

Verilator-3.670

Wilson Snyder  
<http://www.veripool.org>

2008-07-23

## Contents

<b>1</b>	<b>NAME</b>	<b>2</b>
<b>2</b>	<b>SYNOPSIS</b>	<b>2</b>
<b>3</b>	<b>DESCRIPTION</b>	<b>2</b>
<b>4</b>	<b>ARGUMENT SUMMARY</b>	<b>2</b>
<b>5</b>	<b>ARGUMENTS</b>	<b>4</b>
<b>6</b>	<b>VERILOG ARGUMENTS</b>	<b>9</b>
<b>7</b>	<b>EXAMPLE C++ EXECUTION</b>	<b>10</b>
<b>8</b>	<b>EXAMPLE SYSTEMC EXECUTION</b>	<b>11</b>
<b>9</b>	<b>BENCHMARKING &amp; OPTIMIZATION</b>	<b>13</b>
<b>10</b>	<b>FILES</b>	<b>14</b>
<b>11</b>	<b>ENVIRONMENT</b>	<b>15</b>
<b>12</b>	<b>CONNECTING TO C++</b>	<b>16</b>
<b>13</b>	<b>CONNECTING TO SYSTEMC</b>	<b>17</b>
<b>14</b>	<b>CROSS COMPILATION</b>	<b>17</b>
<b>15</b>	<b>VERILOG 2001 (IEEE 1364-2001) SUPPORT</b>	<b>18</b>
<b>16</b>	<b>VERILOG 2005 (IEEE 1364-2005) SUPPORT</b>	<b>18</b>
<b>17</b>	<b>SYSTEMVERILOG (IEEE 1800-2005) SUPPORT</b>	<b>18</b>

<b>18 SUGAR/PSL SUPPORT</b>	<b>19</b>
<b>19 SYNTHESIS DIRECTIVE ASSERTION SUPPORT</b>	<b>19</b>
<b>20 LANGUAGE EXTENSIONS</b>	<b>20</b>
<b>21 LANGUAGE LIMITATIONS</b>	<b>24</b>
<b>22 LANGUAGE KEYWORD LIMITATIONS</b>	<b>28</b>
<b>23 ERRORS AND WARNINGS</b>	<b>29</b>
<b>24 FAQ/FREQUENTLY ASKED QUESTIONS</b>	<b>36</b>
<b>25 BUGS</b>	<b>40</b>
<b>26 HISTORY</b>	<b>41</b>
<b>27 CONTRIBUTORS</b>	<b>41</b>
<b>28 DISTRIBUTION</b>	<b>42</b>
<b>29 AUTHORS</b>	<b>42</b>
<b>30 SEE ALSO</b>	<b>42</b>

## 1 NAME

Verilator - Convert Verilog code to C++/SystemC

## 2 SYNOPSIS

```

verilator --help
verilator --version
verilator --cc [options] [top_level.v] [opt_c_files.cpp/c/cc]
verilator --sc [options] [top_level.v] [opt_c_files.cpp/c/cc]
verilator --sp [options] [top_level.v] [opt_c_files.cpp/c/cc]
verilator --lint-only ...

```

## 3 DESCRIPTION

Verilator converts synthesizable (not behavioral) Verilog code, plus some Synthesis, SystemVerilog and Sugar/PSL assertions, into C++, SystemC or SystemPerl code. It is not a complete simulator, just a compiler.

Verilator is invoked with parameters similar to GCC, Cadence Verilog-XL/NC-Verilog, or Synopsys's VCS. It reads the specified Verilog code, lints it, and optionally adds coverage and waveform tracing code. For C++ and SystemC formats, it outputs .cpp and .h files. For SystemPerl format, it outputs .sp files for the SystemPerl preprocessor, which greatly simplifies writing SystemC code and is available at <http://www.veripool.org>.

The files created by Verilator are then compiled with C++. The user writes a little C++ wrapper file, which instantiates the top level module, and passes this filename on the command line. These C files are compiled in C++, and linked with the Verilated files.

The resulting executable will perform the actual simulation.

## 4 ARGUMENT SUMMARY

This is a short summary of the arguments to Verilator. See the detailed descriptions in the next sections for more information.

<code>{file.v}</code>	Verilog top level filenames
<code>{file.c/cc/cpp}</code>	Optional C++ files to link in
<code>--assert</code>	Enable all assertions

--autoflush	Flush streams after all \$displays
--bin <filename>	Override Verilator binary
--cc	Create C++ output
--compiler <compiler-name>	Tune for specified C++ compiler
--coverage	Enable all coverage
--coverage-line	Enable line coverage
--coverage-user	Enable PSL/SVL user coverage
-D<var>[=<value>]	Set preprocessor define
--debug	Enable debugging
--debug-check	Enable debugging assertions
--dump-tree	Enable dumping .tree files
-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
--help	Display this help.
-I<dir>	Directory to search for includes
--inhibit-sim	Create function to turn off sim
--inline-mult <value>	Tune module inlining
--language <lang>	Language standard to parse
--lint-only	Lint, but do not make output
--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-skip-identical	Disable skipping identical output
-O0	Disable optimizations
-O3	High performance optimizations
-O<optimization-letter>	Selectable optimizations
--output-split <bytes>	Split .cpp files into pieces
--output-split-cfuncs <statements>	Split .ccp functions
--pins64	Use uint64_t's for 33-64 bit sigs
--prefix <topname>	Name of top level class
--profile-cfuncs	Name functions for profiling
--private	Debugging; see docs
--psl	Enable PSL parsing
--public	Debugging; see docs
--sc	Create SystemC output
--sp	Create SystemPerl output
--stats	Create statistics file
--top-module <topname>	Name of top level input module
--trace	Enable waveform creation
--trace-depth <levels>	Depth of tracing
-U<var>	Undefine preprocessor define
--underline-zero	Zero signals with leading _'s
-v <filename>	Verilog library
-Werror-<message>	Convert warning to error
-Wfuture-<message>	Disable unknown message warnings
-Wno-<message>	Disable warning
-x-assign <mode>	Initially assign Xs to this value

<code>-y &lt;dir&gt;</code>	Directory to search for modules
<code>+define+&lt;var&gt;+&lt;value&gt;</code>	Set preprocessor define
<code>+incdir+&lt;dir&gt;</code>	Directory to search for includes
<code>+libext+&lt;ext&gt;+[ext]...</code>	Extensions for finding modules

## 5 ARGUMENTS

### `{file.v}`

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

### `{file.c/cc/cpp}`

Specifies optional C++ files to be linked in with the Verilog code. If any C++ files are specified in this way, Verilator will include a make rule that generates a *module* executable. Without any C++ files, Verilator will stop at the *module \_\_ALL.a* library, and presume you'll continue linking with make rules you write yourself.

### `-assert`

Enable all assertions, includes enabling the `-psl` flag. (If `psl` is not desired, but other assertions are, use `-assert -nopsl`.)

See also `-x-assign`; setting "`-x-assign unique`" may be desirable.

### `-autoflush`

After every `$display` or `$fdisplay`, flush the output stream. This insures that messages will appear immediately but may reduce performance. Defaults off, which will buffer output as provided by the normal C `stdio` calls.

### `-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.

### `-cc`

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

### `-compiler compiler-name`

Enables tunings and work-arounds for the specified C++ compiler.

#### **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 hardcoded 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.

**-coverage**

Enables all forms of coverage, alias for `-coverage-line`, `-coverage-user`

**-coverage-line**

Specifies basic block line coverage analysis code should be inserted.

Coverage analysis adds statements at each code flow change point, which are the branches of IF and CASE statements, a super-set of normal Verilog Line Coverage. 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.pl`.

After running multiple tests, the `vcovage` utility (from the SystemPerl package) is executed. `Vcovage` reads the `logs/coverage.pl` file(s), and creates an annotated source code listing showing code coverage details.

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.

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.

For an example, after running `'make test'` in the Verilator distribution, see the `test_sp/logs/coverage_source` directory. Grep for lines starting with `'%'` to see what lines Verilator believes need more coverage.

**-coverage-user**

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

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

```
// psl default clock = posedge clk;
// psl cover {cyc==9} report "DefaultClock,expect=1";
```

**-debug**

Select the debug built image of Verilator (if available), and enable more internal assertions, debugging messages, and intermediate form dump files.

**-debug-check**

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

**-dump-tree**

Rarely needed. Enable writing `.tree` debug files. This is enabled with `-debug`, so `"-debug -no-dump-tree"` may be useful if the dump files are large and not desired.

**-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 or warnings are encountered, exit. Defaults to 50.

**-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.

**-help**

Displays this message and program version and exits.

**-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  $\leq 1$  will inline everything, will lead to longer compile times, but potentially faster runtimes. This setting is ignored for very small modules; they will always be inlined, if allowed.

**-language *value***

Select the language to be used when first processing each Verilog file. The language value must be "1364-1995", "1364-2001", "1364-2001", "1364-2005", or "1800-2005". This should only be used for legacy code, as the preferable option is to edit the code to repair new keywords, or add appropriate 'begin\_ keywords.

**-lint-only**

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

**-MMD**

Enable creation of .d dependency files, used for make dependency detection, similar to gcc -MMD option. On by default, use -no-MMD to disable.

**-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.



**-mod-prefix *topname***

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

**-no-skip-identical**

Rarely needed. Disables skipping execution of Verilator if all source files are identical, and all output files exist with newer dates.

**-O0**

Disables optimization of the model.

**-O3**

Enables slow optimizations. This may reduce simulation runtimes 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.

**-output-split *bytes***

Enables splitting the output .cpp/.sp files into multiple outputs. When a C++ file exceeds the specified number of operations, a new file will be created. In addition, any slow routines will be placed into \_\_Slow files. This accelerates compilation by as optimization can be disabled on the slow routines, and the remaining files can be compiled on parallel machines. Using -output-split should have only a trivial impact on performance. With GCC 3.3 on a 2GHz Opteron, -output-split 20000 will result in splitting into approximately one-minute-compile chunks.

**-output-split-cfuncs *statements***

Enables splitting functions in the output .cpp/.sp 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 GCC to compile faster, at a small loss in performance that increases with smaller statement values.

**-pins64**

Specifies SystemC outputs of 33-64 bits wide should use uint64\_t instead of the backward-compatible default of sc\_bv's.

**-prefix *topname***

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

**-profile-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.

**-private**

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

**-psl**

Enable PSL parsing. Without this switch, PSL meta-comments are ignored. See the -assert flag to enable all assertions, and -coverage-user to enable functional coverage.

**-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*/`.

**-sc**

Specifies SystemC output mode; see also -cc and -sp.

**-sp**

Specifies SystemPerl output mode; see also -cc and -sc.

**-stats**

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

**-top-module *topname***

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

**-trace**

Adds waveform tracing code to the model. Verilator will generate additional `{prefix}__Trace*.cpp` files that will need to be compiled. In addition `Sp.cpp` (for SystemC traces) or `SpTraceVcdC.c` (for C++ only) from the SystemPerl kit's src directory must be compiled and linked in. If using the Verilator generated Makefiles, these will be added as source targets for you. If you're 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 waveforms are not turned on during model execution.

**-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 runtime and trace file size.

**-underline-zero**

Rarely needed. Signals starting with a underline should be initialized to zero, as was done in Verilator 2. Default is for all signals including those with underlines being randomized. This option may be depreciated in future versions.

**-Werror-message**

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

**-Wfuture-message**

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 message.

**-Wno-lint**

Disable all lint related warning messages. This is equivalent to "-Wno-CASEINCOMPLETE -Wno-CASEOVERLAP -Wno-CASEX -Wno-CASEZWITHX -Wno-CMPCONST -Wno-IMPLICIT -Wno-UNDRIVEN -Wno-UNSIGNED -Wno-UNUSED -Wno-VARHIDDEN -Wno-WIDTH".

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.

**-x-assign 0****-x-assign 1****-x-assign fast (default)****-x-assign unique**

Controls the two-state value that is replaced when an assignment to X is encountered. -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. -x-assign=unique will call a function to determine the value, this allows randomization of all Xs to find reset bugs and is the slowest, but safest for finding reset bugs in code.

## 6 VERILOG ARGUMENTS

The following arguments are compatible with GCC, VCS and most Verilog programs.

**+define+*var*+*value***

**-D*var*=*value***

Defines the given preprocessor symbol.

**-f *file***

Read the specified file, and act as if all text inside it was specified as command line parameters.

**+incdir+*dir***

**-I*dir***

**-y *dir***

Add the directory to the list of directories that should be searched for include directories or libraries.

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

Specify the extensions that should be used for finding modules. If for example module *x* is referenced, look in *x.ext*.

**-U*var***

Undefines the given preprocessor symbol.

**-v *filename***

Read the filename as a Verilog library. Any modules in the file may be used to resolve cell instantiations in the top level module, else ignored.

## 7 EXAMPLE C++ EXECUTION

We'll compile this example into C++.

```
mkdir test_our
cd test_our
```

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

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

```

        while (!Verilated::gotFinish()) { top->eval(); }
        exit(0);
    }
EOF

```

Now we run Verilator on our little example.

```

export VERILATOR_ROOT=/path/to/where/verilator/was/installed
$VERILATOR_ROOT/bin/verilator --cc our.v --exe 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
```

We then can compile it

```

cd obj_dir
make -f Vour.mk Vour

```

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

And now we run it

```

cd ..
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. See the `test_c` directory in the distribution for an example.

## 8 EXAMPLE SYSTEMC EXECUTION

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

```

mkdir test_our_sc
cd test_our_sc

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

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

```

Now we run Verilator on our little example.

```

export VERILATOR_ROOT=/path/to/where/verilator/was/installed
$VERILATOR_ROOT/bin/verilator --sp our.v

```

Then we convert the SystemPerl output to SystemC.

```

cd obj_dir
export SYSTEMPERL=/path/to/where/systemperl/kit/came/from
$SYSTEMPERL/sp_preproc --preproc *.sp

```

(You can also skip the above `sp_preproc` by getting pure SystemC from Verilator by replacing the `verilator -sp` flag in the previous step with `-sc`.)

We then can compile it

```

make -f Vour.mk Vour__ALL.a
make -f Vour.mk ../sc_main.o
make -f Vour.mk verilated.o

```

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

```
export SYSTEMC=/path/to/where/systemc/was/built/or/installed
g++ -L$SYSTEMC/lib-linux ../sc_main.o Vour__ALL*.o 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 `test_sp` directory in the distribution for an example.

## 9 BENCHMARKING & OPTIMIZATION

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

Minor Verilog code changes can also give big wins. You should not have any UNOPT-FLAT 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.

By default, the `lib/verilated.mk` file has optimization turned off. This is for the benefit of new users, as it improves compile times at the cost of runtimes. To add optimization as the default, set one of three variables, `OPT`, `OPT_FAST`, or `OPT_SLOW` in `lib/verilated.mk`. Or, just for one run, pass them on the command line to make:

```
make OPT_FAST="-O2" -f Vour.mk Vour__ALL.a
```

`OPT_FAST` specifies optimizations for those programs that are part of the fast path, mostly code that is executed every cycle. `OPT_SLOW` specifies optimizations for

slow-path files (plus tracing), which execute only rarely, yet take a long time to compile with optimization on. OPT specifies overall optimization and affects all compiles, including those OPT\_FAST and OPT\_SLOW affect. For best results, use OPT="-O2", and link with "-static". Nearly the same results can be had with much better compile times with OPT\_FAST="-O1 -fstrict-aliasing".

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

For best results, use GCC 3.3 or newer. GCC 3.2 and earlier have optimization bugs around pointer aliasing detection, which can result in 2x performance losses.

If you will be running many simulations on a single compile, investigate feedback driven compilation. With GCC, using -fprofile-arcs, then -fbranch-probabilities will yield another 15% or so.

You may uncover further tuning possibilities by profiling the Verilog code. Use Verilator's -profile-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\_proffunc 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. I like to keep tabs on how Verilator compares, and may be able to suggest additional improvements.

## 10 FILES

All output files are placed in the output directory name specified with the -Mdir option, or "obj\_dir" if not specified.

Verilator creates the following files in the output directory:

```
{prefix}.mk           // Make include file for compiling
{prefix}_classes.mk   // Make include file with class names
```

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

```
{prefix}.cpp          // Top level C++ file
{prefix}.h            // Top level header
{prefix}{each_verilog_module}.cpp // Lower level internal C++ files
{prefix}{each_verilog_module}.h   // Lower level internal header files
```

For -sp mode, instead of .cpp and .h it creates:



```
{prefix}.sp                // Top level SystemC file
{prefix}{each_verilog_module}.sp  // Lower level internal SC files
```

In certain optimization modes, it also creates:

```
{prefix}__Inlines.h        // Inline support functions
{prefix}__Slow.cpp         // Constructors and infrequent routines
{prefix}__Syms.cpp         // Global symbol table C++
{prefix}__Syms.h           // Global symbol table header
{prefix}__Trace.cpp        // Wave file generation code (--trace)
{prefix}__stats.txt        // Statistics (--stats)
```

It also creates internal files that can be mostly ignored:

```
{each_verilog_module}.vpp    // Post-processed verilog (--debug)
{prefix}.flags_vbin          // Verilator dependencies
{prefix}.flags_vpp           // Pre-processor dependencies
{prefix}{misc}.d             // Make dependencies (-MMD)
{prefix}{misc}.dot           // Debugging graph files (--debug)
{prefix}{misc}.tree          // Debugging files (--debug)
```

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

```
{prefix}                    // Final executable (w/--exe argument)
{prefix}__ALL.a              // Library of all Verilated objects
{prefix}{misc}.o             // Intermediate objects
```

## 11 ENVIRONMENT

### SYSTEMC

Required for SystemC output mode. If set, specifies the directory containing the SystemC distribution. This is used to find the SystemC include files. If not specified, it will come from a default optionally specified at configure time (before Verilator was compiled).

### SYSTEMC\_ARCH

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.

**SYSTEMPERL**

Specifies the directory containing the Verilog-Perl distribution kit. This is used to find the Verilog-Perl library and include files. If not specified, it will come from a default optionally specified at configure time (before Verilator was compiled).

**VCS\_HOME**

If set, specifies the directory containing the Synopsys VCS distribution. When set, a 'make test' in the Verilator distribution will also run VCS baseline regression tests.

**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\_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).

## 12 CONNECTING TO C++

Verilator creates a .h and .cpp file for the top level module and all modules under it. See the test\_c directory in the kit for an example.

After the modules are completed, there will be a *module.mk* file that may be used with Make to produce a *module\_\_ALL.a* file with all required objects in it. This is then linked with the user's top level to create the simulation executable.

The user must write the top level of the simulation. Here's a simple example:

```
#include <verilated.h>           // Defines common routines
#include "Vtop.h"                 // From Verilating "top.v"

Vtop *top;                       // Instantiation of module

unsigned int main_time = 0;       // Current simulation time

double sc_time_stamp () {        // Called by $time in Verilog
    return main_time;
}

int main() {
    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)
}

```

Note signals are read and written as member variables of the lower module. You call the `eval()` method to evaluate the model. When the simulation is complete call the `final()` method to wrap up any SystemVerilog final blocks, and complete any assertions.

## 13 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`, unless they are marked with `'systemc_clock`, in which case they become `sc_clock`'s (for SystemC 1.2, not needed in SystemC 2.0). Pins 2-32 bits wide become `uint32_t`'s. Pins 33-64 bits wide become `sc_bv`'s or `uint64_t`'s depending on the `-pins64` switch. Wider pins become `sc_bv`'s.

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.

## 14 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.

## 15 VERILOG 2001 (IEEE 1364-2001) SUPPORT

Verilator supports almost all Verilog 2001 language features. This includes signed numbers, "always @\*", comma separated sensitivity lists, generate statements, multidimensional arrays, `localparam`, and C-style declarations inside port lists.

## 16 VERILOG 2005 (IEEE 1364-2005) SUPPORT

Verilator supports the `'begin_keywords` and `'end_keywords` compiler directives.

Verilator supports `$clog2`.

Verilator partially supports the `uwire` keyword.

## 17 SYSTEMVERILOG (IEEE 1800-2005) SUPPORT

Verilator currently has very minimal support for SystemVerilog. As SystemVerilog features enter common usage they will be added. Contact the author if a feature you need is missing.

Verilator implements the full SystemVerilog 1800-2005 preprocessor, including function call-like preprocessor defines.

Verilator supports `==?` and `!=?` operators, `$bits`, `$countones`, `$error`, `$fatal`, `$info`, `$isunknown`, `$onehot`, `$onehot0`, `$warning`, `always_comb`, `always_ff`, `always_latch`,

do-while, and final.

It also supports .name and .\* interconnection.

Verilator partially supports assert.

## 18 SUGAR/PSL SUPPORT

Most future work is being directed towards improving SystemVerilog assertions instead of PSL. If you are using these PSL features, please contact the author as they may be depreciated in future versions.

With the `-assert` switch, Verilator enables support of the Property Specification Language (PSL), specifically the simple PSL subset without time-branching primitives. Verilator currently only converts PSL assertions to simple "if (...) error" statements, and coverage statements to increment the line counters described in the coverage section.

Verilator implements these keywords: `assert`, `assume` (same as `assert`), `default` (for clocking), `countones`, `cover`, `isunknown`, `onehot`, `onehot0`, `report`, `true`.

Verilator implements these operators: `->` (logical if).

Verilator does not support SEREs yet. All assertion and coverage statements must be simple expressions that complete in one cycle. PSL `vmode/vprop/vunits` are not supported. PSL statements must be in the module they reference, at the module level where you would put an `initial...` statement.

Verilator only supports (posedge CLK) or (negedge CLK), where CLK is the name of a one bit signal. You may not use arbitrary expressions as assertion clocks.

## 19 SYNTHESIS DIRECTIVE ASSERTION SUPPORT

With the `-assert` switch, Verilator reads any `//synopsys full_case` or `//synopsys parallel_case` directives. The same applies to any `//cadence` or `//ambit synthesis` 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 runtime and print an "Assertion failed" error message.

## 20 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, like C++’s **\_\_FILE\_\_**.

‘ **\_\_LINE\_\_**

The **\_\_LINE\_\_** define expands to the current line number, like C++’s **\_\_LINE\_\_**.

‘**error** *string*

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

**\_**(*expr*)

A underline followed by an expression in parenthesis returns a Verilog expression. This is different from normal parenthesis in special contexts, such as PSL expressions, and can be used to embed bit concatenation (**{}**) inside of PSL statements.

**\$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 a 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**

Format arguments may use C printf sizes after the **%** escape. Per the Verilog standard, **%x** prints a number with the natural width, **%0x** prints a number with minimum width, however **%5x** prints 5 digits per the C standard (it’s unspecified in Verilog).

**‘coverage\_block\_off**

Specifies the entire begin/end block should be ignored for coverage analysis. Same as `/* verilator coverage_block_off */`.

**‘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. 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. 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. 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. 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. 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. 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 a accessor may result in cleaner code.

**‘verilator****‘verilator3**

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

**‘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 were specified.

**/\*verilator clock\_enable\*/**

Experimental use only. 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.) This will cause the clock gate to be ignored in the scheduling algorithm, improving performance.

**/\*verilator coverage\_block\_off\*/**

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

**/\*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 time with some small loss of trace visibility and modularity. Note signals under inlined submodules will be named *submodule\_\_DOT\_\_subsignal* as C++ does not allow "." in signal names. SystemPerl when tracing such signals will replace the \_\_DOT\_\_ with the period.

**/\*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*/;
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
```



**`/*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_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.

**`/*verilator public*/ (variable)`**

Used after a 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*/`.

**`/*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.)

**/\*verilator public\_flat\*/ (variable)**

Used after a 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.

**/\*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.

**/\*verilator sc\_clock\*/**

Used after a input declaration to indicate the signal should be declared in SystemC as a sc\_clock instead of a bool.

**/\*verilator tracing\_off\*/**

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

**/\*verilator tracing\_on\*/**

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

## 21 LANGUAGE LIMITATIONS

There are some limitations and lack of features relative to a commercial simulator, by intent. User beware.

It is strongly recommended you use a lint tool before running this program. Verilator isn't designed to easily uncover common mistakes that a lint program will find for you.

### Synthesis Subset

Verilator supports only the Synthesis subset with a few minor additions such as \$stop, \$finish and \$display. That is, you cannot use hierarchical references, events or similar features of the Verilog language. It also simulates as Synopsys's Design Compiler would; namely a block of the form

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

will recompute y when there is a change in x or a change in z, which is what Design Compiler will synthesize. A compliant simulator would only calculate y if x changes. (Use verilator-mode's /\*AS\*/ or Verilog 2001's always @\* to prevent these issues.)

## Dotted cross-hierarchy references

Verilator supports dotted references to variables, functions and tasks in different modules. However, references into named blocks and function-local variables are not supported. The portion before the dot must have a constant value; for example `a[2].b` is acceptable, while `a[x].b` is not.

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

Verilator creates numbered "genblk" when a begin: name is not specified around a block inside a generate statement. These numbers may differ between other simulators, but the Verilog specification does not allow users to use these names, so it should not matter.

If you are having trouble determining where a dotted path goes wrong, note that Verilator will print a list of known scopes to help your debugging.

## Floating Point

Floating Point numbers are not synthesizable, and so not supported.

## 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.

## Time

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

## Two State

Verilator is 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.)

First, assigning a variable to a X will actually assign the variable to a random value (see the `-x-assign` switch.) Thus if the value is actually used, the random value

should cause downstream errors. Integers also randomize, even though the Verilog 2001 specification says they initialize to zero.

Identity comparisons (`===` or `!==`) are converted to standard `==`/`!=` when neither side is a constant. This may make the expression result differ from a four state simulator.

All variables are initialized using a function. By running several random simulation runs you can determine that reset is working correctly. On the first run, the function initializes 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.

## Tri/Inout

As a 2 state compiler, tristate and inouts are not supported. Traditionally only chip "cores" are Verilated, the pad rings have been written by hand in C++.

## 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 clocks correctly, however new cases may turn up bugs in the scheduling algorithm. The safest option is to have all clocks as primary inputs to the model, or wires directly attached to primary inputs.

## Ranges must be big-bit-endian

Bit ranges must be numbered with the MSB being numbered greater or the same as the LSB. Little-bit-endian busses [0:15] are not supported as they aren't easily made compatible with C++.

## 32-Bit Divide

The division and modulus operators are limited to 32 bits. This can be easily fixed if someone contributes the appropriate wide-integer math functions.

## 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.

## Array Initialization

When initializing an array, you need 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
  end
```

## 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-sized 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. (Arguably SEREs are much of the point, but one must start somewhere.)

## 22 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`, `'timescale`, `'undef`, `'verilog`

Fully supported.

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

Generally supported.

### `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.

`$bits`, `$countones`, `$error`, `$fatal`, `$finish`, `$info`, `$isunknown`, `$onehot`, `$one-`  
`hot0`, `$readmemb`, `$readmemh`, `$signed`, `$stime`, `$stop`, `$time`, `$un-`  
`signed`, `$warning`.

Generally supported.

### `$display`, `$write`, `$fdisplay`, `$fwrite`

`$display` and friends must have a constant format string as the first argument (as with C's `printf`), you cannot simply list variables standalone.

**\$displayb, \$displayh, \$displayo, \$writeb, \$writeh, \$writeo, etc**

The sized display functions are rarely used and so not supported. Replace them with a \$write with the appropriate format specifier.

**\$finish, \$stop**

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

**\$fopen, \$fclose, \$fdisplay, \$feof, \$fflush, \$fgetc, \$fgets, \$fscanf, \$fwrite**

File descriptors passed to the file PLI calls must be file descriptors, not MCDs, which includes the mode parameter to \$fopen being mandatory. Verilator will convert the integer used to hold the file descriptor into a internal FILE\*. To prevent core dumps due to mis-use, and because integers are 32 bits while FILE\*s may be 64 bits, the descriptor must be stored in a reg [63:0] rather than an integer. The define 'verilator\_file\_descriptor in verilated.v can be used to hide this difference.

**\$fscanf, \$sscanf**

Only integer formats are supported; %e, %f, %m, %r, %v, and %z are not supported.

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

All specify blocks and timing checks are ignored.

**\$random**

\$random does not support the optional argument to set the seed. Use the srand function in C to accomplish this, and note there is only one random number generator (not one per module).

**\$readmemb, \$readmemh**

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

**\$realtime**

Treated as \$time.

## 23 ERRORS AND WARNINGS

Warnings may be disabled in two ways. First, when the warning is printed it will include a warning code. Simply surround the offending line with a warn\_off/warn\_on pair:

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

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.

List of all warnings:

### BLKANDNBLK

BLKANDNBLK is an error that a variable comes from a mix of blocked and non-blocking assignments. Generally, this is caused by a register driven by both combo logic and a flop:

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

Simply use a different register for the flop:

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

This is good coding practice anyways.

It is also possible to disable this error when one of the assignments is inside a public task.

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

### 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.

### 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](http://www.sunburst-design.com/papers/CummingsSNUG1999Boston_FullParallelCase_rev1_1.pdf)

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.

**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.

**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](http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA_rev1).

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

**GENCLK**

Warns that the specified signal is generated, but is also being used as a clock. Verilator needs to evaluate sequential logic multiple times in this situation. In somewhat contrived cases having any generated clock can reduce performance by almost a factor of two. For fastest results, generate ALL clocks outside in C++/SystemC and make them primary inputs to your Verilog model. (However once need to you have even one, don't sweat additional ones.)

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

**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 a cell, and can lead to dangling nets. A better option is the `/*AUTOWIRE*/` feature of Verilog-Mode for Emacs, available from <http://www.veripool.org/>

Ignoring this warning will only suppress the lint check, 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.

**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 runtimes due to reduced optimizations.

Ignoring this warning will only slow simulations, it will simulate correctly.

**MULTITOP**

Error that there are multiple top level modules, that is modules not instantiated by any other module. Verilator only supports a single top level, if you need more, create a module that wraps all of the top modules.

Often this error is because some low level cell is being read in, but is not really needed. The best solution is to insure that each module is in a unique file by the same name. Otherwise, make sure all library files are read in as libraries with -v, instead of automatically with -y.

**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
```

**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.

**TASKNSVAR**

Error when a call to a task or function has a output 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; output 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.

### UNDRIVEN

Warns that the specified signal is never sourced.

Ignoring this warning will only suppress the lint check, 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 runtime.

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 specified 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 runtime; two times better performance may be possible by fixing these warnings.

Unlike the UNOPT warning, this occurs after netlist flattening, 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.

The UNOPTFLAT warning may also be due to clock enables, identified from the reported path going through a clock gating cell. 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.

Ignoring this warning will only slow simulations, it will simulate correctly.

## UNSIGNED

Warns that you are comparing a unsigned value in a way that implies it is signed, for example "X < 0" will always be true when X is unsigned.

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

## UNUSED

Warns that the specified signal is never sinked. This is a future message, currently Verilator will not produce this warning.

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

## 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.

Ignoring this warning will only suppress the lint check, 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 a undeterminate 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.

## Verilated model didn't converge

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

```
always @ (a) b=~a;
always @ (b) a=b
```

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

To debug this, run Verilator with `-profile-cfuncs`. Run `make` on the generated files with `"OPT=DVL_DEBUG"`. Then call `Verilated::debug(1)` in your `main.cpp`.

This will cause each change in a variable to print a message. Near the bottom you'll see the code and variable that causes the problem. For the program above:

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

## 24 FAQ/FREQUENTLY ASKED QUESTIONS

### Does it run under Windows?

Yes, using Cygwin. Verilated output should also compile under Microsoft Visual C++ Version 7 or newer, but this is not tested by the author.

### Can you provide binaries?

Verilator is available as a RPM for SuSE and perhaps other systems; this is done by porters and may slightly lag the primary distribution. If there isn't a binary build for your distribution, how about you set one up? Please contact the authors for assistance.

Note people sometimes request binaries when they are having problems with their C++ compiler. Alas, binaries won't help this, as in the end a fully working C++ compiler is required to compile the output of Verilator.

### How can it be faster than (name-the-simulator)?

Generally, the implied part of the question is "... with all of their manpower they can put into it."

Most commercial simulators have to be Verilog compliant, meaning 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.

Non-compliance shouldn't be scary. Your synthesis program isn't compliant, so your simulator shouldn't have to be – and Verilator is closer to the synthesis interpretation, so this is a good thing for getting working silicon.

### May programs I create with Verilator remain under my own copyright?

Yes, it's just like using GCC on your programs. If you change Verilator itself, you must make the source code available under the GNU Public License. However, the include and generated files use the GNU Lesser Public License, which means that files using them are NOT required to be released.

You also have the option of using the Perl Artistic License, which again does not require you release your Verilog or generated code.

**Why is running Verilator so slow?**

Verilator needs more memory than the resulting simulator will require, as Verilator creates internally all of the state of the resulting simulator in order to optimize it. If it takes more than a minute or so (and you're not using `-debug`), see if your machine is paging; most likely you need to run it on a machine with more memory. Verilator is a full 64 bit application and may use more than 4GB, but about 1GB is the maximum typically needed.

**How do I generate waveforms (traces) in C++ or SystemC?**

See the next question for tracing in SystemPerl mode.

Add the `-trace` switch to Verilator, and make sure the SystemPerl package is installed. SystemC itself does not need to be installed for C++ only tracing. You do not even need to compile SystemPerl; you may simply untar the SystemPerl kit and point the `SYSTEMPERL` environment variable to the untarred directory.

In your top level C code, call `Verilated::traceEverOn(true)`. Then create a `SpTraceVcdC` object, and in your main loop call `"trace_object->dump(time)"` every time step, and finally call `"trace_object->close()"`. For an example, see the call to `SpTraceVcdC` in the `test_c/sim_main.cpp` file of the distribution.

You also need to compile `SpTraceVcdC.cpp` and add it to your link. This is done for you if using the Verilator `-exe` flag.

**How do I generate waveforms (traces) in SystemPerl?**

Add the `-trace` switch to Verilator, and make sure the SystemPerl package is installed.

In your top level C `sc_main` code, call `Verilated::traceEverOn(true)`. Then create a `SpTraceFile` object as you would create a normal SystemC trace file. For an example, see the call to `SpTraceFile` in the `test_sp/sc_main.cpp` file of the distribution.

**How do I view waveforms (traces)?**

Verilator makes standard VCD (Value Change Dump) files. They are viewable with the public domain Dinotrace or GtkWave programs, or any of the many commercial offerings.

**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 a `ifndef` region:

```
'ifndef verilator
    Something_Verilator_Dislikes;
'endif
```

**Why do I get "unexpected 'do'" or "unexpected 'bit'" errors?**

Do, bit, ref, return, and other words are now SystemVerilog keywords. You should change your code to not use them to insure 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 file to be parsed as Verilog 2001, just create a file with

```
'begin_keywords "1364-2001"
```

and add it before other Verilog files on the command line. (Note this will also change the default for `-prefix`, so if you're not using `-prefix`, you will now need to.)

### 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 C++ (non SystemC) code you need to define this function so that the simulator knows the current time. See the "CONNECTING TO C++" examples.

### Why do I get "undefined reference to 'VL\_RANDOM\_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.

### Is the PLI supported?

No.

More specifically, the common PLI-ish calls `$display`, `$finish`, `$stop`, `$time`, `$write` are converted to C++ equivalents. If you want something more complex, since Verilator emits standard C++ code, you can simply write your own C++ routines that can access and modify signal values without needing any PLI interface code, and call it with `$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?

Between GCC 3.0 to 3.3, each compiled progressively slower, thus if you can use GCC 2.95, or GCC 3.4 you'll have faster builds. Two ways to cheat are to compile on parallel machines and avoid compilations altogether. See the `-output-split` option, and the web for the `ccache`, `distcc` and `icecream` packages, and the `Make::Cache` package available from <http://www.veripool.org/>. `Make::Cache` will skip GCC runs between identical source builds, even across different users.



**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 functions/tasks in C?**

Write a Verilog function or task with input/outputs that match what you want to call in with C. Then mark that function public.

Verilog inputs of one bit become C++ bool inputs. Inputs 32 bits or smaller become C uint32\_t inputs, 64-32 bits become C uint64\_t inputs, and wider signals become arrays of 32 bits. Outputs are passed as references to bool, uint32\_t, uint64\_t or uint32\_t[] arrays.

Signals wider than 64 bits are passed 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.)

In the SYSTEMC example above, if you had in our.v:

```
task publicSetBool;
    // verilator public
    input in_bool;
    var_bool = in_bool;
endtask
```

From the sc\_main.cpp file, you'd then:

```
#include "Vour.h"
#include "Vour_our.h"
top->v.publicSetBool(value);
```

See additional notes under the `/*verilator public*/` section.

**How do I access signals in C?**

The best thing is to make a Verilator public task or function accessor that can read or write that signal, as described in the previous FAQ. This will allow Verilator to better optimize the model.

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 chars, 16 bit shorts, 32 bit longs, or 64 bit long longs that fits the width of the signal. Generally, you can use just uint32\_t's for 1 to 32 bits, or uint64\_t for 1 to 64 bits, and the compiler will properly up-convert smaller entities.

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.)

In the SYSTEMC example above, if you had in `our.v`:

```
input clk /*verilator public*/;
```

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

```
#include "Vour.h"
#include "Vour_our.h"
cout << "clock is " << top->v.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 cells, 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 cells as to if they are SystemC or Verilog. Then:

A module with only SystemC cells below must be SystemC.

A module with a mix of Verilog and SystemC cells below must be SystemC. (As Verilator cannot connect to lower-level SystemC cells.)

A module with only Verilog cells 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 cells multiple times into a SystemC module *may* be faster.)

## 25 BUGS

First, check the the coding 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
```

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 test_regress
t/t_BUG.pl
```

Finally, report the bug using the bug tracker at <http://www.veripool.org/verilator>. The bug will become publicly visible; if this is unacceptable, mail the bug report to [wsnyder@wsnyder.org](mailto:wsnyder@wsnyder.org).

## 26 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 Verilator3 by rewriting Verilator from scratch in C++. This added many optimizations, yielding about a 2-5x performance gain.

Currently, various language features and performance enhancements are added as the need arises. Verilator is now about 2x faster than in 2002, and is faster than many popular commercial simulators.

## 27 CONTRIBUTORS

Many people have provided ideas and other assistance with Verilator.

The major corporate sponsors of Verilator, by providing funds or equipment grants, are Compaq Corporation, Digital Equipment Corporation, Maker Communications, Sun Microsystems, Nauticus Networks, and SiCortex.

The people who have contributed code or other major functionality are Paul Wasson, Duane Galbi, and Wilson Snyder. Major testers include Jeff Dutton, Ralf Karge, David Hewson, Wim Michiels, and Gene Weber.

Some of the people who have provided ideas and feedback for Verilator include Hans Van Antwerpen, Jens Arm, David Black, Gregg Bouchard, Chris Boumenot, John Brownlee, Lauren Carlson, Robert A. Clark, John Deroo, Danny Ding, Jeff Dutton, Eugen Fekete, Sam Gladstone, Thomas Hawkins, Mike Kagen, Ralf Karge, Dan Katz, Sol Katzman, Gernot Koch, Steve Kolecki, Steve Lang, Charlie Lind, Dan Lussier, Fred Ma, Wim Michiels, John Murphy, Richard Myers, Paul Nitza, Lisa Noack, Renga Sundararajan, Shawn Wang, Greg Waters, Eugene Weber, Leon Wildman, and Mat Zeno.

## 28 DISTRIBUTION

The latest version is available from <http://www.veripool.org/>.

Copyright 2003-2008 by Wilson Snyder. Verilator is free software; you can redistribute it and/or modify it under the terms of either the GNU Lesser General Public License or the Perl Artistic License.

## 29 AUTHORS

Wilson Snyder <[wsnyder@wsnyder.org](mailto:wsnyder@wsnyder.org)>

Major concepts by Paul Wasson and Duane Galbi.

## 30 SEE ALSO

`verilator_proffunc`, *systemperl*, *vcovrage*, *make*