_off` in §20.

'systemc_header

Take remaining text up to the next 'verilog or 'systemc_... mode switch and place it verbatim into the output .h file's header. Must be placed as a module item, e.g. directly inside a module/endmodule pair. Despite the name of this macro, this also works in pure C++ code.

'systemc_ctor

Take remaining text up to the next 'verilog or 'systemc_... mode switch and place it verbatim into the C++ class constructor. Must be placed as a module item, e.g. directly inside a module/endmodule pair. Despite the name of this macro, this also works in pure C++ code.

‘systemc_dtor

Take remaining text up to the next ‘verilog or ‘systemc_... mode switch and place it verbatim into the C++ class destructor. Must be placed as a module item, e.g. directly inside a module/endmodule pair. Despite the name of this macro, this also works in pure C++ code.

‘systemc_interface

Take remaining text up to the next ‘verilog or ‘systemc_... mode switch and place it verbatim into the C++ class interface. Must be placed as a module item, e.g. directly inside a module/endmodule pair. Despite the name of this macro, this also works in pure C++ code.

‘systemc_imp_header

Take remaining text up to the next ‘verilog or ‘systemc_... mode switch and place it verbatim into the header of all files for this C++ class implementation. Must be placed as a module item, e.g. directly inside a module/endmodule pair. Despite the name of this macro, this also works in pure C++ code.

‘systemc_implementation

Take remaining text up to the next ‘verilog or ‘systemc_... mode switch and place it verbatim into a single file of the C++ class implementation. Must be placed as a module item, e.g. directly inside a module/endmodule pair. Despite the name of this macro, this also works in pure C++ code.

If you will be reading or writing any Verilog variables in the C++ functions, the Verilog signals must be declared with /*verilator public*/. See also the public task feature; writing an accessor may result in cleaner code.

‘SYSTEMVERILOG

The SYSTEMVERILOG, SV_COV_START and related standard defines are set by default when --language is 1800-*.

‘VERILATOR

‘verilator

‘verilator3

The VERILATOR, verilator and verilator3 defines are set by default so you may ‘ifdef around tool specific constructs.

‘verilator_config

Take remaining text up to the next ‘verilog mode switch and treat it as Verilator configuration commands.

‘verilog

Switch back to processing Verilog code after a ‘systemc_... mode switch. The Verilog code returns to the last language mode specified with ‘begin_keywords, or SystemVerilog if none was specified.

/*verilator clock_enable*/

Used after a signal declaration to indicate the signal is used to gate a clock, and the user takes responsibility for insuring there are no races related to it. (Typically by adding a latch, and running static timing analysis.) For example:

```
reg enable_r /*verilator clock_enable*/;
wire gated_clk = clk & enable_r;
always_ff @ (posedge clk)
   enable_r <= enable_early;
```

The clock_enable attribute will cause the clock gate to be ignored in the scheduling algorithm, sometimes required for correct clock behavior, and always improving performance. It's also a good idea to enable the IMPERFECTSCH warning, to ensure all clock enables are properly recognized.

Same as `clock_enable` in configuration files, see §20 for more information.

/*verilator clocker*/

/*verilator no_clocker*/

Used after a signal declaration to indicate the signal is used as clock or not. This information is used by Verilator to mark the signal as clocker and propagate the clocker attribute automatically to derived signals. See `--clk` for more information.

Same as `clocker` and `no_clocker` in configuration files, see §20 for more information.

/*verilator coverage_block_off*/

Specifies the entire begin/end block should be ignored for coverage analysis purposes.

Same as `coverage_block_off` in configuration files, see §20 for more information.

/*verilator coverage_off*/

Specifies that following lines of code should have coverage disabled. Often used to ignore an entire module for coverage analysis purposes.

/*verilator coverage_on*/

Specifies that following lines of code should have coverage re-enabled (if appropriate --coverage flags are passed) after being disabled earlier with /*verilator coverage_off*/.

/*verilator hier_block*/

Specifies that the module is a unit of hierarchical Verilation. This metacomment must be between "module module_name(...);" and "endmodule". The module will not be inlined nor uniquified for each instance in hierarchical Verilation. Note that the metacomment is ignored unless --hierachical option is specified.

See §18 for more information.

/*verilator inline_module*/

Specifies the module the comment appears in may be inlined into any modules that use this module. This is useful to speed up simulation runtime. Note if using "--public" that signals under inlined submodules will be named *submodule*__DOT__*subsignal* as C++ does not allow "." in signal names.

Same as `inline` in configuration files, see §20 for more information.

/*verilator isolate_assignments*/

Used after a signal declaration to indicate the assignments to this signal in any blocks should be isolated into new blocks. When there is a large combinatorial block that is resulting in a UNOPTFLAT warning, attaching this to the signal causing a false loop may clear up the problem.

IE, with the following

```
reg splitme /* verilator isolate_assignments*/;
// Note the placement of the semicolon above
always @* begin
  if (....) begin
     splitme = ....;
     other assignments
  end
end
```

Verilator will internally split the block that assigns to "splitme" into two blocks:

It would then internally break it into (sort of):

```
// All assignments excluding those to splitme
always @* begin
  if (....) begin
     other assignments
  end
end
// All assignments to splitme
always @* begin
  if (....) begin
     splitme = ....;
  end
end
```

Same as `isolate_assignments` in configuration files, see §20 for more information.

/*verilator lint_off *msg/**

Disable the specified warning message for any warnings following the comment.

/*verilator lint_on *msg/**

Re-enable the specified warning message for any warnings following the comment.

/*verilator lint_restore*/

After a /*verilator lint_save*/, pop the stack containing lint message state. Often this is useful at the bottom of include files.

/*verilator lint_save*/

Push the current state of what lint messages are turned on or turned off to a stack. Later meta-comments may then lint_on or lint_off specific messages, then return to the earlier message state by using /*verilator lint_restore*/. For example:

```
// verilator lint_save
// verilator lint_off SOME_WARNING
...  // code needing SOME_WARNING turned off
// verilator lint_restore
```

If SOME_WARNING was on before the lint_off, it will now be restored to on, and if it was off before the lint_off it will remain off.

/*verilator no_inline_module*/

Specifies the module the comment appears in should not be inlined into any modules that use this module.

Same as `no_inline` in configuration files, see §20 for more information.

/*verilator no_inline_task*/

Used in a function or task variable definition section to specify the function or task should not be inlined into where it is used. This may reduce the size of the final executable when a task is used a very large number of times. For this flag to work, the task and tasks below it must be pure; they cannot reference any variables outside the task itself.

Same as `no_inline` in configuration files, see §20 for more information.

/*verilator public*/ (parameter)

Used after a parameter declaration to indicate the emitted C code should have the parameter values visible. Due to C++ language restrictions, this may only be used on 64-bit or narrower integral enumerations.

```
parameter [2:0] PARAM /*verilator public*/ = 2'b0;
```

/*verilator public*/ (typedef enum)

Used after an enum typedef declaration to indicate the emitted C code should have the enum values visible. Due to C++ language restrictions, this may only be used on 64-bit or narrower integral enumerations.

```
typedef enum logic [2:0] { ZERO = 3'b0 } pub_t /*verilator public*/;
```

/*verilator public*/ (variable)

Used after an input, output, register, or wire declaration to indicate the signal should be declared so that C code may read or write the value of the signal. This will also declare this module public, otherwise use /*verilator public_flat*/.

Instead of using public variables, consider instead making a DPI or public function that accesses the variable. This is nicer as it provides an obvious entry point that is also compatible across simulators.

Same as `public` in configuration files, see §20 for more information.

/*verilator public*/ (task/function)

Used inside the declaration section of a function or task declaration to indicate the function or task should be made into a C++ function, public to outside callers. Public tasks will be declared as a void C++ function, public functions will get the appropriate non-void (bool, uint32_t, etc) return type. Any input arguments will become C++ arguments to the function. Any output arguments will become C++ reference arguments. Any local registers/integers will become function automatic variables on the stack.

Wide variables over 64 bits cannot be function returns, to avoid exposing complexities. However, wide variables can be input/outputs; they will be passed as references to an array of 32-bit numbers.

Generally, only the values of stored state (flops) should be written, as the model will NOT notice changes made to variables in these functions. (Same as when a signal is declared public.)

You may want to use DPI exports instead, as it's compatible with other simulators.

Same as `public` in configuration files, see §20 for more information.

/*verilator public_flat*/ (variable)

Used after an input, output, register, or wire declaration to indicate the signal should be declared so that C code may read or write the value of the signal. This will not declare this module public, which means the name of the signal or path to it may change based upon the module inlining which takes place.

Same as `public_flat` in configuration files, see §20 for more information.

/*verilator public_flat_rd*/ (variable)

Used after an input, output, register, or wire declaration to indicate the signal should be declared public_flat (see above), but read-only.

Same as `public_flat_rd` in configuration files, see §20 for more information.

/*verilator public_flat_rw @(<edge_list>) */ (variable)

Used after an input, output, register, or wire declaration to indicate the signal should be declared public_flat_rd (see above), and also writable, where writes should be considered to have the timing specified by the given sensitivity edge list. Set for all variables, ports and wires using the --public-flat-rw switch.

Same as `public_flat_rw` in configuration files, see §20 for more information.

/*verilator public_module*/

Used after a module statement to indicate the module should not be inlined (unless specifically requested) so that C code may access the module. Verilator automatically sets this attribute when the module contains any public signals or 'systemc_ directives. Also set for all modules when using the --public switch.

Same as `public` in configuration files, see §20 for more information.

/*verilator sc_clock*/

Deprecated and ignored. Previously used after an input declaration to indicate the signal should be declared in SystemC as a sc_clock instead of a bool. This was needed in SystemC 1.1 and 1.2 only; versions 2.0 and later do not require clock pins to be sc_clocks and this is no longer needed and is ignored.

/*verilator sc_bv*/

Used after a port declaration. It sets the port to be of sc_bv<*width*> type, instead of bool, vluint32_t or vluint64_t. This may be useful if the port width is parameterized and the instantiating C++ code wants to always have a sc_bv so it can accept any width. In general you should avoid using this attribute when not necessary as with increasing usage of sc_bv the performance decreases significantly.

Same as `sc_bv` in configuration files, see §20 for more information.

/*verilator sformat*/

Attached to the final argument of type "input string" of a function or task to indicate the function or task should pass all remaining arguments through $sformatf. This allows creation of DPI functions with $display like behavior. See the test_regress/t/t_dpi_display.v file for an example.

Same as `sformat` in configuration files, see §20 for more information.

/*verilator split_var*/

Attached to a variable or a net declaration to break the variable into multiple pieces typically to resolve UNOPTFLAT performance issues. Typically the variables to attach this to are recommended by Verilator itself, see UNOPTFLAT below.

For example, Verilator will internally convert a variable with the metacomment such as:

```
logic [7:0] x [0:1]  /*verilator split_var*/;
```

To:

```
logic [7:0] x__BRA__0__KET__ /*verilator split_var*/;
logic [7:0] x__BRA__1__KET__ /*verilator split_var*/;
```

Note that the generated packed variables retain the split_var metacomment because they may be split into further smaller pieces according to the access patterns.

This only supports unpacked arrays, packed arrays, and packed structs of integer types (reg, logic, bit, byte, int...); otherwise if a split was requested but cannot occur a SPLITVAR warning is issued. Splitting large arrays may slow down the Verilation speed, so use this only on variables that require it.

Same as `split_var` in configuration files, see §20 for more information.

/*verilator tag <text...>*/

Attached after a variable or structure member to indicate opaque (to Verilator) text that should be passed through to the XML output as a tag, for use by downstream applications.

/*verilator tracing_off*/

Disable waveform tracing for all future signals that are declared in this module, or instances below this module. Often this is placed just after a primitive's module statement, so that the entire module and instances below it are not traced.

/*verilator tracing_on*/

Re-enable waveform tracing for all future signals or instances that are declared.

23 LANGUAGE LIMITATIONS

There are some limitations and lack of features relative to the major closed-source simulators, by intent.

Synthesis Subset

Verilator supports the Synthesis subset with other verification constructs being added over time. Verilator also simulates events as Synopsys's Design Compiler would; namely given a block of the form:

```
always @ (x)   y = x & z;
```

This will recompute y when there is even a potential for change in x or a change in z, that is when the flops computing x or z evaluate (which is what Design Compiler will synthesize.) A compliant simulator would only calculate y if x changes. We recommend using always_comb to make the code run the same everywhere. Also avoid putting $displays in combo blocks, as they may print multiple times when not desired, even on compliant simulators as event ordering is not specified.

Signal Naming

To avoid conflicts with C symbol naming, any character in a signal name that is not alphanumeric nor a single underscore will be replaced by __0hh where hh is the hex code of the character. To avoid conflicts with Verilator's internal symbols, any double underscore are replaced with ___05F (5F is the hex code of an underscore.)

Bind

Verilator only supports "bind" to a target module name, not an instance path.

Class

Verilator class support is limited but in active development. Verilator supports members, and methods. Verilator does not support class static members, class extend, or class parameters.

Dotted cross-hierarchy references

Verilator supports dotted references to variables, functions and tasks in different modules. The portion before the dot must have a constant value; for example a[2].b is acceptable, while a[x].b is generally not.

References into generated and arrayed instances use the instance names specified in the Verilog standard; arrayed instances are named {instanceName}[{instanceNumber}] in Verilog, which becomes {instanceName}__BRA__{instanceNumber}__KET__ inside the generated C++ code.

Latches

Verilator is optimized for edge sensitive (flop based) designs. It will attempt to do the correct thing for latches, but most performance optimizations will be disabled around the latch.

Structures and Unions

Presently Verilator only supports packed structs and packed unions. Rand and randc tags on members are simply ignored. All structures and unions are represented as a single vector, which means that generating one member of a structure from blocking, and another from non-blocking assignments is unsupported.

Time

All delays (#) are ignored, as they are in synthesis.

Unknown states

Verilator is mostly a two state simulator, not a four state simulator. However, it has two features which uncover most initialization bugs (including many that a four state simulator will miss.)

Identity comparisons (=== or !==) are converted to standard ==/!= when neither side is a constant. This may make the expression yield a different result compared to a four state simulator. An === comparison to X will always be false, so that Verilog code which checks for uninitialized logic will not fire.

Assigning X to a variable will actually assign a constant value as determined by the --x-assign switch. This allows runtime randomization, thus if the value is actually used, the random value should cause downstream errors. Integers also get randomized, even though the Verilog 2001 specification says they initialize to zero. Note however that randomization happens at initialization time and hence during a single simulation run, the same constant (but random) value will be used every time the assignment is executed.

All variables, depending on --x-initial setting, are typically randomly initialized using a function. By running several random simulation runs you can determine that reset is working correctly. On the first run, have the function initialize variables to zero. On the second, have it initialize variables to one. On the third and following runs have it initialize them randomly. If the results match, reset works. (Note this is what the hardware will really do.) In practice, just setting all variables to one at startup finds most problems (since typically control signals are active-high).

--x-assign applies to variables explicitly initialized or assigned an X. Uninitialized clocks are initialized to zero, while all other state holding variables are initialized to a random value. Event driven simulators will generally trigger an edge on a transition from X to 1 (`posedge`) or X to 0 (`negedge`). However, by default, since clocks are initialized to zero, Verilator will not trigger an initial negedge. Some code (particularly for reset) may rely on X->0 triggering an edge. The --x-initial-edge switch enables this behavior. Comparing runs with and without this switch will find such problems.

Tri/Inout

Verilator converts some simple tristate structures into two state. Pullup, pulldown, bufif0, bufif1, notif0, notif1, pmos, nmos, tri0 and tri1 are also supported. Simple comparisons with === 1'bz are also supported.

An assignment of the form:

```
inout driver;
wire driver = (enable) ? output_value : 1'bz;
```

Will be converted to

```
input driver;       // Value being driven in from "external" drivers
output driver__en;  // True if driven from this module
output driver__out; // Value being driven from this module
```

External logic will be needed to combine these signals with any external drivers.

Tristate drivers are not supported inside functions and tasks; an inout there will be considered a two state variable that is read and written instead of a four state variable.

Functions & Tasks

All functions and tasks will be inlined (will not become functions in C.) The only support provided is for simple statements in tasks (which may affect global variables).

Recursive functions and tasks are not supported. All inputs and outputs are automatic, as if they had the Verilog 2001 "automatic" keyword prepended. (If you don't know what this means, Verilator will do what you probably expect -- what C does. The default behavior of Verilog is different.)

Generated Clocks

Verilator attempts to deal with generated and gated clocks correctly, however some cases cause problems in the scheduling algorithm which is optimized for performance. The safest option is to have all clocks as primary inputs to the model, or wires directly attached to primary inputs. For proper behavior clock enables may also need the /*verilator clock_enable*/ attribute.

Gate Primitives

The 2-state gate primitives (and, buf, nand, nor, not, or, xnor, xor) are directly converted to behavioral equivalents. The 3-state and MOS gate primitives are not supported. Tables are not supported.

Specify blocks

All specify blocks and timing checks are ignored. All min:typ:max delays use the typical value.

Array Initialization

When initializing a large array, you need to use non-delayed assignments. Verilator will tell you when this needs to be fixed; see the BLKLOOPINIT error for more information.

Array Out of Bounds

Writing a memory element that is outside the bounds specified for the array may cause a different memory element inside the array to be written instead. For power-of-2 sized arrays, Verilator will give a width warning and the address. For non-power-of-2-sizes arrays, index 0 will be written.

Reading a memory element that is outside the bounds specified for the array will give a width warning and wrap around the power-of-2 size. For non-power-of-2 sizes, it will return a unspecified constant of the appropriate width.

Assertions

Verilator is beginning to add support for assertions. Verilator currently only converts assertions to simple "if (...) error" statements, and coverage statements to increment the line counters described in the coverage section.

Verilator does not support SEREs yet. All assertion and coverage statements must be simple expressions that complete in one cycle.

Encrypted Verilog

Open source simulators like Verilator are unable to use encrypted RTL (i.e. IEEE P1735). Talk to your IP vendor about delivering IP blocks via Verilator's --protect-lib feature.

Language Keyword Limitations

This section describes specific limitations for each language keyword.

‘__FILE__, ‘__LINE__, ‘begin_keywords, ‘begin_keywords, ‘begin_keywords, ‘begin_keywords, ‘begin_keywords, ‘define, ‘else, ‘elsif, ‘end_keywords, ‘endif, ‘error, ‘ifdef, ‘ifndef, ‘include, ‘line, ‘systemc_ctor, ‘systemc_dtor, ‘systemc_header, ‘systemc_imp_header, ‘systemc_implementation, ‘systemc_interface, ‘undef, ‘verilog

Fully supported.

always, always_comb, always_ff, always_latch, and, assign, begin, buf, byte, case, casex, casez, default, defparam, do-while, else, end, endcase, endfunction, endgenerate, endmodule, endspecify, endtask, final, for, function, generate, genvar, if, initial, inout, input, int, integer, localparam, logic, longint, macromodule, module, nand, negedge, nor, not, or, output, parameter, posedge, reg, scalared, shortint, signed, supply0, supply1, task, time, tri, typedef, var, vectored, while, wire, xnor, xor

Generally supported.

++, -- operators

Increment/decrement can only be used as standalone statements or in certain limited cases.

’{} operator

Assignment patterns with order based, default, constant integer (array) or member identifier (struct/union) keys are supported. Data type keys and keys which are computed from a constant expression are not supported.

‘uselib

Uselib, a vendor specific library specification method, is ignored along with anything following it until the end of that line.

cast operator

Casting is supported only between simple scalar types, signed and unsigned, not arrays nor structs.

chandle

Treated as a "longint"; does not yet warn about operations that are specified as illegal on chandles.

disable

Disable statements may be used only if the block being disabled is a block the disable statement itself is inside. This was commonly used to provide loop break and continue functionality before SystemVerilog added the break and continue keywords.

inside

Inside expressions may not include unpacked array traversal or $ as an upper bound. Case inside and case matches are also unsupported.

interface

Interfaces and modports, including with generated data types are supported. Generate blocks around modports are not supported, nor are virtual interfaces nor unnamed interfaces.

shortreal

Short floating point (shortreal) numbers are converted to real. Most other simulators either do not support float, or convert likewise.

specify specparam

All specify blocks and timing checks are ignored.

uwire

Verilator does not perform warning checking on uwires, it treats the uwire keyword as if it were the normal wire keyword.

$bits, $countbits, $countones, $error, $fatal, $finish, $info, $isunknown, $onehot, $onehot0, $readmemb, $readmemh, $signed, $stime, $stop, $time, $unsigned, $warning.

Generally supported.

$dump/$dumpports and related

$dumpfile or $dumpports will create a VCD or FST file (which is based on the --trace argument given when the model was Verilated). This will take effect starting at the next eval() call. If you have multiple Verilated designs under the same C model, then this will dump signals only from the design containing the $dumpvars.

$dumpvars and $dumpports module identifier is ignored; the traced instances will always start at the top of the design. The levels argument is also ignored, use tracing_on/tracing_off pragmas instead.

$dumpportson/$dumpportsoff/$dumpportsall/$dumpportslimit filename argument is ignored, only a single trace file may be active at once.

$dumpall/$dumpportsall, $dumpon/$dumpportson, $dumpoff/$dumpportsoff, and $dumplimit/$dumpportlimit are currently ignored.

$exit, $finish, $stop

The rarely used optional parameter to $finish and $stop is ignored. $exit is aliased to $finish.

$fopen, $fclose, $fdisplay, $ferror, $feof, $fflush, $fgetc, $fgets, $fscanf, $fwrite, $fscanf, $sscanf

Generally supported.

$fullskew, $hold, $nochange, $period, $recovery, $recrem, $removal, $setup, $setuphold, $skew, $timeskew, $width

All specify blocks and timing checks are ignored.

$random, $urandom, $urandom_range

Use +verilator+seed argument to set the seed if there is no $random or $urandom optional argument to set the seed. There is one random seed per C thread, not per module for $random, nor per object for random stability of $urandom/$urandom_range.

$readmemb, $readmemh

Read memory commands should work properly. Note Verilator and the Verilog specification does not include support for readmem to multi-dimensional arrays.

$test$plusargs, $value$plusargs

Supported, but the instantiating C++/SystemC testbench must call

```
Verilated::commandArgs(argc, argv);
```

to register the command line before calling $test$plusargs or $value$plusargs.

24 ERRORS AND WARNINGS

Warnings may be disabled in three ways. First, when the warning is printed it will include a warning code. Simply surround the offending line with a lint_off/lint_on pair:

```
// verilator lint_off UNSIGNED
if (`DEF_THAT_IS_EQ_ZERO <= 3) $stop;
// verilator lint_on UNSIGNED
```

Second, warnings may be disabled using a configuration file with a lint_off command. This is useful when a script is suppressing warnings and the Verilog source should not be changed.

Warnings may also be globally disabled by invoking Verilator with the `-Wno-`*`warning`* switch. This should be avoided, as it removes all checking across the designs, and prevents other users from compiling your code without knowing the magic set of disables needed to successfully compile your design.

Error and Warning Format

Warnings and errors printed by Verilator always match this regular expression:

```
%(Error|Warning)(-[A-Z0-9_]+)?: ((\S+):(\d+):((\d+):)? )?.*
```

Errors and warning start with a percent sign (historical heritage from Digital Equipment Corporation). Some errors or warning have a code attached, with meanings described below. Some errors also have a filename, line number and optional column number (starting at column 1 to match GCC).

Following the error message, Verilator will typically show the user's source code corresponding to the error, prefixed by the line number and a " | ". Following this is typically an arrow and ~ pointing at the error on the source line directly above.

List of all warnings

ALWCOMBORDER

Warns that an always_comb block has a variable which is set after it is used. This may cause simulation-synthesis mismatches, as not all simulators allow this ordering.

```
always_comb begin
   a = b;
   b = 1;
end
```

Ignoring this warning will only suppress the lint check, it will simulate correctly.

ASSIGNIN

Error that an assignment is being made to an input signal. This is almost certainly a mistake, though technically legal.

```
input a;
assign a = 1'b1;
```

Ignoring this warning will only suppress the lint check, it will simulate correctly.

ASSIGNDLY

Warns that you have an assignment statement with a delayed time in front of it, for example:

```
a <= #100 b;
assign #100 a = b;
```

Ignoring this warning may make Verilator simulations differ from other simulators, however at one point this was a common style so disabled by default as a code style warning.

BLKANDNBLK

BLKANDNBLK is an error that a variable comes from a mix of blocking and non-blocking assignments.

This is not illegal in SystemVerilog, but a violation of good coding practice. Verilator reports this as an error, because ignoring this warning may make Verilator simulations differ from other simulators.

It is generally safe to disable this error (with a "// verilator lint_off BLKANDNBLK" metacomment or the -Wno-BLKANDNBLK option) when one of the assignments is inside a public task, or when the blocking and non-blocking assignments have non-overlapping bits and structure members.

Generally, this is caused by a register driven by both combo logic and a flop:

```
logic [1:0] foo;
always @ (posedge clk)  foo[0] <= ...
always @* foo[1] = ...
```

Simply use a different register for the flop:

```
logic [1:0] foo;
always @ (posedge clk)  foo_flopped[0] <= ...
always @* foo[0] = foo_flopped[0];
always @* foo[1] = ...
```

Or, this may also avoid the error:

```
logic [1:0] foo /*verilator split_var*/;
```

BLKSEQ

This indicates that a blocking assignment (=) is used in a sequential block. Generally non-blocking/delayed assignments (<=) are used in sequential blocks, to avoid the possibility of simulator races. It can be reasonable to do this if the generated signal is used ONLY later in the same block, however this style is generally discouraged as it is error prone.

```
always @ (posedge clk)  foo = ...
```

Disabled by default as this is a code style warning; it will simulate correctly.

BLKLOOPINIT

This indicates that the initialization of an array needs to use non-delayed assignments. This is done in the interest of speed; if delayed assignments were used, the simulator would have to copy large arrays every cycle. (In smaller loops, loop unrolling allows the delayed assignment to work, though it's a bit slower than a non-delayed assignment.) Here's an example

```
always @ (posedge clk)
    if (~reset_l) begin
        for (i=0; i<`ARRAY_SIZE; i++) begin
            array[i] = 0;  // Non-delayed for verilator
        end
```

This message is only seen on large or complicated loops because Verilator generally unrolls small loops. You may want to try increasing --unroll-count (and occasionally --unroll-stmts) which will raise the small loop bar to avoid this error.

BOUNDED

This indicates that bounded queues (e.g. "var name[$: 3]") are unsupported.

Ignoring this warning may make Verilator simulations differ from other simulators.

BSSPACE

Warns that a backslash is followed by a space then a newline. Likely the intent was to have a backslash directly followed by a newline (e.g. when making a 'define) and there's accidentally white space at the end of the line. If the space is not accidental, suggest removing the backslash in the code as it serves no function.

Ignoring this warning will only suppress the lint check, it will simulate correctly.

CASEINCOMPLETE

Warns that inside a case statement there is a stimulus pattern for which there is no case item specified. This is bad style, if a case is impossible, it's better to have a "default: $stop;" or just "default: ;" so that any design assumption violations will be discovered in simulation.

Ignoring this warning will only suppress the lint check, it will simulate correctly.

CASEOVERLAP

Warns that inside a case statement you have case values which are detected to be overlapping. This is bad style, as moving the order of case values will cause different behavior. Generally the values can be respecified to not overlap.

Ignoring this warning will only suppress the lint check, it will simulate correctly.

CASEWITHX

Warns that a case statement contains a constant with a `x`. Verilator is two-state so interpret such items as always false. Note a common error is to use a `X` in a case or casez statement item; often what the user instead intended is to use a casez with `?`.

Ignoring this warning will only suppress the lint check, it will simulate correctly.

CASEX

Warns that it is simply better style to use casez, and `?` in place of `x`'s. See http://www.sunburst-design.com/papers/CummingsSNUG1999Boston_FullParallelCase_rev1_1.pdf

Ignoring this warning will only suppress the lint check, it will simulate correctly.

CASTCONST

Warns that a dynamic cast ($cast) is unnecessary as the $cast will always succeed or fail. If it will always fail, the $cast is useless. If it will always succeed a static cast may be preferred.

Ignoring this warning will only suppress the lint check, it will simulate correctly. On other simulators, not fixing CASTCONST may result in decreased performance.

CDCRSTLOGIC

With --cdc only, warns that asynchronous flop reset terms come from other than primary inputs or flopped outputs, creating the potential for reset glitches.

CLKDATA

Warns that clock signal is mixed used with/as data signal. The checking for this warning is enabled only if user has explicitly marked some signal as clocker using command line option or in-source meta comment (see `--clk`).

The warning can be disabled without affecting the simulation result. But it is recommended to check the warning as this may degrade the performance of the Verilated model.

CMPCONST

Warns that you are comparing a value in a way that will always be constant. For example "X > 1" will always be true when X is a single bit wide.

Ignoring this warning will only suppress the lint check, it will simulate correctly.

COLONPLUS

Warns that a :+ is seen. Likely the intent was to use +: to select a range of bits. If the intent was a range that is explicitly positive, suggest adding a space, e.g. use ": +".

Ignoring this warning will only suppress the lint check, it will simulate correctly.

COMBDLY

Warns that you have a delayed assignment inside of a combinatorial block. Using delayed assignments in this way is considered bad form, and may lead to the simulator not matching synthesis. If this message is suppressed, Verilator, like synthesis, will convert this to a non-delayed assignment, which may result in logic races or other nasties. See http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA_rev1_2.pdf

Ignoring this warning may make Verilator simulations differ from other simulators.

CONTASSREG

Error that a continuous assignment is setting a reg. According to IEEE Verilog, but not SystemVerilog, a wire must be used as the target of continuous assignments.

This error is only reported when "--language 1364-1995", "--language 1364-2001", or "--language 1364-2005" is used.

Ignoring this error will only suppress the lint check, it will simulate correctly.

DECLFILENAME

Warns that a module or other declaration's name doesn't match the filename with path and extension stripped that it is declared in. The filename a modules/interfaces/programs is declared in should match the name of the module etc. so that -y directory searching will work. This warning is printed for only the first mismatching module in any given file, and -v library files are ignored.

Disabled by default as this is a code style warning; it will simulate correctly.

DEFPARAM

Warns that the "defparam" statement was deprecated in Verilog 2001 and all designs should now be using the #(...) format to specify parameters.

Disabled by default as this is a code style warning; it will simulate correctly.

DETECTARRAY

Error when Verilator tries to deal with a combinatorial loop that could not be flattened, and which involves a datatype which Verilator cannot handle, such as an unpacked struct or a large unpacked array. This typically occurs when -Wno-UNOPTFLAT has been used to override an UNOPTFLAT warning (see below).

The solution is to break the loop, as described for UNOPTFLAT.

DIDNOTCONVERGE

Error at simulation runtime when model did not properly settle.

Verilator sometimes has to evaluate combinatorial logic multiple times, usually around code where a UNOPTFLAT warning was issued, but disabled. For example:

```
always_comb b = ~a;
always_comb a = b
```

This code will toggle forever, and thus to prevent an infinite loop, the executable will give the didn't converge error.

To debug this, first review any UNOPTFLAT warnings that were ignored. Though typically it is safe to ignore UNOPTFLAT (at a performance cost), at the time of issuing a UNOPTFLAT Verilator did not know if the logic would eventually converge and assumed it would.

Next, run Verilator with --prof-cfuncs. Run make on the generated files with "CPP_FLAGS=-DVL_DEBUG", to allow enabling simulation runtime debug messages. Rerun the test. Now just before the convergence error you should see additional output similar to this:

```
CHANGE: filename.v:1: b
CHANGE: filename.v:2: a
```

This means that signal b and signal a keep changing, inspect the code that modifies these signals. Note if many signals are getting printed then most likely all of them are oscillating. It may also be that e.g. "a" may be oscillating, then "a" feeds signal "c" which then is also reported as oscillating.

Finally, rare, more difficult cases can be debugged like a "C" program; either enter GDB and use its tracing facilities, or edit the generated C++ code to add appropriate prints to see what is going on.

ENDLABEL

Warns that a label attached to a "end"-something statement does not match the label attached to the block start.

Ignoring this warning will only suppress the lint check, it will simulate correctly.

GENCLK

Deprecated and no longer used as a warning. Used to indicate that the specified signal was is generated inside the model, and also being used as a clock.

HIERBLOCK

Warns that the top module is marked as a hierarchy block by hier_block metacomment, which is not legal. This setting on the top module will be ignored. =item IFDEPTH

Warns that if/if else statements have exceeded the depth specified with --if-depth, as they are likely to result in slow priority encoders. Statements below unique and priority if statements are ignored. Solutions include changing the code to a case statement, or a SystemVerilog 'unique if' or 'priority if'.

Disabled by default as this is a code style warning; it will simulate correctly.

IGNOREDRETURN

Warns that a non-void function is being called as a task, and hence the return value is being ignored.

This warning is required by IEEE. The portable way to suppress this warning (in SystemVerilog) is to use a void cast, e.g.

```
void'(function_being_called_as_task());
```

Ignoring this warning will only suppress the lint check, it will simulate correctly.

IMPERFECTSCH

Warns that the scheduling of the model is not absolutely perfect, and some manual code edits may result in faster performance. This warning defaults to off, is not part of -Wall, and must be turned on explicitly before the top module statement is processed.

IMPLICIT

Warns that a wire is being implicitly declared (it is a single bit wide output from a sub-module.) While legal in Verilog, implicit declarations only work for single bit wide signals (not buses), do not allow using a signal before it is implicitly declared by an instance, and can lead to dangling nets. A better option is the /*AUTOWIRE*/ feature of Verilog-Mode for Emacs, available from https://www.veripool.org/verilog-mode

Ignoring this warning will only suppress the lint check, it will simulate correctly.

IMPORTSTAR

Warns that an "import *package*::*" statement is in $unit scope. This causes the imported symbols to pollute the global namespace, defeating much of the purpose of having a package. Generally "import ::*" should only be used inside a lower scope such as a package or module.

Disabled by default as this is a code style warning; it will simulate correctly.

IMPURE

Warns that a task or function that has been marked with /*verilator no_inline_task*/ references variables that are not local to the task. Verilator cannot schedule these variables correctly.

Ignoring this warning may make Verilator simulations differ from other simulators.

INCABSPATH

Warns that an 'include filename specifies an absolute path. This means the code will not work on any other system with a different file system layout. Instead of using absolute paths, relative paths (preferably without any directory specified whatsoever) should be used, and +incdir used on the command line to specify the top include source directories.

Disabled by default as this is a code style warning; it will simulate correctly.

INFINITELOOP

Warns that a while or for statement has a condition that is always true. and thus results in an infinite loop if the statement ever executes.

This might be unintended behavior if the loop body contains statements that in other simulators would make time pass, which Verilator is ignoring due to e.g. STMTDLY warnings being disabled.

Ignoring this warning will only suppress the lint check, it will simulate correctly (i.e. hang due to the infinite loop).

INITIALDLY

Warns that you have a delayed assignment inside of an initial or final block. If this message is suppressed, Verilator will convert this to a non-delayed assignment. See also the COMBDLY warning.

Ignoring this warning may make Verilator simulations differ from other simulators.

INSECURE

Warns that the combination of options selected may be defeating the attempt to protect/obscure identifiers or hide information in the model. Correct the options provided, or inspect the output code to see if the information exposed is acceptable.

Ignoring this warning will only suppress the lint check, it will simulate correctly.

LATCH

Warns that a signal is not assigned in all control paths of a combinational always block, resulting in the inference of a latch. For intentional latches, consider using the always_latch (SystemVerilog) keyword instead. The warning may be disabled with a lint_off pragma around the always block.

Ignoring this warning will only suppress the lint check, it will simulate correctly.

LITENDIAN

Warns that a packed vector is declared with little endian bit numbering (i.e. [0:7]). Big endian bit numbering is now the overwhelming standard, and little numbering is now thus often due to simple oversight instead of intent.

Also warns that an instance is declared with little endian range (i.e. [0:7] or [7]) and is connected to a N-wide signal. Based on IEEE the bits will likely be backwards from what you expect (i.e. instance [0] will connect to signal bit [N-1] not bit [0]).

Ignoring this warning will only suppress the lint check, it will simulate correctly.

MODDUP

Warns that a module has multiple definitions. Generally this indicates a coding error, or a mistake in a library file and it's good practice to have one module per file (and only put each file once on the command line) to avoid these issues. For some gate level netlists duplicates are sometimes unavoidable, and MODDUP should be disabled.

Ignoring this warning will cause the more recent module definition to be discarded.

MULTIDRIVEN

Warns that the specified signal comes from multiple always blocks. This is often unsupported by synthesis tools, and is considered bad style. It will also cause longer simulation runtimes due to reduced optimizations.

Ignoring this warning will only slow simulations, it will simulate correctly.

MULTITOP

Warns that there are multiple top level modules, that is modules not instantiated by any other module, and both modules were put on the command line (not in a library). Three likely cases:

1. A single module is intended to be the top. This warning then occurs because some low level instance is being read in, but is not really needed as part of the design. The best solution for this situation is to ensure that only the top module is put on the command line without any flags, and all remaining library files are read in as libraries with -v, or are automatically resolved by having filenames that match the module names.

2. A single module is intended to be the top, the name of it is known, and all other modules should be ignored if not part of the design. The best solution is to use the --top option to specify the top module's name. All other modules that are not part of the design will be for the most part ignored (they must be clean in syntax and their contents will be removed as part of the Verilog module elaboration process.)

3. Multiple modules are intended to be design tops, e.g. when linting a library file. As multiple modules are desired, disable the MULTITOP warning. All input/outputs will go uniquely to each module, with any conflicting and identical signal names being made unique by adding a prefix based on the top module name followed by __02E (a Verilator-encoded ASCII '.'). This renaming is done even if the two modules' signals seem identical, e.g. multiple modules with a "clk" input.

NOLATCH

Warns that no latch was detected in an always_latch block. The warning may be disabled with a lint_off pragma around the always block, but recoding using a regular always may be more appropriate.

Ignoring this warning will only suppress the lint check, it will simulate correctly.

PINCONNECTEMPTY

Warns that an instance has a pin which is connected to .pin_name(), e.g. not another signal, but with an explicit mention of the pin. It may be desirable to disable PINCONNECTEMPTY, as this indicates intention to have a no-connect.

Disabled by default as this is a code style warning; it will simulate correctly.

PINMISSING

Warns that a module has a pin which is not mentioned in an instance. If a pin is not missing it should still be specified on the instance declaration with a empty connection, using "(.pin_name())".

Ignoring this warning will only suppress the lint check, it will simulate correctly.

PINNOCONNECT

Warns that an instance has a pin which is not connected to another signal.

Disabled by default as this is a code style warning; it will simulate correctly.

PKGNODECL

Error that a package/class appears to have been referenced that has not yet been declared. According to IEEE 1800-2017 26.3 all packages must be declared before being used.

PROCASSWIRE

Error that a procedural assignment is setting a wire. According to IEEE, a var/reg must be used as the target of procedural assignments.

RANDC

Warns that the 'randc' keyword is currently unsupported, and that it is being converted to 'rand'.

REALCVT

Warns that a real number is being implicitly rounded to an integer, with possible loss of precision.

REDEFMACRO

Warns that you have redefined the same macro with a different value, for example:

```
‘define MACRO def1
//...
‘define MACRO otherdef
```

The best solution is to use a different name for the second macro. If this is not possible, add a undef to indicate the code is overriding the value:

```
‘define MACRO def1
//...
‘undef MACRO
‘define MACRO otherdef
```

SELRANGE

Warns that a selection index will go out of bounds:

```
wire vec[6:0];
initial out = vec[7];  // There is no 7
```

Verilator will assume zero for this value, instead of X. Note that in some cases this warning may be false, when a condition upstream or downstream of the access means the access out of bounds will never execute or be used.

```
wire vec[6:0];
initial begin
    seven = 7;
    ...
    if (seven != 7) out = vec[seven];  // Never will use vec[7]
```

SHORTREAL

Warns that Verilator does not support "shortreal" and they will be automatically promoted to "real". The recommendation is to replace any "shortreal" in the code with "real", as "shortreal" is not widely supported across industry tools.

Ignoring this warning may make Verilator simulations differ from other simulators, if the increased precision of real affects your model or DPI calls.

SPLITVAR

Warns that a variable with a `split_var` metacomment was not split. Some possible reasons for this are:

* The datatype of the variable is not supported for splitting. (e.g. is a real).

* The access pattern of the variable can not be determined statically. (e.g. is accessed as a memory).

* The index of the array exceeds the array size.

* The variable is accessed from outside using dotted reference. (e.g. top.instance0.variable0 = 1).

* The variable is not declared in a module, but in a package or an interface.

* The variable is a parameter, localparam, genvar, or queue.

* The variable is tristate or bidirectional. (e.g. inout or ref).

STMTDLY

Warns that you have a statement with a delayed time in front of it, for example:

```
#100 $finish;
```

Ignoring this warning may make Verilator simulations differ from other simulators.

SYMRSVDWORD

Warning that a symbol matches a C++ reserved word and using this as a symbol name would result in odd C++ compiler errors. You may disable this warning, but the symbol will be renamed by Verilator to avoid the conflict.

SYNCASYNCNET

Warns that the specified net is used in at least two different always statements with posedge/negedges (i.e. a flop). One usage has the signal in the sensitivity list and body, probably as an async reset, and the other usage has the signal only in the body, probably as a sync reset. Mixing sync and async resets is usually a mistake. The warning may be disabled with a lint_off pragma around the net, or either flopped block.

Disabled by default as this is a code style warning; it will simulate correctly.

TASKNSVAR

Error when a call to a task or function has an inout from that task tied to a non-simple signal. Instead connect the task output to a temporary signal of the appropriate width, and use that signal to set the appropriate expression as the next statement. For example:

```
task foo(inout sig); ... endtask
always @* begin
     foo(bus_we_select_from[2]);  // Will get TASKNSVAR error
end
```

Change this to:

```
reg foo_temp_out;
always @* begin
     foo(foo_temp_out);
     bus_we_select_from[2] = foo_temp_out;
end
```

Verilator doesn't do this conversion for you, as some more complicated cases would result in simulator mismatches.

TICKCOUNT

Warns that the number of ticks to delay a $past variable is greater than 10. At present Verilator effectively creates a flop for each delayed signals, and as such any large counts may lead to large design size increases.

Ignoring this warning will only slow simulations, it will simulate correctly.

TIMESCALEMOD

Error that 'timescale is used in some but not all modules. Recommend using --timescale argument, or in front of all modules use:

```
`include "timescale.vh"
```

Then in that file set the timescale.

This is an error due to IEEE specifications, but it may be disabled similar to warnings. Ignoring this error may result in a module having an unexpected timescale.

UNDRIVEN

Warns that the specified signal has no source. Verilator is fairly liberal in the usage calculations; making a signal public, or setting only a single array element marks the entire signal as driven.

Disabled by default as this is a code style warning; it will simulate correctly.

UNOPT

Warns that due to some construct, optimization of the specified signal or block is disabled. The construct should be cleaned up to improve simulation performance.

A less obvious case of this is when a module instantiates two submodules. Inside submodule A, signal I is input and signal O is output. Likewise in submodule B, signal O is an input and I is an output. A loop exists and a UNOPT warning will result if AI & AO both come from and go to combinatorial blocks in both submodules, even if they are unrelated always blocks. This affects performance because Verilator would have to evaluate each submodule multiple times to stabilize the signals crossing between the modules.

Ignoring this warning will only slow simulations, it will simulate correctly.

UNOPTFLAT

Warns that due to some construct, optimization of the specified signal is disabled. The signal reported includes a complete scope to the signal; it may be only one particular usage of a multiply instantiated block. The construct should be cleaned up to improve simulation performance; two times better performance may be possible by fixing these warnings.

Unlike the UNOPT warning, this occurs after flattening the netlist, and indicates a more basic problem, as the less obvious case described under UNOPT does not apply.

Often UNOPTFLAT is caused by logic that isn't truly circular as viewed by synthesis which analyzes interconnection per-bit, but is circular to simulation which analyzes per-bus:

```
wire [2:0] x = {x[1:0], shift_in};
```

This statement needs to be evaluated multiple times, as a change in "shift_in" requires "x" to be computed 3 times before it becomes stable. This is because a change in "x" requires "x" itself to change value, which causes the warning.

For significantly better performance, split this into 2 separate signals:

```
wire [2:0] xout = {x[1:0], shift_in};
```

and change all receiving logic to instead receive "xout". Alternatively, change it to

```
wire [2:0] x = {xin[1:0], shift_in};
```

and change all driving logic to instead drive "xin".

With this change this assignment needs to be evaluated only once. These sort of changes may also speed up your traditional event driven simulator, as it will result in fewer events per cycle.

The most complicated UNOPTFLAT path we've seen was due to low bits of a bus being generated from an always statement that consumed high bits of the same bus processed by another series of always blocks. The fix is the same; split it into two separate signals generated from each block.

Another way to resolve this warning is to add a `split_var` metacomment described above. This will cause the variable to be split internally, potentially resolving the conflict. If you run with --report-unoptflat Verilator will suggest possible candidates for `split_var`.

The UNOPTFLAT warning may also be due to clock enables, identified from the reported path going through a clock gating instance. To fix these, use the clock_enable meta comment described above.

The UNOPTFLAT warning may also occur where outputs from a block of logic are independent, but occur in the same always block. To fix this, use the isolate_assignments meta comment described above.

To assist in resolving UNOPTFLAT, the option `--report-unoptflat` can be used, which will provide suggestions for variables that can be split up, and a graph of all the nodes connected in the loop. See the *Arguments* section for more details.

Ignoring this warning will only slow simulations, it will simulate correctly.

UNOPTTHREADS

Warns that the thread scheduler was unable to partition the design to fill the requested number of threads.

One workaround is to request fewer threads with `--threads`.

Another possible workaround is to allow more MTasks in the simulation runtime, by increasing the value of --threads-max-mtasks. More MTasks will result in more communication and synchronization overhead at simulation runtime; the scheduler attempts to minimize the number of MTasks for this reason.

Ignoring this warning will only slow simulations, it will simulate correctly.

UNPACKED

Warns that unpacked structs and unions are not supported.

Ignoring this warning will make Verilator treat the structure as packed, which may make Verilator simulations differ from other simulators. This downgrading may also result what would normally be a legal unpacked struct/array inside an unpacked struct/array becoming an illegal unpacked struct/array inside a packed struct/array.

UNSIGNED

Warns that you are comparing a unsigned value in a way that implies it is signed, for example "X < 0" will always be false when X is unsigned.

Ignoring this warning will only suppress the lint check, it will simulate correctly.

UNSUPPORTED

UNSUPPORTED is an error that the construct might be legal according to IEEE but is not currently supported.

This error may be ignored with --bbox-unsup, however this will make the design simulate incorrectly; see the details under --bbox-unsup.

UNUSED

Warns that the specified signal or parameter is never used/consumed. Verilator is fairly liberal in the usage calculations; making a signal public, a signal matching --unused-regexp ("*unused*") or accessing only a single array element marks the entire signal as used.

Disabled by default as this is a code style warning; it will simulate correctly.

A recommended style for unused nets is to put at the bottom of a file code similar to the following:

```
wire _unused_ok = &{1'b0,
                    sig_not_used_a,
                    sig_not_used_yet_b,  // To be fixed
                    1'b0};
```

The reduction AND and constant zeros mean the net will always be zero, so won't use simulation runtime. The redundant leading and trailing zeros avoid syntax errors if there are no signals between them. The magic name "unused" (-unused-regexp) is recognized by Verilator and suppresses warnings; if using other lint tools, either teach it to the tool to ignore signals with "unused" in the name, or put the appropriate lint_off around the wire. Having unused signals in one place makes it easy to find what is unused, and reduces the number of lint_off pragmas, reducing bugs.

USERINFO, USERWARN, USERERROR, USERFATAL

A SystemVerilog elaboration-time assertion print was executed.

VARHIDDEN

Warns that a task, function, or begin/end block is declaring a variable by the same name as a variable in the upper level module or begin/end block (thus hiding the upper variable from being able to be used.) Rename the variable to avoid confusion when reading the code.

Disabled by default as this is a code style warning; it will simulate correctly.

WIDTH

Warns that based on width rules of Verilog, two operands have different widths. Verilator generally can intuit the common usages of widths, and you shouldn't need to disable this message like you do with most lint programs. Generally other than simple mistakes, you have two solutions:

If it's a constant 0 that's 32 bits or less, simply leave it unwidthed. Verilator considers zero to be any width needed.

Concatenate leading zeros when doing arithmetic. In the statement

```
wire [5:0] plus_one = from[5:0] + 6'd1 + carry[0];
```

The best fix, which clarifies intent and will also make all tools happy is:

```
wire [5:0] plus_one = from[5:0] + 6'd1 + {5'd0, carry[0]};
```

Ignoring this warning will only suppress the lint check, it will simulate correctly.

WIDTHCONCAT

Warns that based on width rules of Verilog, a concatenate or replication has an indeterminate width. In most cases this violates the Verilog rule that widths inside concatenates and replicates must be sized, and should be fixed in the code.

```
wire [63:0] concat = {1, 2};
```

An example where this is technically legal (though still bad form) is:

```
parameter PAR = 1;
wire [63:0] concat = {PAR, PAR};
```

The correct fix is to either size the 1 ("32'h1"), or add the width to the parameter definition ("parameter [31:0]"), or add the width to the parameter usage ("{PAR[31:0],PAR[31:0]}".

The following describes the less obvious errors:

Internal Error

This error should never occur first, though may occur if earlier warnings or error messages have corrupted the program. If there are no other warnings or errors, submit a bug report.

Unsupported: ....

This error indicates that you are using a Verilog language construct that is not yet supported in Verilator. See the Limitations chapter.

25 DEPRECATIONS

The following deprecated items are scheduled for future removal:

C++11 compiler support

Verilator currently requires C++11 or newer compilers. Verilator will require C++14 or newer compilers for both compiling Verilator and compiling Verilated models no sooner than January 2022.

--inhibit-sim

The --inhibit-sim option is planned for removal no sooner than July 2021.

Configuration File -msg

The -msg argument to lint_off has been replaced with -rule. -msg is planned for removal no sooner than January 2021.

XML locations

The XML `fl` attribute has been replaced with `loc`. `fl` is planned for removal no sooner than January 2021.

26 FAQ/FREQUENTLY ASKED QUESTIONS

Can I contribute?

Please contribute! Just submit a pull request, or raise an issue to discuss if looking for something to help on. For more information see our contributor agreement.

How widely is Verilator used?

Verilator is used by many of the largest silicon design companies, and all the way down to college projects. Verilator is one of the "big 4" simulators, meaning one of the 4 main SystemVerilog simulators available, namely the commercial products Synopsys VCS (tm), Mentor Questa/ModelSim (tm), Cadence Xcelium/Incisive/NC-Verilog/NC-Sim (tm), and the open-source Verilator. The three commercial offerings are often collectively called the "big 3" simulators.

Does Verilator run under Windows?

Yes, using Cygwin. Verilated output also compiles under Microsoft Visual C++, but this is not tested every release.

Can you provide binaries?

You can install Verilator via the system package manager (apt, yum, etc.) on many Linux distributions, including Debian, Ubuntu, SuSE, Fedora, and others. These packages are provided by the Linux distributions and generally will lag the version of the mainline Verilator repository. If no binary package is available for your distribution, how about you set one up? Please contact the authors for assistance.

How can it be faster than (name-a-big-3-closed-source-simulator)?

Generally, the implied part of the question is "... with all of the manpower they can put into developing it."

Most simulators have to be compliant with the complete IEEE 1364 (Verilog) and IEEE 1800 (SystemVerilog) standards, meaning they have to be event driven. This prevents them from being able to reorder blocks and make netlist-style optimizations, which are where most of the gains come from.

You should not be scared by non-compliance. Your synthesis tool isn't compliant with the whole standard to start with, so your simulator need not be either. Verilator is closer to the synthesis interpretation, so this is a good thing for getting working silicon.

Will Verilator output remain under my own license?

Yes, it's just like using GCC on your programs; this is why Verilator uses the "GNU *Lesser* Public License Version 3" instead of the more typical "GNU Public License". See the licenses for details, but in brief, if you change Verilator itself or the header files Verilator includes, you must make the source code available under the GNU Lesser Public License. However, Verilator output (the Verilated code) only "include"s the licensed files, and so you are NOT required to release any output from Verilator.

You also have the option of using the Perl Artistic License, which again does not require you to release your Verilog or generated code, and also allows you to modify Verilator for internal use without distributing the modified version. But please contribute back to the community!

One limit is that you cannot under either license release a commercial Verilog simulation product incorporating Verilator without making the source code available.

As is standard with Open Source, contributions back to Verilator will be placed under the Verilator copyright and LGPL/Artistic license. Small test cases will be released into the public domain so they can be used anywhere, and large tests under the LGPL/Artistic, unless requested otherwise.

Why is running Verilator (to create a model) so slow?

Verilator needs more memory than the resulting simulator will require, as Verilator internally creates all of the state of the resulting generated simulator in order to optimize it. If it takes more than a few

minutes or so (and you're not using --debug since debug mode is disk bound), see if your machine is paging; most likely you need to run it on a machine with more memory. Very large designs are known to have topped 16GB resident set size.

How do I generate waveforms (traces) in C++?

See the next question for tracing in SystemC mode.

A. Add the --trace switch to Verilator, and in your top level C code, call Verilated::traceEverOn(true). Then you may use $dumpfile and $dumpvars to enable traces, same as with any Verilog simulator. See `examples/make_tracing_c`.

B. Or, for finer-grained control, or C++ files with multiple Verilated modules you may also create the trace purely from C++. Create a VerilatedVcdC object, and in your main loop call "trace_object->dump(time)" every time step, and finally call "trace_object->close()". You also need to compile verilated_vcd_c.cpp and add it to your link, preferably by adding the dependencies in $(VK_GLOBAL_OBJS) to your Makefile's link rule. This is done for you if using the Verilator --exe flag. Note you can also call ->trace on multiple Verilated objects with the same trace file if you want all data to land in the same output file.

```
#include "verilated_vcd_c.h"
...
int main(int argc, char** argv, char** env) {
    ...
    Verilated::traceEverOn(true);
    VerilatedVcdC* tfp = new VerilatedVcdC;
    topp->trace(tfp, 99);  // Trace 99 levels of hierarchy
    tfp->open("obj_dir/t_trace_ena_cc/simx.vcd");
    ...
    while (vl_time_stamp64() < sim_time && !Verilated::gotFinish()) {
        main_time += #;
        tfp->dump(main_time);
    }
    tfp->close();
}
```

How do I generate waveforms (traces) in SystemC?

A. Add the --trace switch to Verilator, and in your top level sc_main, call Verilated::traceEverOn(true). Then you may use $dumpfile and $dumpvars to enable traces, same as with any Verilog simulator, see the non-SystemC example in `examples/make_tracing_c`. This will trace only the module containing the $dumpvar.

B. Or, you may create a trace purely from SystemC, which may trace all Verilated designs in the SystemC model. Create a VerilatedVcdSc object as you would create a normal SystemC trace file. For an example, see the call to VerilatedVcdSc in the examples/make_tracing_sc/sc_main.cpp file of the distribution, and below.

Alternatively you may use the C++ trace mechanism described in the previous question, note the timescale and timeprecision will be inherited from your SystemC settings.

You also need to compile verilated_vcd_sc.cpp and verilated_vcd_c.cpp and add them to your link, preferably by adding the dependencies in $(VK_GLOBAL_OBJS) to your Makefile's link rule. This is done for you if using the Verilator --exe flag.

Note you can also call ->trace on multiple Verilated objects with the same trace file if you want all data to land in the same output file.

When using SystemC 2.3, the SystemC library must have been built with the experimental simulation phase callback based tracing disabled. This is disabled by default when building SystemC

with its configure based build system, but when building SystemC with CMake, you must pass -DENABLE_PHASE_CALLBACKS_TRACING=OFF to disable this feature.

```
#include "verilated_vcd_sc.h"
...
int main(int argc, char** argv, char** env) {
    ...
    Verilated::traceEverOn(true);
    VerilatedVcdSc* tfp = new VerilatedVcdSc;
    topp->trace(tfp, 99);  // Trace 99 levels of hierarchy
    tfp->open("obj_dir/t_trace_ena_cc/simx.vcd");
    ...
    sc_start(1);
    ...
    tfp->close();
}
```

How do I generate FST waveforms (traces) in C++?

FST is a trace file format developed by GTKWave. Verilator provides basic FST support. To dump traces in FST format, add the --trace-fst switch to Verilator and either A. use $dumpfile/$dumpvars in Verilog as described in the VCD example above, or B. in C++ change the include described in the VCD example above:

```
#include "verilated_fst_c.h"
VerilatedFstC* tfp = new VerilatedFstC;
```

Note that currently supporting both FST and VCD in a single simulation is impossible, but such requirement should be rare. You can however ifdef around the trace format in your C++ main loop, and select VCD or FST at build time, should you require.

How do I generate FST waveforms (aka dumps or traces) in SystemC?

The FST library from GTKWave does not currently support SystemC; use VCD format instead.

How do I view waveforms (aka dumps or traces)?

Verilator creates standard VCD (Value Change Dump) and FST files. VCD files are viewable with the open source GTKWave (recommended) or Dinotrace (legacy) programs, or any of the many closed-source offerings; FST is supported only by GTKWave.

How do I reduce the size of large waveform (trace) files?

First, instead of calling VerilatedVcdC->open at the beginning of time, delay calling it until the time stamp where you want tracing to begin. Likewise you can also call VerilatedVcdC->open before the end of time (perhaps a short period after you detect a verification error).

Next, add /*verilator tracing_off*/ to any very low level modules you never want to trace (such as perhaps library cells). Finally, use the --trace-depth option to limit the depth of tracing, for example --trace-depth 1 to see only the top level signals.

Also be sure you write your trace files to a local solid-state drive, instead of to a network drive. Network drives are generally far slower.

You can also consider using FST tracing instead of VCD. FST dumps are a fraction of the size of the equivalent VCD. FST tracing can be slower than VCD tracing, but it might be the only option if the VCD file size is prohibitively large.

How do I do coverage analysis?

Verilator supports both block (line) coverage and user inserted functional coverage.

First, run verilator with the --coverage option. If you are using your own makefile, compile the model with the GCC flag -DVM_COVERAGE (if using the makefile provided by Verilator, it will do this for you).

At the end of your test, call VerilatedCov::write passing the name of the coverage data file (typically "logs/coverage.dat").

Run each of your tests in different directories. Each test will create a logs/coverage.dat file.

After running all of your tests, execute the verilator_coverage tool. The verilator_coverage tool reads the logs/coverage.dat file(s), and creates an annotated source code listing showing code coverage details.

For an example, after running 'make test' in the Verilator distribution, see the examples/make_tracing_c/logs directory. Grep for lines starting with '%' to see what lines Verilator believes need more coverage.

Info files can be written by verilator_coverage for import to `lcov`. This enables use of `genhtml` for HTML reports and importing reports to sites such as https://codecov.io.

Where is the translate_off command? (How do I ignore a construct?)

Translate on/off pragmas are generally a bad idea, as it's easy to have mismatched pairs, and you can't see what another tool sees by just preprocessing the code. Instead, use the preprocessor; Verilator defines the "VERILATOR" define for you, so just wrap the code in an ifndef region:

```
‘ifndef VERILATOR
   Something_Verilator_Dislikes;
‘endif
```

Most synthesis tools similarly define SYNTHESIS for you.

Why do I get "unexpected `do`" or "unexpected `bit`" errors?

The words `do`, `bit`, `ref`, `return`, and others are reserved keywords in SystemVerilog. Older Verilog code might use these as identifiers. You should change your code to not use them to ensure it works with newer tools. Alternatively, surround them by the Verilog 2005/SystemVerilog begin_keywords pragma to indicate Verilog 2001 code.

```
‘begin_keywords "1364-2001"
   integer bit; initial bit = 1;
‘end_keywords
```

If you want the whole design to be parsed as Verilog 2001, please see the `--default-language` option.

How do I prevent my assertions from firing during reset?

Call Verilated::assertOn(false) before you first call the model, then turn it back on after reset. It defaults to true. When false, all assertions controlled by --assert are disabled.

Why do I get "undefined reference to ‘sc_time_stamp()’"?

In 4.200 and later, using the timeInc function is recommended instead. See the "CONNECTING TO C++" examples. Some linkers (MSVC++) still require sc_time_stamp() to be defined, either define this with ("double sc_time_stamp() { return 0; }") or compile the Verilated code with -DVL_TIME_CONTEXT.

Prior to Verilator 4.200, the sc_time_stamp function needs to be defined in C++ (non SystemC) to return the current simulation time.

Why do I get "undefined reference to 'VL_RAND_RESET_I' or 'Verilated::...'"?

You need to link your compiled Verilated code against the verilated.cpp file found in the include directory of the Verilator kit. This is one target in the $(VK_GLOBAL_OBJS) make variable, which should be part of your Makefile's link rule. If you use --exe, this is done for you.

Is the PLI supported?

Only somewhat. More specifically, the common PLI-ish calls $display, $finish, $stop, $time, $write are converted to C++ equivalents. You can also use the "import DPI" SystemVerilog feature to call C code (see the chapter above). There is also limited VPI access to public signals.

If you want something more complex, since Verilator emits standard C++ code, you can simply write your own C++ routines that can access and modify signal values without needing any PLI interface code, and call it with $c("{any_c++_statement}").

How do I make a Verilog module that contain a C++ object?

You need to add the object to the structure that Verilator creates, then use $c to call a method inside your object. The test_regress/t/t_extend_class files show an example of how to do this.

How do I get faster build times?

When running make pass the make variable VM_PARALLEL_BUILDS=1 so that builds occur in parallel. Note this is now set by default if an output file was large enough to be split due to the --output-split option.

Verilator emits any infrequently executed "cold" routines into separate __Slow.cpp files. This can accelerate compilation as optimization can be disabled on these routines. See the OPT_FAST and OPT_SLOW make variables and the BENCHMARKING & OPTIMIZATION section of the manual.

Use a recent compiler. Newer compilers tend to be faster.

Compile in parallel on many machines and use caching; see the web for the ccache, distcc and icecream packages. ccache will skip GCC runs between identical source builds, even across different users. If ccache was installed when Verilator was built it is used, or see OBJCACHE environment variable to override this. Also see the --output-split option.

To reduce the compile time of classes that use a Verilated module (e.g. a top CPP file) you may wish to add /*verilator no_inline_module*/ to your top level module. This will decrease the amount of code in the model's Verilated class, improving compile times of any instantiating top level C++ code, at a relatively small cost of execution performance.

Why do so many files need to recompile when I add a signal?

Adding a new signal requires the symbol table to be recompiled. Verilator uses one large symbol table, as that results in 2-3 less assembly instructions for each signal access. This makes the execution time 10-15% faster, but can result in more compilations when something changes.

How do I access Verilog functions/tasks in C?

Use the SystemVerilog Direct Programming Interface. You write a Verilog function or task with input/outputs that match what you want to call in with C. Then mark that function as a DPI export function. See the DPI chapter in the IEEE Standard.

How do I access C++ functions/tasks in Verilog?

Use the SystemVerilog Direct Programming Interface. You write a Verilog function or task with input/outputs that match what you want to call in with C. Then mark that function as a DPI import function. See the DPI chapter in the IEEE Standard.

How do I access signals in C?

The best thing to do is to make a SystemVerilog "export DPI" task or function that accesses that signal, as described in the DPI chapter in the manual and DPI tutorials on the web. This will allow Verilator to better optimize the model and should be portable across simulators.

If you really want raw access to the signals, declare the signals you will be accessing with a /*verilator public*/ comment before the closing semicolon. Then scope into the C++ class to read the value of the signal, as you would any other member variable.

Signals are the smallest of 8-bit unsigned chars (equivalent to uint8_t), 16-bit unsigned shorts (uint16_t), 32-bit unsigned longs (uint32_t), or 64-bit unsigned long longs (uint64_t) that fits the width of the signal. Generally, you can use just uint32_t's for 1 to 32 bits, or vluint64_t for 1 to 64 bits, and the compiler will properly up-convert smaller entities. Note even signed ports are declared as unsigned; you must sign extend yourself to the appropriate signal width.

Signals wider than 64 bits are stored as an array of 32-bit uint32_t's. Thus to read bits 31:0, access signal[0], and for bits 63:32, access signal[1]. Unused bits (for example bit numbers 65-96 of a 65-bit vector) will always be zero. If you change the value you must make sure to pack zeros in the unused bits or core-dumps may result, because Verilator strips array bound checks where it believes them to be unnecessary to improve performance.

In the SYSTEMC example above, if you had in our.v:

```
input clk /*verilator public*/;
// Note the placement of the semicolon above
```

From the sc_main.cpp file, you'd then:

```
#include "Vour.h"
#include "Vour_our.h"
cout << "clock is " << top->our->clk << endl;
```

In this example, clk is a bool you can read or set as any other variable. The value of normal signals may be set, though clocks shouldn't be changed by your code or you'll get strange results.

Should a module be in Verilog or SystemC?

Sometimes there is a block that just interconnects instances, and have a choice as to if you write it in Verilog or SystemC. Everything else being equal, best performance is when Verilator sees all of the design. So, look at the hierarchy of your design, labeling instances as to if they are SystemC or Verilog. Then:

A module with only SystemC instances below must be SystemC.

A module with a mix of Verilog and SystemC instances below must be SystemC. (As Verilator cannot connect to lower-level SystemC instances.)

A module with only Verilog instances below can be either, but for best performance should be Verilog. (The exception is if you have a design that is instantiated many times; in this case Verilating one of the lower modules and instantiating that Verilated instances multiple times into a SystemC module *may* be faster.)

27 BUGS

First, check the LANGUAGE LIMITATIONS section.

Next, try the --debug switch. This will enable additional internal assertions, and may help identify the problem.

Finally, reduce your code to the smallest possible routine that exhibits the bug. Even better, create a test in the test_regress/t directory, as follows:

```
cd test_regress
cp -p t/t_EXAMPLE.pl t/t_BUG.pl
cp -p t/t_EXAMPLE.v t/t_BUG.v
```

There are many hints on how to write a good test in the driver.pl documentation which can be seen by running:

```
cd $VERILATOR_ROOT  # Need the original distribution kit
test_regress/driver.pl --help
```

Edit t/t_BUG.pl to suit your example; you can do anything you want in the Verilog code there; just make sure it retains the single clk input and no outputs. Now, the following should fail:

```
cd $VERILATOR_ROOT  # Need the original distribution kit
cd test_regress
t/t_BUG.pl  # Run on Verilator
t/t_BUG.pl --debug # Run on Verilator, passing --debug to Verilator
t/t_BUG.pl --vcs  # Run on VCS simulator
t/t_BUG.pl --nc|--iv|--ghdl  # Likewise on other simulators
```

The test driver accepts a number of options, many of which mirror the main Verilator option. For example the previous test could have been run with debugging enabled. The full set of test options can be seen by running driver.pl --help as shown above.

Finally, report the bug using the bug tracker at https://verilator.org/issues. The bug will become publicly visible; if this is unacceptable, mail the bug report to wsnyder@wsnyder.org.

28 HISTORY

Verilator was conceived in 1994 by Paul Wasson at the Core Logic Group at Digital Equipment Corporation. The Verilog code that was converted to C was then merged with a C based CPU model of the Alpha processor and simulated in a C based environment called CCLI.

In 1995 Verilator started being used also for Multimedia and Network Processor development inside Digital. Duane Galbi took over active development of Verilator, and added several performance enhancements. CCLI was still being used as the shell.

In 1998, through the efforts of existing DECies, mainly Duane Galbi, Digital graciously agreed to release the source code. (Subject to the code not being resold, which is compatible with the GNU Public License.)

In 2001, Wilson Snyder took the kit, and added a SystemC mode, and called it Verilator2. This was the first packaged public release.

In 2002, Wilson Snyder created Verilator 3.000 by rewriting Verilator from scratch in C++. This added many optimizations, yielding about a 2-5x performance gain.

In 2009, major SystemVerilog and DPI language support was added.

In 2018, Verilator 4.000 was released with multithreaded support.

Currently, various language features and performance enhancements are added as the need arises. Verilator is now about 3x faster than in 2002, and is faster than most (if not every) other simulator.

29 AUTHORS

When possible, please instead report bugs to https://verilator.org/issues.

Wilson Snyder <wsnyder@wsnyder.org>

Major concepts by Paul Wasson, Duane Galbi, John Coiner and Jie Xu.

30 CONTRIBUTORS

Many people have provided ideas and other assistance with Verilator.

Verilator is receiving major development support from the CHIPS Alliance.

Previous major corporate sponsors of Verilator, by providing significant contributions of time or funds included include Atmel Corporation, Cavium Inc., Compaq Corporation, Digital Equipment Corporation, Embecosm Ltd., Hicamp Systems, Intel Corporation, Mindspeed Technologies Inc., MicroTune Inc., picoChip Designs Ltd., Sun Microsystems Inc., Nauticus Networks Inc., and SiCortex Inc.

The people who have contributed major functionality are Byron Bradley, Jeremy Bennett, Lane Brooks, John Coiner, Duane Galbi, Geza Lore, Todd Strader, Stefan Wallentowitz, Paul Wasson, Jie Xu, and Wilson Snyder. Major testers included Jeff Dutton, Jonathon Donaldson, Ralf Karge, David Hewson, Iztok Jeras, Wim Michiels, Alex Solomatnikov, Sebastien Van Cauwenberghe, Gene Weber, and Clifford Wolf.

Some of the people who have provided ideas, and feedback for Verilator include: David Addison, Tariq B. Ahmad, Nikana Anastasiadis, Hans Van Antwerpen, Vasu Arasanipalai, Jens Arm, Sharad Bagri, Matthew Ballance, Andrew Bardsley, Matthew Barr, Geoff Barrett, Julius Baxter, Jeremy Bennett, Michael Berman, Victor Besyakov, Moinak Bhattacharyya, David Binderman, Piotr Binkowski, Johan Bjork, David Black, Tymoteusz Blazejczyk, Daniel Bone, Gregg Bouchard, Christopher Boumenot, Nick Bowler, Byron Bradley, Bryan Brady, Maarten De Braekeleer, Charlie Brej, J Briquet, Lane Brooks, John Brownlee, Jeff Bush, Lawrence Butcher, Tony Bybell, Ted Campbell, Chris Candler, Lauren Carlson, Donal Casey, Sebastien Van Cauwenberghe, Alex Chadwick, Terry Chen, Yi-Chung Chen, Enzo Chi, Robert A. Clark, Allan Cochrane, John Coiner, Gianfranco Costamagna, Sean Cross, George Cuan, Joe DErrico, Lukasz Dalek, Laurens van Dam, Gunter Dannoritzer, Ashutosh Das, Bernard Deadman, John Demme, Mike Denio, John Deroo, Philip Derrick, John Dickol, Ruben Diez, Danny Ding, Jacko Dirks, Ivan Djordjevic, Jonathon Donaldson, Leendert van Doorn, Sebastian Dressler, Alex Duller, Jeff Dutton, Tomas Dzetkulic, Usuario Eda, Charles Eddleston, Chandan Egbert, Joe Eiler, Ahmed El-Mahmoudy, Trevor Elbourne, Mats Engstrom, Robert Farrell, Eugen Fekete, Fabrizio Ferrandi, Udi Finkelstein, Brian Flachs, Andrea Foletto, Bob Fredieu, Duane Galbi, Benjamin Gartner, Christian Gelinek, Peter Gerst, Glen Gibb, Michael Gielda, Shankar Giri, Dan Gisselquist, Petr Gladkikh, Sam Gladstone, Amir Gonnen, Chitlesh Goorah, Kai Gossner, Sergi Granell, Al Grant, Alexander Grobman, Xuan Guo, Driss Hafdi, Neil Hamilton, James Hanlon, Oyvind Harboe, Jannis Harder, Junji Hashimoto, Thomas Hawkins, Mitch Hayenga, Robert Henry, Stephen Henry, David Hewson, Jamey Hicks, Joel Holdsworth, Andrew Holme, Hiroki Honda, Alex Hornung, David Horton, Peter Horvath, Jae Hossell, Alan Hunter, James Hutchinson, Jamie Iles, Ben Jackson, Shareef Jalloq, Krzysztof

Jankowski, HyungKi Jeong, Iztok Jeras, James Johnson, Christophe Joly, Franck Jullien, James Jung, Mike Kagen, Arthur Kahlich, Kaalia Kahn, Guy-Armand Kamendje, Vasu Kandadi, Kanad Kanhere, Patricio Kaplan, Pieter Kapsenberg, Ralf Karge, Dan Katz, Sol Katzman, Ian Kennedy, Jonathan Kimmitt, Olof Kindgren, Kevin Kiningham, Dan Kirkham, Sobhan Klnv, Gernot Koch, Soon Koh, Nathan Kohagen, Steve Kolecki, Brett Koonce, Will Korteland, Wojciech Koszek, Varun Koyyalagunta, David Kravitz, Roland Kruse, Sergey Kvachonok, Charles Eric LaForest, Ed Lander, Steve Lang, Stephane Laurent, Walter Lavino, Christian Leber, Larry Lee, Igor Lesik, John Li, Eivind Liland, Yu Sheng Lin, Charlie Lind, Andrew Ling, Jiuyang Liu, Paul Liu, Derek Lockhart, Jake Longo, Geza Lore, Arthur Low, Stefan Ludwig, Dan Lussier, Fred Ma, Duraid Madina, Affe Mao, Julien Margetts, Mark Marshall, Alfonso Martinez, Yves Mathieu, Patrick Maupin, Jason McMullan, Elliot Mednick, Wim Michiels, Miodrag Milanovic, Wai Sum Mong, Peter Monsson, Sean Moore, Dennis Muhlestein, John Murphy, Matt Myers, Nathan Myers, Richard Myers, Dimitris Nalbantis, Peter Nelson, Bob Newgard, Cong Van Nguyen, Paul Nitza, Yossi Nivin, Pete Nixon, Lisa Noack, Mark Nodine, Kuba Ober, Andreas Olofsson, Aleksander Osman, James Pallister, Vassilis Papaefstathiou, Brad Parker, Dan Petrisko, Maciej Piechotka, David Pierce, Dominic Plunkett, David Poole, Mike Popoloski, Roman Popov, Rich Porter, Niranjan Prabhu, Usha Priyadharshini, Mark Jackson Pulver, Prateek Puri, Marshal Qiao, Danilo Ramos, Chris Randall, Anton Rapp, Josh Redford, Odd Magne Reitan, Frederic Requin, Frederick Requin, Dustin Richmond, Alberto Del Rio, Eric Rippey, Oleg Rodionov, Ludwig Rogiers, Paul Rolfe, Arjen Roodselaar, Tobias Rosenkranz, Huang Rui, Jan Egil Ruud, Denis Rystsov, John Sanguinetti, Galen Seitz, Salman Sheikh, Hao Shi, Mike Shinkarovsky, Rafael Shirakawa, Jeffrey Short, Anderson Ignacio Da Silva, Rodney Sinclair, Steven Slatter, Brian Small, Garrett Smith, Tim Snyder, Maciej Sobkowski, Stan Sokorac, Alex Solomatnikov, Wei Song, Art Stamness, David Stanford, John Stevenson, Pete Stevenson, Patrick Stewart, Rob Stoddard, Todd Strader, John Stroebel, Sven Stucki, Howard Su, Emerson Suguimoto, Gene Sullivan, Qingyao Sun, Renga Sundararajan, Rupert Swarbrick, Yutetsu Takatsukasa, Peter Tengstrand, Wesley Terpstra, Rui Terra, Stefan Thiede, Gary Thomas, Ian Thompson, Kevin Thompson, Mike Thyer, Hans Tichelaar, Viktor Tomov, Steve Tong, Michael Tresidder, Neil Turton, Srini Vemuri, Yuri Victorovich, Bogdan Vukobratovic, Holger Waechtler, Philipp Wagner, Stefan Wallentowitz, Shawn Wang, Paul Wasson, Greg Waters, Thomas Watts, Eugene Weber, David Welch, Thomas J Whatson, Marco Widmer, Leon Wildman, Daniel Wilkerson, Gerald Williams, Trevor Williams, Jan Van Winkel, Jeff Winston, Joshua Wise, Clifford Wolf, Tobias Wolfel, Johan Wouters, Junyi Xi, Ding Xiaoliang, Jie Xu, Mandy Xu, Takatsukasa Y, Luke Yang, and Amir Yazdanbakhsh.

Thanks to them, and all those we've missed including above, or wished to remain anonymous.

31 DISTRIBUTION

The latest version is available from https://verilator.org.

32 SEE ALSO

verilator_coverage, verilator_gantt, verilator_profcfunc, *make*,

verilator --help which is the source for this document,

and docs/internals.rst in the distribution.