



ModelSim 5.7 SE Performance Guidelines

**Model Technology
November 2002**

Table of Contents

GENERAL PERFORMANCE CONSIDERATIONS.....	3
WHY IS THIS DOCUMENT IMPORTANT?	3
WHAT ARE YOU MEASURING?	3
HOW ARE YOU MEASURING PERFORMANCE?	4
<i>Measuring Time With Operating System commands</i>	4
<i>Measuring Time and Memory Usage with ModelSim commands</i>	5
USING THE PERFORMANCE ANALYZER	5
OTHER USEFUL TIPS.....	5
VERILOG DESIGNS.....	6
GENERAL VERILOG FLOW	6
<i>RTL considerations</i>	7
<i>Gate level considerations</i>	8
MAINTAINING DESIGN OBJECT VISIBILITY	8
ASSESSING AND INCREASING OPTIMIZATIONS	9
<i>Generating an instance report</i>	9
<i>Cross-referencing reports</i>	9
<i>Relaxing optimization constraints</i>	10
USING -FORCECODE.....	11
VHDL DESIGNS.....	11
MIXED HDL DESIGNS	12
USING ELABORATION FILES TO IMPROVE REGRESSION TEST THROUGHPUT	13
IMPROVING GATE-LEVEL PERFORMANCE WITH SIMULATOR OPTIONS	14
IMPROVING PERFORMANCE BY RESERVING MEMORY	14
<i>LOCKING MEMORY ON HPUX 10.2 AND 11.0</i>	14
<i>ENABLING SHARED MEMORY ON SUN/SOLARIS</i>	16

General performance considerations

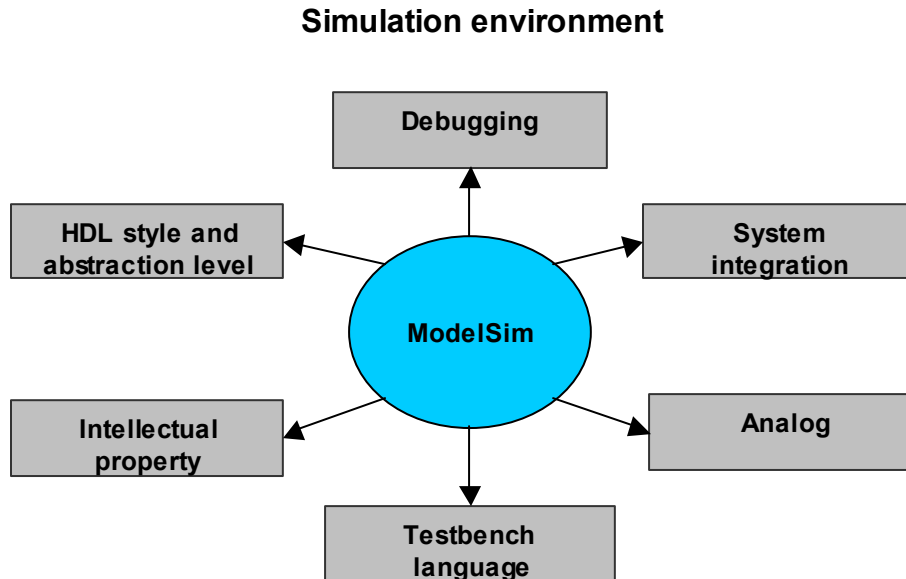
Any discussion about simulator performance should begin with two questions: “*What are you measuring?*” and “*How are you measuring it?*” This section addresses those questions and discusses a few other general considerations for analyzing and improving performance.

Why is this document important?

The difference in optimizing and not optimizing a simulation is very dramatic. Can your environment run 2x, 4x or even 10x faster? There is no solid number that is the same for every environment. By following the flow described in this document you can maximize ModelSim performance for your Simulation environment.

What are you measuring?

HDL simulation is only part of verification. You must determine the impact of the entire environment when considering simulation performance. The picture below illustrates the various flows that may affect your simulation:



Any of these flows may negatively impact simulation performance. The HDL code may be inefficient; testbench languages and 3rd party debug tools may slow the simulation; gate-level IP may be un-optimized. Consider that 3rd party testbench implementation alone often account for greater than 80% of the overall simulation performance. If this is the case you should consider investigating the reason 80% of the time is being spent on the testbench and it's interface. The Performance Analyzer™ in ModelSim (discussed below) may help you identify which of these flows is impacting simulation speed the most.

Turning to the simulator itself, it's critical to realize simulators are run in two modes, interactive and batch. Interactive mode is generally associated with debugging where maximum visibility into the design is needed. Batch mode jobs are run in the background without the User Interface (UI). Performance is generally the highest priority when running in batch mode. Simulators in optimized performance mode remove visibility into a design. ModelSim, especially with Verilog is in debug mode. For performance you must employ compiler optimization switches. Therefore, to accurately compare simulation results you must realize all simulators have two modes: optimized for performance and un-optimized for debug. Make sure you are comparing the same modes.

Another consideration is what part of the simulation you are measuring. ModelSim has separate compilation and simulation steps. Furthermore simulation is a two-phase process. During phase 1 (known as elaboration), ModelSim generates native code for your specific OS. During phase 2, ModelSim runs the native code. You'll gain the most accurate performance statistics by measuring the elaboration phase and run phase separately. As discussed below, you can use the **-elab** switch or the **ps** command to measure these two phases independently.

How are you measuring performance?

Different measurement methods may report different simulation times. Simply stop-watching a simulation may not produce an accurate measurement.

Measuring Time With Operating System commands

There are two types of time: "wall clock time" and "cpu time". If the ModelSim simulator, **vsim**, is the only process running on a machine, these two times should be approximately the same. However, if other jobs are taking a large percentage of the machine's processor time, "wall clock time" will not accurately represent true simulation time. Measure cpu time to eliminate interference from unrelated processes.

The **ps** command provides one way of measuring cpu time. The time it reports is the cpu time since the process was started. This command:

```
exec ps -ef | grep vsim
```

returns the following:

```
user 25508 25507 48 16:12:09 pts/9    29:13 vsim top
```

This example shows a **vsim** process that has been running for 29 minutes and 13 seconds.

Unless you are using the **vsim -elab** option (discussed below), you must execute the **ps** command twice to obtain the run time. For example the command shown below would report two times. The first time reflects elaboration time and the second reflects the total time for both elaboration and simulation. To get the simulation time alone, subtract the first time from the second time.

```
vsim -c -L cell_lib top -do "exec ps -ef | grep vsim; run -all; exec ps -ef | grep vsim"
```

Another way of measuring time is the UNIX **time** command. However, this command does not allow you to separate elaboration and run times. This command:

```
time vsim -c -L cell_lib top -do "run -all"
```

returns one of two formats depending on the shell:

```
11.0u 38.0s 1:45 46% 0+0k 0+0io 0pf+0w
```

or

```
real    1:45.5
user    0:11.2
sys     0:38.2
```

The three times in the first example are user (u), system (s), and real. The system time represents the sum of elaboration and run times. The real time is "wall clock time."

The numbers from the time command may be misleading due to heavy system load. To check the numbers' accuracy, sum the user and system times. The total should be pretty close to the real time. If it is not, there is a large load on the system, and you shouldn't rely on the numbers.

Measuring Time and Memory Usage with ModelSim commands

The ModelSim **simstats** command reports various statistics about the current simulation. Executing **simstats** on the ModelSim command line returns the following:

```
{memory 7856} {{working set} 6032} {time 17.1826} {{cpu time} 17.1} {context 2} {{page faults} 1}
```

where:

memory = Total memory being allocated for the ModelSim process
working set = Portion of total memory in use for the current simulation
time = Cumulative “wall clock time” for run commands
cpu time = Cumulative processor time for run commands
context = The number of context swaps that have occurred during the run commands (vsim being swapped out for another process)
page faults = The number of page faults the have occurred during the run commands - a large number can indicate insufficient physical memory

The **simstats** command uses OS calls for its information. Not all OSs support queries for every parameter, in which case **simstats** will return zero for that parameter. If you are using **simstats** in batch mode it may be necessary to use the **echo** command to force the results to be written to the transcript, instead of “**simstats**” use “**echo [simstats]**”.

Using the Performance Analyzer

The Performance Analyzer identifies bottlenecks in your design. Once these bottlenecks are corrected, you should see substantially faster simulations.

To enable the Performance Analyzer, invoke the **profile on** command before the simulation run begins. After the simulation stops, invoke **profile report -file profile.rpt** to save the results. These commands can also be used interactively with the UI.

One option of note is the **keep_unknown** argument. This argument tells the analyzer to keep statistics about items not found in the HDL code. This helps locate bottlenecks in FLI/PLI routines, third-party interfaces, and the like. Use this command to enable the argument:

```
profile option keep_unknown
```

See the Performance Analyzer chapter in the ModelSim SE User’s Manual for further details.

NOTE: The Performance Analyzer can increase simulation times by up to 10%. Therefore, do not use it when timing simulations. Invoke the analyzer in a separate run.

For most accurate line number information when using the +opt or -fast Verilog compile options also use the +acc=l compile option. This insures that line number information is available for the Performance Analyzer.

Other useful tips

- Another useful command for measuring memory footprints is the **mti_kcmd memstats** command. Execute this command from the VSIM> prompt to print memory statistics to the shell window that invoked ModelSim. For windows OS this command must be run from a dos shell. If ModelSim is invoked from a shortcut on a Window’s machine, no information is returned.
- Compile and run designs from a local drive/disk whenever possible. Network traffic can significantly slow processes that require large amounts of file I/O. If you have a large numbers of files, you may want to copy them to a local disk prior to compiling and simulating.

- Avoid using the GUI when running benchmarks. The GUI adds overhead and is not needed unless you are interactively debugging a design.
- To maximize ModelSim performance, use the flows based on the mix of HDL in your environment. The following sections discuss flows for specific HDL mixes.
- Make sure you are running in the highest simulation resolution possible. For example, do not run in ps mode if ns resolution is functional.
- Make sure that you have enough physical memory to run the process. Swapping to virtual memory can significantly impact performance of any run. Choose the right machine for the job.
- Monitor the load of the machine on which you are running. A machine with multiple jobs competing for CPU and memory resources will impact wall clock run time. Also multi-cpu machines must compete for the same memory interface and will impact the run time of a job.
- ModelSim has support for 32- and 64-bit OS. The 32-bit OS memory address limit is 4GB. For simulation jobs that require more than 4Gb of memory, you must use 64-bit OS versions of ModelSim. The use of 64-bit OS version of ModelSim should be restricted to those jobs that require more than 4Gb of memory to run. 64-bit OS versions use approximately 30% more memory and are approximately 30% slower than 32-bit versions of the same OS.
- The use of self-checking testbenches to eliminate the need for file IO can improve performance.

Verilog designs

General Verilog flow

There are two optimization flows for Verilog design: Verilog RTL and Verilog gate. The general flows are the same with some noted exceptions. For a more complete discussion, see the Verilog chapter in the *ModelSim SE User's Manual*.

It's critical to realize simulators are run in two modes, interactive and batch. Interactive mode is generally associated with debugging where maximum visibility into the design is needed. Batch mode jobs are run the background without the User Interface (UI). The opportunity for increased performance is generally the highest when running in batch mode since you generally need less visibility into the design. Simulators in optimized performance mode remove visibility into a design. ModelSim default mode with Verilog is in debug mode, which provides highest visibility. For performance you must employ compiler optimization switches. Therefore, to improve ModelSim simulation results you must engage the global compiler optimizations.

Improving Verilog performance starts with using compiler optimization arguments. You can increase simulation speed significantly by compiling with the **+opt** global optimization compile argument. This option merges always blocks, in-lines instantiated modules, and performs cell-level optimizations. It also reduces or eliminates events and improves memory management.

ModelSim's Verilog compiler "vlog" has two global optimization switches. They are very similar in that they both engage the same performance algorithms. The main differences are that you may use **+opt** to update a previously compiled non-optimized design, as in the example 4 below. The **-fast** does not allow optimization of previously optimized designs, but does support incremental compile. If you changed one file in a long list of previously compiled files you can incrementally compile only the source file that was modified by using **-incr** option with **-fast** (**vlog -f list.f -fast -incr**). Remember to use the original full vlog compile options when using **-incr**. The **-incr** option does not work with **+opt**.

The sample compile scripts below demonstrate several methods for compiling a Verilog design with **+opt**. The examples use two other compiler arguments: **-O5**, which optimizes loops and case statements, and **-debugCellOpt**, which prints messages regarding cell-level optimizations. All examples are appropriate for designs with RTL, gates, or both.

NOTE: ModelSim recognizes a module as a gate if the module contains a non-empty specify block. Earlier versions of ModelSim identified gate cells using the compiler directive ``celldefine`. This is no longer the case.

```
##### Verilog compile script example 1 #####
#
# +opt option enables the global optimizations
#
vlib work
vlog -O5 +opt -debugCellOpt tcounter.v counter.v
#
##### end compile script example 1 #####
```

If you have a more extensive list of files, you can use the `-f` compile argument to specify a text file that contains a list of your design's files. In the example below, *list.f* includes *tcounter.v* and *counter.v*.

```
##### Verilog compile script example 2 #####
#
# +opt option enables the global optimizations
# -f option will use the file to get list of files to compile
#
vlib work
vlog -O5 +opt -debugCellOpt -f list.f
#
##### end compile script example 2 #####
```

If you have multiple, pre-compiled libraries, you can use the `-L` compiler argument to access them while using `+opt`. In the next example, the counter was compiled into a separate library (*dut*), and the testbench into the default work library. The `-L` argument makes the *counter* module visible when you compile the testbench.

```
##### Verilog compile script example 3 #####
#
# The counter module is compiled into the library dut
# The testbench module is compiled into the default work library
# +opt option enables the global optimizations
# -L option will provide access to the counter module
#
vlib work
vlib dut
vlog -O5 -work dut counter.v
vlog -O5 +opt -debugCellOpt -L dut tcounter.v
#
##### end compile script example 3 #####
```

Note that *counter.v* was *not* compiled with `+opt`. When the top-level cell in *tcounter.v* is compiled with `+opt`, all instances in the hierarchy are optimized, including any modules in the *dut* library.

`+opt` can also handle designs that were previously compiled without optimizations. This typically occurs when designers are moving from a debug phase to a regression phase. The following example uses a variation of the `+opt` argument to optimize a previously compiled design. In this example the *counter* module is compiled in the library *dut* as in example 3, and the testbench *tcounter* is compiled into the default work library. Neither of these modules has been optimized at this point.

```
##### Verilog compile script example 4 #####
#
# +opt+tcounter option enables the global optimizations
# Note only the top level module name and library references are needed
#
vlog -O5 +opt+tcounter -debugCellOpt -L dut
#
##### end compile script example 4 #####
```

RTL considerations

With RTL designs, verify that the modules are being in-lined by the compiler. When invoked with the `+opt`

argument, the compiler reports how many modules are in-lined:

```
# Analyzing design...
# Optimizing 48 modules of which 24 are inlined:
```

This indicates that **+opt** in-lined 24 of the 48 modules in the design. 50% module in-lining is low. The greater the percentage of in-lined modules, the better the performance. If you have a low percentage of in-lined modules, please contact your ModelSim support personnel.

You should also try to optimize any gate-level cells in an RTL design. RTL designs often have gate-level cells, and sometimes you may not even know they have been added. For example it is typical to add IO pad cells as a project nears completion. Un-optimized gate-level cells significantly impact RTL performance. In addition to using the **-debugCellOpt** compiler argument to identify un-optimized cells, you can generate reports on modules and their optimization with the **write cell_report** or the **write report** command. See “Assessing and increasing optimizations” below for details.

Gate level considerations

In most cases the examples presented above will work equally well for both gate-level and RTL designs. However, multi-million gate netlists may compile slowly with **+opt**. If netlist compile time is an issue for a gate-level design, you may prefer to use the following modified flow:

- Create separate work directories for the cell library and the rest of the design.
- Compile only the cell library using **+opt**.
- Compile the device under test and testbench *without* **+opt**.
- If supported by your platform, reserve system memory for **vsim**. See “Improving Performance by Reserving Memory” below for details.

However, because this flow does not perform global optimizations on the testbench, you may see slower simulation performance than when using **+opt** on the whole design. Consider the tradeoff between netlist compile time and complete optimization.

Another caveat to the modified flow is that it can cause problems if the testbench has hierarchical references into the cell library. Optimizing the library alone results in unresolved references. In such cases you must use the original flow. The original flow considers hierarchical cell references before enacting optimizations.

Finally, if you have a choice between VHDL Vital and Verilog, use Verilog. With Verilog cells and a Verilog netlist (regardless of testbench language), performance can be 4-8x faster than the same design in VHDL Vital. The memory footprint will also be 4-8x smaller.

Maintaining design object visibility

Some of the optimizations performed by **+opt** may impact design visibility of nets, ports, and registers. If you need to maintain access to these objects for debugging purposes, use the **+acc** option in conjunction with **+opt**. Keep in mind, however, that enabling design object access may reduce simulation performance.

For example suppose you need to dump nets and registers of a particular instance in the design using the \$dumpvars system task. You would have something like the following \$dumpvars call in your testbench:

```
initial $dumpvars(1, testbench.u1);
```

In this case, compile your design as follows to enable net and register access for the module (assuming testbench.u1 refers to a module *design*):

```
% vlog +opt +acc=rn+design testbench.v design.v
```


For a more detailed discussion of the **+acc** option, see “Enabling design object visibility with the **+acc** option” in the Verilog simulation chapter of the *ModelSim User’s Manual*.

Assessing and increasing optimizations

Generating an instance report

For designs that contain cells, always verify that cells with the highest instance counts are being optimized. Use the **write cell_report** commands to generate a list of all instances in the design and then cross-reference this with the output from **-debugCellOpt** (see Cross-referencing reports below). It’s possible you can “force” a cell to be optimized, thereby improving performance.

For example, consider the compile script below:

```
##### Verilog Gate Compile Script Example #####
#
# shell commands to help remove directories
# IMPORTANT: gate-level libraries can be enormous
# move them instead of removing as part of the
# script. This will make the scripts run faster
# ANYTHING you can do to make the compile go faster!!!!
# you can remove the *_remove directories as a background task
#
# +opt option is used to enable optimizations
# -debugCellOpt will provide optimization information
# compile.txt will be used as a cross-reference
#
touch work asic_lib_fast
mv work /tmp/work_remove
mv asic_lib_fast /tmp/asic_lib_fast_remove
# compile the asic library
vlib work
vlib asic_lib_fast
vlog -work asic_lib_fast asic_lib_src/*.v
# compile the rest of the design using +opt and reference the library with the -L
vlog +opt -debugCellOpt -L asic_lib_fast ./src/device.v ./src/Teststring.v > compile.txt
#
##### End Verilog Gate Compile Script Example #####
```

The **write cell_report** command identifies whether cells have or have not been optimized. Once the design has been compiled, invoke **vsim** on the top level (tb) as follows:

```
vsim tb -L asic_lib_fast -do "write cell_report report_cell.txt; quit -f"
```

The **write cell_report** output is sorted by instance count. In this example the *report_cell.txt* file would contain this type of information:

```
3600 of FF_PRE are Optimized

1823 of cellA are Optimized

384 of FF are Not Optimized

338 of cellB are Optimized
```

Cross-referencing reports

Once you have output from **write cell_report** you can compare it against the information generated during compilation. If any of the most instantiated cells are not optimized, you should try to optimize them. Suppose that cell FF is instantiated frequently, and it is not optimized. You might see a message like the following in the compilation output:

```
-- Optimizing module CELL_OR7(fast)
-- Optimizing module CELL_OR2(fast)
```

```
-- Optimizing module FF(fast)
WARNING[10]: asic_lib_src/FF.v(26): Not optimizing library module because the UDP has
non-zero delay
WARNING[10]: asic_lib_src/FF.v(10): Module FF could not be compiled as an optimized
cell
```

This type of issue occurs often and can be resolved easily. The extracted code below shows that line 26 from *FF.v* has a structural delay (#.01(out_i,clk_i, input, en, sense, reset);). This type of delay is not supported with cell library accelerations.

```
and(clken, rstn, en);
and(reset_enable, rstn, en);
buf(out_e, out_i);

`ifdef func
    ff_udp (out_i, clk_i, input, en, sense, reset);
`else
    ff_udp #0.01(out_i, clk_i, input, en, sense, reset);
`endif
```

There are two options for optimizing this cell. The first is to use the **vlog** compiler argument **+delay_mode_path**. This argument causes the compiler to ignore all non-zero delays. The command below demonstrates the use of this argument:

```
vlog -work asic_lib_fast +opt -debugCellOpt +delay_mode_path ./asic_lib_src/FF.v
```

The second option is to define the compile variable *func*. This variable is used to selectively instantiate either the delayed or non-delayed version of the UDP. To employ the functional, non-delayed output version of the UDP instance *ff_udp*, invoke the following command:

```
vlog -work asic_lib_fast +opt -debugCellOpt +define+func ./asic_lib_src/FF.v
```

Regardless of which method you use, the new compile results will look as follows:

```
Model Technology ModelSim SE vlog 5.5 Compiler 2000.12 Dec 14 2000
-- Compiling module FF
-- Compiling UDP ffsrce

Top level modules:
  FF

Analyzing design...
Optimizing 2 modules of which 0 are inlined:
-- Optimizing UDP ff_udp(fast)
-- Optimizing module FF(fast)
NOTE: asic_lib_src/FF.v(10): Optimizing cell module FF
NOTE: asic_lib_src/FF.v(10): All path delays specified for module FF were simple
```

Relaxing optimization constraints

Another way to gain performance is by reducing optimization constraints. **+opt** uses fairly conservative algorithms to implement optimizations. This reduces the chance of incorrect results but also impacts simulation performance. Most designs can be simulated correctly with these constraints removed; however, results should always be checked if the constraints are removed.

The **+nocheck** arguments described below remove these constraints. See the *ModelSim Command Reference* for complete syntax.

Argument	Description
+nocheckALL	Enables all +nocheck arguments described below

+nocheckCLUP	Allows connectivity loops in a cell to be optimized
+nocheckDNET	Allows both the port and the delayed port (created for negative setup/hold) to be used in the functional section of the cell.
+nocheckOPRD	Allows an output port to be read internally by the cell. Note that if the value read is the only value contributed to the output by the cell, and if there's a driver on the net outside the cell, the value read will not reflect the resolved value.
+nocheckSUDP	Allows a sequential UDP to drive another sequential UDP.

Using -forcecode

As noted previously, **+opt** attempts to in-line (combine) any module(s) that are referenced. When this in-lining occurs, no *fast.asm* file is created in the in-lined modules' library directory. Without a *.asm* file, the simulator could not instantiate this module directly. The **-forcecode** argument ensures that the *.asm* file is generated for in-lined modules.

For example suppose that you have four library cells, *inv_a*, *inv_b*, *inv_c*, and *inverter*. The cell *inverter* is instantiated in each of the other cells (*inv_a*, *inv_b*, and *inv_c* might be some sort of timing wrappers). If these cells are compiled with **+opt** and without **-forcecode**, only *inv_a*, *inv_b*, and *inv_c* will have a *fast.asm* file generated in their library directory. The cell *inverter* will not have a *fast.asm* file in its library directory. Alternatively, when these cells are compiled with **-forcecode**, the compiler will create this file for every cell regardless if it has been in-lined. We recommend that customers use this switch when compiling all gate-level libraries.

NOTE: In-lining will occur only if the lower-level module is not a cell. ModelSim 5.6 and later treats any module with a non-empty specify block as a cell. ModelSim 5.5 and earlier used the compiler directive ``celldefine` to identify cells. Also, ModelSim never in-lines UDPs.

VHDL designs

For most designs, ModelSim VHDL is optimized for performance with the default compiler options. Some designs with many "for" loops or many arrays may simulate faster if you use additional compiler arguments. The **-O5** option implements additional compiler optimizations, especially for loops. The **-nocheck** arguments eliminate checks for out-of-bounds scalar assignments or out-of-bound access to arrays. These arguments are summarized below:

Argument	Description
-nocheck	Disable run-time range and index checks
-noindexcheck	Disable run-time index checks
-norangecheck	Disable run-time range checks
-O5	Enable additional compiler optimizations

In a mixed HDL environment you can optimize for performance using the `+opt` on sections of hierarchy contain Verilog. In the figure below, compiling the top Verilog instances with `+opt` will optimize the two areas of Verilog hierarchy.

VHDL RTL users find that using Verilog Gate level netlist and Verilog Gate libraries run much faster than using VHDL and VITAL. Using the existing VHDL testbench and Verilog gates is a very common Mixed HDL flow. In addition to this Mixed Gate level flow, many projects are now using both VHDL and Verilog RTL flows. The +opt is well suited for both RTL and Gate flows.

```
# WARNING[10]: design.v(1507): Instantiation of VHDL entity "low_level" is not optimized.
```

The top level Verilog modules in the diagram above are `top_left` and `top_right`, and are initially compiled into the default work library. They also refer to instances in a `spare_parts` library. The `-L` option is used to refer to this `spare_parts` library located elsewhere on the network. The following command will optimize from the two highest Verilog modules.

Model Technology

Using this mixed HDL optimization flow can greatly improve your simulation run time. Prior to the 5.6b release you could optimize only the top_left hierarchy, now with the new flow in 5.6b you can optimize both Verilog hierarchy top_left and top_right.

Using elaboration files to improve regression test throughput

Elaboration refers to the process of generating native code for your platform. The ModelSim simulator, **vsim**, elaborates every time you load a design. If elaboration is a significant part of your overall simulation run time, you can isolate the elaboration phase to improve your throughput. In other words, you create an elaboration file once, and then simulate it multiple times. Elaboration files can be used for RTL or gate-level runs.

For example a multi-million, gate-level run may take 20 minutes to elaborate and annotate SDF timing, and an additional 20 minutes to run. A second run with different testbench stimulus also takes 20 minutes to load and 20 minutes to run. If you generate an elaboration file on the first run, you eliminate the 20-minute elaboration and SDF annotation time for the second and subsequent runs. Loading an elaboration file takes seconds, instead of minutes.

In many cases design-loading time is not that important. For example if you're doing "iterative design," where you simulate the design, modify the source, recompile and re-simulate, the load time is just a small part of the overall flow. However, if your design is locked down and only the test vectors are modified between runs, loading time may materially impact overall simulation time, particularly for large designs loading SDF files.

Another reason to use elaboration files is for benchmarking purposes. Other simulator vendors use elaboration files, and they distinguish between elaboration and run times. If you are benchmarking ModelSim against another simulator that uses elaboration, make sure you use it with ModelSim as well so you're comparing like to like.

The **vsim** arguments for creating and using elaboration files are summarized below. See the *ModelSim Command Reference* for complete details.

Argument	Description
-elab <filename>	Creates an elaboration file
-load_elab <filename>	Loads an elaboration file
-compress_elab	Compresses an elaboration file when it is created
-filemap_elab	Establishes a map between files named during the original elaboration file generation, and alternate file(s) to be used for subsequent runs.

Design considerations for use of elab option

For gate level designs it is best to specify SDF annotation files on command line. If you use `$sdf_annotate()` task, it must be in an init block so that it is included in the elaboration file.

Test vectors should be read from a file. The load elab feature has support for file mapping (`-elab_filemap`) so that a single elab image can read different files.

Improving gate-level performance with simulator options

As noted earlier, ModelSim's default simulation behavior promotes maximum debugging capability. However, you can specify simulator arguments that will promote simulation speed instead.

The following arguments to `vsim` will improve performance when simulating gate-level Verilog designs. Keep in mind that you are disabling functionality by using these arguments.

Argument	Description
<code>+notimingchecks</code>	Disables Verilog and VITAL timing checks for faster simulation. By default, Verilog timing check system tasks (\$setup, \$hold,...) in specify blocks are enabled. For VITAL, the timing check default is controlled by the ASIC or FPGA vendor, but most default to enabled.
<code>+nonotifier</code>	Speeds simulation by disabling unknown (X) propagation for timing constraint violations. Timing messages for the violations are still issued.

Your **vsim** command might look like this:

```
vsim tb -L asic_lib_fast +notimingchecks +nonotifier -do run -all; quit -f"
```

NOTE: **+notimingchecks** is also a compiler option. Using **+notimingchecks** at compile time reduces the memory footprint, since the data structures for the timing information are not generated.

Improving performance by reserving memory

HP and Sun both allow you to reserve memory for specific processes. This can significantly increase performance, particularly with large simulations. The sections below discuss these methods.

Locking memory on HPUX 10.2 and 11.0

Memory locking on HPUX serves two purposes on the 10.2 platform, allows larger page sizes and allows memory addressing above 2GB. The 11.0 platform includes large page sizes as part of the OS. Therefore the only benefit memory locking provide for 11.0 is memory addressing above 2GB.

ModelSim 5.3 and later versions contain a feature to allow HPUX to use locked memory. This feature provides significant acceleration of simulation time for large designs – i.e. with a memory footprint > 500Mb. (Test cases showed 2x acceleration of large simulations.) The following steps show how to set up HPUX so memory can be locked.

- 1 Allow the average-user to lock memory. By default, this privilege is not allowed, so it has to be enabled. To allow everyone MLOCK privileges, the administrator needs to execute this command on the machine that will be running ModelSim:

```
/usr/sbin/setprivgrp -g MLOCK
```

To only allow a particular group MLOCK privileges, use the command:

```
/usr/sbin/setprivgrp <group-name> MLOCK
```

This allows you to lock memory. No other privileges are enabled.

- 2 Once the MLOCK privilege is enabled, you merely have to modify the modelsim.ini file, and add the following entry to the [vsim] section:

```
LockedMemory = <some-value>
```

where *<some-value>* is an integer representing the number of megabytes of memory to be locked. Once this is done, the memory will be locked when vsim invokes (using this .ini file).

ModelSim will not lock more memory than is available in the system. The maximum memory that can be locked is: system physical memory (RAM) - 100 Mb = locked memory

When ModelSim locks memory, other processes will not have access to it. Therefore, you should consider how much memory is locked on a per-design basis to avoid locking more than is needed.

System parameters used for shared/locked memory may not be set (by default) high enough to take full advantage of this feature in later generations of HPUX. Using the "sam" program, go to the "Configurable Parameters" window (under "Kernel Configuration"). There are several values that may need to be increased.

First, enable shared memory. The value for "shmem" should be equal to 1. Set the value for "shmmax" as large as possible. The defaults for the values of "shmmin" and "shmseg" should be ok. To change these parameters, you have to rebuild the kernel and reboot.

Enabling shared memory on Sun/Solaris

Starting with the ModelSim 5.5b you can improve simulation performance on Sun/Solaris by enabling shared memory. Up to a 2x improvement has been seen in large, Verilog gate-level simulations.

Follow these steps to use the feature:

- 1 Enable a large shared memory segment by adding the following line to the /etc/system file:

```
set shmsys:shminfo_shmmax=0xffffffff
```

- 2 Reboot your machine.

- 3 *Immediately* after the machine has been rebooted, run the program **vshminit** (found in the modeltech tree). The program takes a single parameter that is the amount of memory in megabytes to reserve for use by the simulator. For example, running **vshminit** like this:

```
<modeltech-tree>/sunos5/vshminit 700
```

would reserve 700mb of space for use by the simulator. The next time you run the simulator it will automatically detect the reserved memory and use it.

Important: **vshminit** must be run immediately after you reboot the machine. (You might want to add the program to the system startup scripts.) There may be no performance benefit if you don't run it immediately after reboot.

The amount of memory you supply as a parameter to **vshminit** depends on the configuration of your system. Typically you might want to reserve 50-80% of the system memory for the simulator, depending on whether the machine is multi-use or is dedicated to running simulations.

To free the memory reserved by **vshminit**, execute the following command:

```
/bin/ipcrm -M 0x10761364
```