cadence®

Gate-Level Simulation Methodology

Improving Gate-Level Simulation Performance

Author: Gagandeep Singh

The increase in design sizes and the complexity of timing checks at 40nm technology nodes and below is responsible for longer run times, high memory requirements, and the need for a growing set of gate-level simulation (GLS) applications including design for test (DFT) and low-power considerations. As a result, in order to complete the verification requirements on time, it becomes extremely important for GLS to be started as early in the design cycle as possible, and for the simulator to be run in high-performance mode. This application note describes new methodologies and simulator use models that increase GLS productivity, focusing on two techniques for GLS to make the verification process more effective.

Contents

Objective

The purpose of this document is to present the best practices that can help improve the performance of gate-level simulation (GLS). These best practices have been collected from our experience in gate-level design, and also based on the results of the Gate-Level Methodology Customer Survey carried out by Cadence.

Starting GLS early is important because netlist modifications can continue late into the design cycle, and are driven by timing-, area-, and power-closure issues. It is also equally important to reduce the turn-around time for GLS debug by setting up a flow that enables running meaningful GLS focused on specific classes of bugs, so that those expensive simulation cycles are not wasted toward re-verifying working circuits.

This application note describes new methodologies and simulator use models that increase GLS productivity, focusing on two techniques for GLS to make the verification process more effective:

- Extracting information from the tools applied to the gate netlist, such as static timing analysis and linting. Then, passing the extracted information to the GLS.
- Improving the performance of each GLS run by using recommended tool settings and switches

These approaches can help designers focus on the verification of real gate-level issues and not spend time on issues that can be caught more efficiently with other tools. The new methodologies and simulator use models described in this document can increase GLS productivity, which is important because productivity measures both the number of cycles executed and the throughput of the simulator. Addressing the latter without addressing the former results in only half of the productivity benefit.

The first part of this document presents information on fine-tuning Cadence® Incisive® Enterprise Simulator to maximize cycle speed and minimize memory consumption. The second part is dedicated to general steps designers can take with the combination of any simulator, synthesis tool, DFT engine, and logic equivalence checking (LEC) engine.

While only Incisive Enterprise Simulator users will find real benefits in the first section, all GLS users will find value in this entire application note.

Introduction

Project teams are finding that they need more GLS cycles on larger designs as they transition to finer process technologies. The transition from 65nm to 40nm, and the growth of 28nm and finer design starts, are driven by the need to access low-power and mixed-signal processes (data from IBS, Inc.). These design starts do account for some required increase in GLS cycles because they translate into larger designs. However, it is the low-power and mixed-signal aspects, as well as the new timing rules below 40nm, of these designs that are creating the need to run more GLS simulations. Given that the GLS jobs tend to require massive compute servers, and run for hours, even days and weeks, they are creating a strain on verification closure cycles.

Simulation providers are continuing to improve GLS execution speed and reduce memory consumption to keep up with demand at the finer process nodes. While faster engines are part of the solution, new methodologies for GLS are also required.

A closer examination shows that many teams are using the same approaches in test development and simulation settings to run GLS today as they did in the 1980s when Verilog-based GLS began. Given that the size of today's designs was almost inconceivable in the 1980s, and the dependencies on modern low-power, mixed-signal, and DFT didn't exist then, new methodologies are now warranted. The sections in this document describe these new methodologies in detail.

Gate-Level Simulation Flow Overview

The typical RTL to gate-level netlist flow is shown in the following illustration.

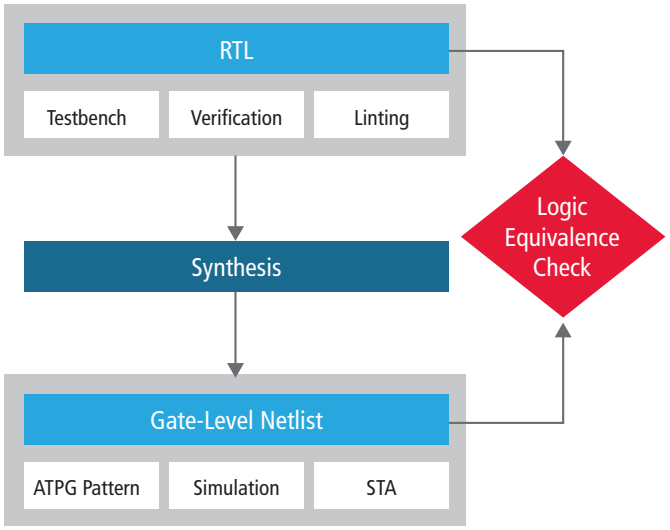


Figure 1: Gate-Level Simulation Flow

Why Gate-Level Simulation is Required

GLS can catch issues that static timing analysis (STA) or logical equivalence tools are not able to report. The areas where GLS is useful are in:

1. Overcoming the limitations of STA, such as:
   - The inability of STA to identify asynchronous interfaces
   - Static timing constraint requirements, such as those for false and multi-cycle paths

2. Verifying system initialization and that the reset sequence is correct

3. DFT verification, since scan-chains are inserted after RTL synthesis

4. Clock-tree synthesis

5. For switching factor to estimate power

6. Analyzing X state pessimism or an optimistic view, in RTL or GLS

Techniques to Improve Gate-Level Performance

This section gives you key techniques that can help reduce GLS run time and debug time. The section is presented in two parts:

1. Improving Gate-Level Simulation Performance with Incisive Enterprise Simulator

2. A Methodology for Improving Gate-Level Simulation

Improving Gate-Level Simulation Performance with Incisive Enterprise Simulator

This section describes techniques that can help improve the performance of GLS by running Incisive Enterprise Simulator in high-performance mode using specific tool features.

1. Applying More Zero-Delay Simulation

While timing simulations do provide a complete verification of the design, during the early stages of GLS, when the design is still in the timing closure process, more zero-delay simulations can be applied to verify the design functionality. Simulations in zero-delay mode run much faster than simulations using full timing.

Zero-delay mode can be enabled using the `-NOSpecify` switch. This option works for Verilog designs only and disables timing information described in *specify* blocks, such as module paths and delays and timing checks. For negative timing checks, delayed signals are processed to establish correct logic connections, with zero delays between the connections, but the timing checks are ignored.

Since zero-delay mode can introduce race conditions into the design, and can also introduce zero delay loops, Incisive Enterprise Simulator has many built-in delay mode control features that can help designers run zero-delay simulations more effectively. These features are listed in the following sections.

1.1 Controlling Gate Delays

Incisive Enterprise Simulator provides delay mode control through command-line options and compiler directives to allow you to alter the delay values. These delays can be replaced in selected portions of the model.

You can specify delay modes on a global basis, or on a module basis. If you assign a specific delay mode to a module, then all instances of that module simulate in that mode. Moreover, the delay mode of each module is determined at compile time and cannot be altered dynamically. There are delay mode options that use the plus (+) option prefix, and the 12.2 and later releases of the tool include a minus (-) delay mode option.

The (+) options listed below are ordered from the highest to lowest precedence. When more than one plus option is used on the command line, the compiler issues a warning and selects the mode with the highest precedence.

delay_mode_path

This option causes the design to simulate in path delay mode, except for modules with no module path delays. In this mode, Incisive Enterprise Simulator derives its timing information from specify blocks. If a module contains a specify block with one or more module path delays, all structural and continuous assignment delays within that module, except *trireg* charge decay times, are set to zero (0). In path delay mode, *trireg* charge decay remains active. The module simulates with black box timing, which means it uses module path delays only.

delay_mode_distributed

This option causes the design to simulate in distributed delay mode. Distributed delays are delays on nets, primitives, or continuous assignments. In other words, delays other than those specified in procedural assignments and specify blocks simulate in distributed delay mode. In distributed delay mode, Incisive Enterprise Simulator ignores all module path delay information and uses all distributed delays and timing checks.

delay_mode_unit

This option causes the design to simulate in unit delay mode. In unit delay mode, the tool ignores all module path delay information and timing checks, and converts all non-zero structural and continuous assignment delay expressions to a unit delay of one (1) simulation time unit.

delay_mode_zero

This option causes modules to simulate in zero-delay mode. Zero-delay mode is similar to unit delay mode in that all module path delay information, timing checks, and structural and continuous assignment delays are ignored.

Reasons for Selecting a Delay Mode

Replacing delay path, or distributed with global zero or unit delays, can reduce simulation time by an appreciable amount. You can use delay modes during design debugging phases, when checking design functionality is more important than timing correctness. You can also speed up simulation during debugging by selectively disabling delays in sections of the model where timing is not currently a concern. If these are major portions of a large design, the time saved can be significant.

The distributed and path delay modes allow you to develop or use modules that define both path and distributed delays, and then to choose either the path delay or the distributed delay at compile time. This feature allows you to use the same source description with all the *Veritools* and then select the appropriate delay mode when using the sources with Incisive Enterprise Simulator. You can set the delay mode for the tool by placing a compiler directive for the distributed or path mode in the module source description file, or by specifying a global delay mode at run time.

The *-default_delay_mode* option has been added to enable command-line control of delay modes at the module level and is available in release version 12.2 and later. An explicit delay mode can be applied to all the modules that do not have a delay mode specified by using one of the following command-line options:

```
-default _ delay _ mode <arg>
```

```
-default _ delay _ mode full _ path[...]=delay _ mode
```

Typical Use Model

Typically, models are shipped with compiler directives that enable a specific *`delay _ mode`* and features. For example:

```
`ifdef functional
  `delay _ mode _ distributed
    `timescale   1ps/1ps
`else
  `delay _ mode _ path
  `timescale 1ns / 1ps
`endif
```

This means the default *`delay _ mode`* is the `delay _ mode _ path`, with path delays defined within a specify block being overwritten by the values in the SDF.

Also, you can use the `functional` macro at compilation time to get a faster representation, with usually `#1` distributed delays (often on the buffer of the output path).

However, in some cases, it is important to be able to override this *`delay_mode`*, which is why the different modes are available.

Delay Mode Summary Table

Delay Modes	Zero	Unit	Distributed	Path	Default
Timing	Ignored	Ignored	Ignored	Used	Used
#delays	Set to 0	Set to 1 except null	Used	Ignored	Used

Controlling Delays in Your Model

Keyword	Specifies
-DELAY_MODE [<FULL|PATH>[...]=]<MODE>	Specify a delay_mode for all/selected
-DEFAULT_DELAY_MODE DELAY_MODE	Specify a delay_mode for all

1.2 Identifying Zero-Delay Loops

Apart from static tools, Incisive Enterprise Simulator also has a built-in feature that detects potential zero-delay gate loops and issues a warning if any are detected.

This feature can be enabled by using `-GAteloopwarn` (Verilog only) on the command line.

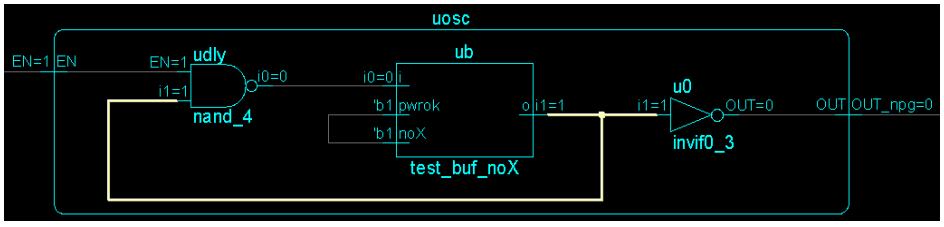
This option can help identify zero-delay gate oscillations in gate-level designs. The option sets a counter limit on continuous zero-delay loops. When the limit is reached, the simulation stops and a warning is generated stating that a possible zero-delay gate oscillation was detected. You can also use a TCL command at the *ncsim* prompt to detect the loop. The simulation will stop after the number of delta cycles specified hits a specified number.

```
ncsim>stop -delta <number> -timestep -delbreak 1
```

Once the loop is detected, you can then use the Tcl `drivers -active` command to identify the active signals and trace these signals to the zero-delay loop.

The detected loop can be fixed by adding delays to the gates that are involved in the loop.

For example, the image below shows a zero-delay loop in the design. Once the loop is detected using Tcl stop command or `-gateloopwarn`, you can add delay either on nand_4 (instance udly) or `test_buf_noX` (ub instance) to fix the loop issue.



1.3 Handling Zero-Delay Race Conditions

Races in zero-delay mode can occur because the delay for all gates is zero by default. The following sections show you how these races can be fixed.

1.3.1 Updating the Design by Adding # Delays

You can use # delays in designs to correct race conditions. For example, consider the following simple design.

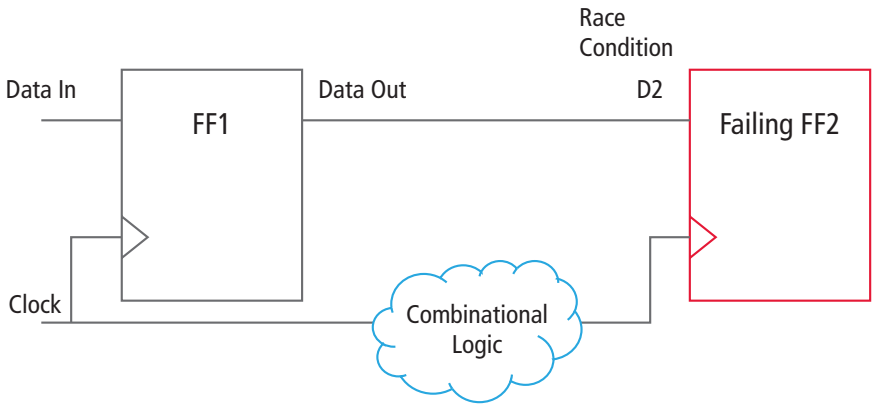


Figure 2: Design with a Race Condition

In the figure above, the simulation is running in zero-delay mode and there is some combinational logic in the clock path. In such a case, a race condition can occur and the data at `FF2` might get latched at the same clock edge.

In order to fix this race condition, you can add a unit delay `#1` at the output of `FF1`. This will delay the output of `FF1`. Consequently, the data out from `FF1` will get latched at `FF2` at the next clock edge.

The following line is an example of adding a buffer at the output of `FF1` with a unit delay:

```
buf #1 buf_prim_delay1 (D2_FF2, Data_OUT_FF1); //Add buffer with unit delay.
```

1.3.2 Using SDF Annotation for Race Conditions

For this example, consider the same design shown in Figure 2. You can add a delay in SDF for `FF1` instead of adding it directly in the design code. When the design is simulated with SDF, the delays will remove the race conditions.

1.3.3 Sequential Circuits Have Unit Delays

For this case, consider the design in Figure 2. You can add unit delays by default for all sequential cells in the design library. All the sequential cells will then have a unit delay, while combinational cells will have zero delay. For example, the following listing shows adding a buffer at the output in the flip-flop:

```
module dff (clock ,din, q);

.......... // Original Flip-Flop logic

 buf #1 buf_prim_delay1 (q, q_internal); // Add buffer with unit delay.

endmodule
```

1.3.4 Incisive Enterprise Simulator Race Condition Correcting Features

The tool has built-in features that can help designers fix race conditions in the design. The following sections present these features.

1. The SEQ_UDP_DELAY delay_specification Argument

This feature applies a specified delay value to the input/output paths of all sequential UDPs in the design. The shift register example in the following figure illustrates this.

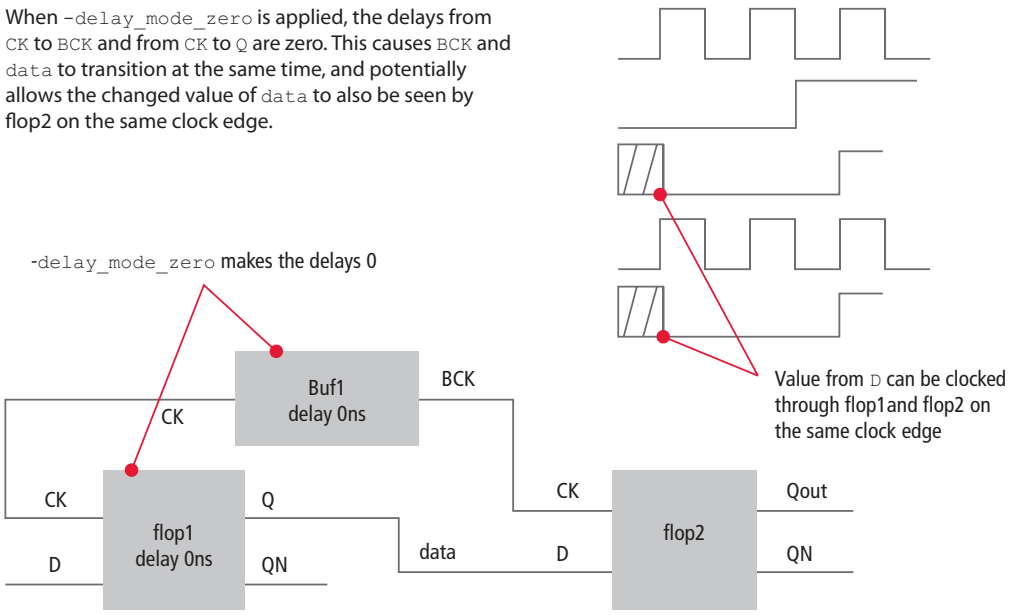


Figure 3: Applying a Delay Value to I/O Paths of Sequential UPDs

Use the `-seq_udp_delay` option to set all of the delays to zero (as if you used the `-delay_mode_zero` option) except for the sequential UDPs. The delay specified with `-seq_udp_delay` overrides any delay specified for the sequential UDPs in the design that are in specify blocks, through SDF annotation, and so on. The option also removes any timing checks associated with the sequential UDPs.

The *`delay_specification`* argument can be a real number, or a real number followed by a time unit. The time unit can be `fs`, `ps`, `ns`, or `us`. If no time unit is specified, ns is the default.

Examples

The following options assign a 10 ns delay to all sequential UDP paths.

```
-seq_udp_delay 10
```

```
-seq_udp_delay 10ns
```

The following option assigns a 0.7 ns delay to all sequential UDP paths.

```
-seq_udp_delay 0.7ns
```

The following option assigns a 5 ps delay to all sequential UDP paths.

```
-seq_udp_delay 5ps
```

The following figure illustrates the effect of using -seq_udp_delay for the shift register example.

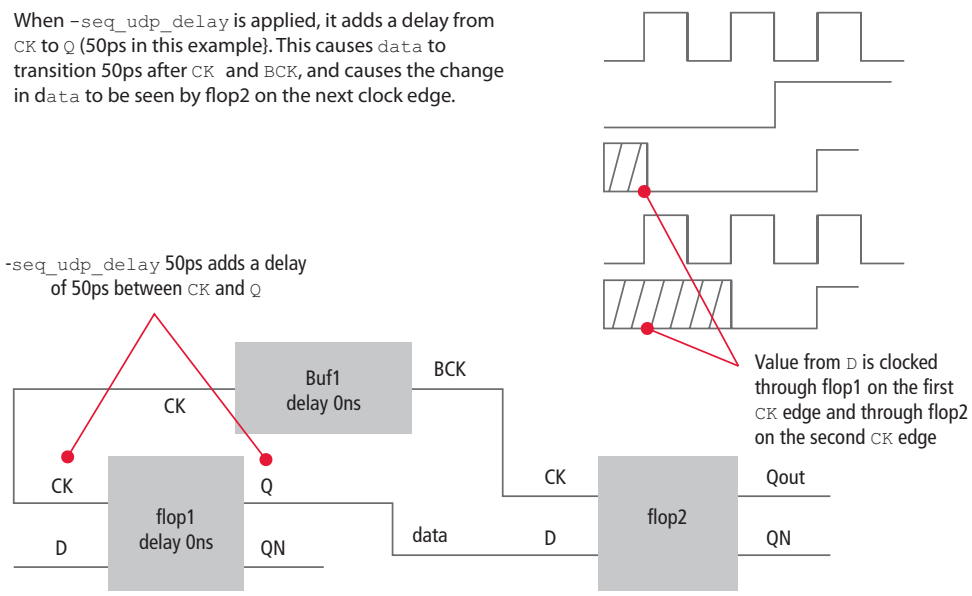


Figure 4: Using –seq_udp_delay for a Shift Register

Note that `seq _ udp _ delay` also does the following:

- Adds a delay to the sequential UDPs
- Implies `–DELAY _ MODE ZERO`
- Implies `–NOSPECIFY`

2. ADD_SEQ_DELAY delay_specification

As mentioned above, `seq _ udp _ delay` also internally implies `delay _ mode _ zero` and `nospecify` mode. In case you are not interested in this mode, `add _ seq _ delay` can be used instead. This will add a non-zero delay to the UDPs. The `add _ seq _ delay` option can also be applied for a specific instance.

For example:

```
-add _ seq _ delay top.dut _ top.u3 40ps
```

3. DELTA_SEQUDP_DELAY delay_specification

If you are not interested in adding UDP delays, `delta _ sequdp _ delay` can also be used. In most cases, just adding a delta delay to the UDP is enough to avoid the race condition, and `delta _ sequdp _ delay` also adds less overhead than a non-zero delay.

4. TIMING FILE

Unit delays can also be controlled through the new `tfile` feature, which is available in release version 12.2 and later. Using this feature, you can add unit delays in cells. This can be done at both the cell level and instance level.

To add unit delays at the cell level, the `tfile` syntax is:

```
CELLLIB L.C:V ADDUNIT //Add unit primitive delays
```

Example

```
CELLLIB worklib.sdffclrq _ f1 _ hd _ dh ADDUNIT //Adds unit delay to cell sdffclrq _ f1 _ hd _ dh
```

Similarly, to add a delay at the instance level, the `tfile` syntax is:

```
PATH <TOP>.<INST1>.<INST2> ADDUNIT //Adds unit delay to instance INST2
```

Example

```
PATH tb.top1.clock _ div1.divide0 ADDUNIT //Adds unit delay to instance divide0
```

Overriding Default Cell Delays

The timing file feature also provides a mechanism to override the default delays specified at the cell level. As mentioned in section 1.3.3 Sequential Circuits Have Unit Delays, in case `#0` and `#1` delays have already been added in the cell library, and you can override these values using the `ALLUNIT`, `UNIT`, or `ZERO` keywords. This override works only with the `` `delay_mode_distributed `` compilation directive. The following lines show this.

```
CELLIB L.C:V ALLUNIT //all non-zero and zero primitive delays will be changed to unit delay

CELLLIB L.C:V UNIT //all non-zero primitive delays will be changed to unit delay and zero
delay (#0) will remain as it is

CELLLIB L.C:V ZERO //all non-zero primitive delays will be changed to zero delay
```

Make Zero Delay

```
CELLLIB worklib.sdffclrq_f1_hd_dh zero
```

This feature also works at an instance level, as follows:

```
PATH <TOP>.<INST1>.<INST2> zero //Adds zero delay to instance INST2
```

Similarly, you can use the unit or `allunit` keywords.

2. Improving the Performance of Gate-Level Simulation with Timing

2.1 Timing Checks

Timing checks have a significant impact on performance, so it is recommended that you disable timing checks if they are not required. Timing checks can also be enabled partially, only for required blocks, by passing a timing file to Incisive Enterprise Simulator. The following sections present features that can help you control timing checks.

2.1.1 Use of the –NOTImingchecks Option

The `NOTImingchecks` option prevents the execution of timing checks. This option turns off both Verilog and accelerated VITAL timing checks.

Please Note: The `-notimingchecks` option turns off all timing checks. Because the timing checks have been turned off, any calculation of delays that would normally occur because of negative limits specified in the timing checks is disabled. If your design requires that these delays be calculated in order for the design to simulate correctly, use the `-ntcnotchks` option mentioned below instead.

2.1.2 Controlling Timing Through a `tfile`

You can turn off timing in specific parts of a Verilog design by using a timing file, which you specify on the command line with the `-tfile` option. The timing file can help you disable timing for selected portions of a design. For example:

```
% ncelab -tfile myfile.tfile [other_options] worklib.top:module

% irun -tfile myfile.tfile [other_options] source_files
```

If you are annotating with an SDF file, the design is annotated using the information in the SDF file, and then the timing constructs that you specify in the timing file are removed from the specified instances.

Using a timing file does not cause any new SDF warnings or remove any timing warnings that you would get without a timing file. However, there is one exception, as follows:

> *The connectivity test for register-driven interconnect delays happens much later than the normal interconnect delays. Any warning that may have existed for that form of the interconnect will not be generated if that interconnect has been removed by a timing file.*

The timing specifications you can use in the timing file are listed in the following table:

Timing Specification	Description
`-iopath`	Disable module path delays.

Timing Specification	Description
+iopath	Enable module path delays.
-prim	Disable primitive delays. Sets any primitive delay within the specified instance(s) to 0.
+prim	Enable primitive delays within the specified instance(s).
-port	Remove any port delays at the specified instance(s) or any interconnects whose destination is contained by the instance. interconnect sources are not affected by the -port construct.
+port	Enable port delays at the specified instance(s) or any interconnects whose destination is contained by the instance.
[*list _ of _* tchecks] -tcheck	Remove the listed timing checks from the instance(s).

[list _ of _ tchecks]
+tcheck
	Enable the listed timing checks in the instance(s).
	If specific timing checks are not listed, all timing checks will be disabled or enabled.
	Examples:
	// Disable all timing checks in top.foo
	PATH top.foo -tcheck
	// Disable $setup timing check in top.foo
	PATH top.foo $setup -tcheck
	// Disable $setup and $hold timing checks in top.foo
	PATH top.foo $setup $hold -tcheck

-timing +timing	This is an alias for the four specifications shown above.

Example Timing File

The following listing shows an example timing file:

```
// Disable all timing checks in top.inst1
PATH top.inst1 -tcheck
/* Disable all timing checks in all scopes below top.inst1 except
for instance top.inst1.U3 */
PATH top.inst1... -tcheck
PATH top.inst1.U3 +tcheck
/* Disable $setup timing check for all instances under top.inst2,
except for top.inst2.U1. */
PATH top.inst2... $setup -tcheck
PATH top.inst2.U1 $setup +tcheck
// Disable $setup and $recrem timing checks for instance top.inst3
PATH top.inst3 $setup $recrem -tcheck
// Disable timing checks for all objects in the library mylib
CELLLIB mylib -tcheck
// Disable module path delays for all instances in the 2nd levels of hierarchy
PATH *.* -iopath
```

2.1.3 Use of the –NTCNOtchks *Option*

This option generates negative timing check (NTC) delays, but does not execute timing checks. The option is available for Verilog designs only.

You can use the –notimingchecks option to turn off all timing checks in your design. However, if you have negative timing checks in the design, this option also disables the generation of delayed internal signals, and you may get the wrong simulation results if the design requires these delayed signals to function correctly. That is, if you have negative timing checks, simulation results may be different when using –notimingchecks and without –notimingchecks.

Use the –ntcnotchks option instead of the –notimingchecks option if you want the delayed signals to be generated but want to turn off timing checks. This option removes the timing checks from the simulation after the negative timing check (NTC) delays have been generated.

Controlling Timing Checks with the -NTCNOtchks *Option Using a* tfile

By default, all timing checks are disabled by the –ntcnotchks option. However, in case you are interested in the timing violations for some portion of the design (typically the top block), the timing checks can be controlled using the tfile timing file.

Please Note: This feature is available from release version 12.2 onwards.

The keyword ntcnotchks is available for use in the timing file. The keyword controls the –ntcnotchks option at the instance level. Using the keyword enables the timing checks, and a delayed net will be generated for those instances using the –ntcnotchks option.

Note: The – or + prefix is not required for the ntcnotchks option in the tfile. If ntcnotchks is present in the file, it is always enabled.

Some example cases are presented here.

Case 1

Enable negative timing computation delays for all instances, with timing checks enabled for a particular instance only.

Timing File Sample

```
//Enables Negative timing computation delays on the entire design but turns off timing
checks on all instances.

DEFAULT ntcnotchks

//Enables timing checks for a particular instance.
path top.u1 +tcheck
```

Case 2

Selectively enable negative timing check delays and timing checks.

Timing File Sample

```
//Disables timing checks & negative timing computation delays on the entire design
DEFAULT –tcheck

//Enables timing check & ntcnotchks for a particular instance.
path top.u1.u2 +tcheck

//Enables negative timing computations for a particular instance and disables tchecks
path top.u1.u3 ntcnotchks
path top.u1.u4 ntcnotchks
```

2.1.3 Controlling the Number of Timing Check Violations

The results of GLS are often not usable if timing violations are reported. Consequently, many designers do not want to continue the simulation further if a timing violation is reported.

When timing violations are reported, you can cause the simulation to immediately exit by using the following switch:

```
–max _ tchk _ errors <number>
```

When this option is specified, the value of the number argument specifies the maximum number of allowable timing check violations before the simulation exits.

Please Note: This feature is available in release 12.2 and onwards.

2.1.4 Disabling Specific Timing Checks During Simulation

The `tcheck` command turns timing check messages and notifier updates on or off for a specified Verilog instance. The specified Verilog instance can also be an instance of a Verilog module instantiated in VHDL.

Command syntax:

```
tcheck instance_path -off | -on
```

2.1.5 Table - Disabling Timing Information

Keyword	Specifies
`-NOSPecify`	Disable timing information described in specify blocks and SDF annotation
`-NOTImingchecks`	Do not execute timing checks.
`-NONOtifier`	Ignore notifiers in timing checks
`-NO_TCHK_msg`	Suppress the display of timing check messages, while allowing delays to be calculated from the negative limits
`-NTCNOtchks`	Generate negative timing check (NTC) delays, but do not execute timing checks (in cases where you want the delayed signals to be generated but want to turn off timing checks).

2.2 Removing Zero-Delay MIPDs, MITDs, and SITDs

You can improve performance by removing interconnect or port delays that have a value of 0 from the SDF file. The SDF Annotator parses and interprets zero-delay timing information but does not annotate it. By removing the zero-delay information from the SDF file, you eliminate unnecessary processing of this information.

Note: This performance improvement recommendation applies only to MIPDs, MITDs, and SITDs.

3. Improving Simulation Performance Using Incisive Enterprise Simulator Multi-Core Solution

The benefit of multiple CPU cores can be obtained in two ways. One way is to run separate simulations on all available CPU cores. This is useful when running many tests in parallel, as in a regression environment. However, running tests in parallel does not reduce the total simulation time for individual designs. To do this, different parts of the design must be run on different cores in parallel.

If you are concerned about how to reduce a long-running simulation, you could use the Incisive Enterprise Simulator multi-core capability, available in release 13.1 and onwards, to run the design under test (DUT) in parallel. This approach is called design-level parallelism (DLP). DLP is implemented when multiple threads are run on different cores to achieve parity.

To specify the parts of the DUT that are to run in parallel on separate threads, you write a **partition file**. This file specifies which portions of the hierarchy belong to each partition. The partition file is included at elaboration time using the **-mcfile** option (**ncelab -mcfile** or **irun -mcfile**). This is the only option required for running in multi-threaded mode. The following section explains the flow of creating a partition file by using different sources of information.

Multi-Core Use Model

The multi-core solution works well for scenarios where a simulation has enough parallel activities. In general, an automatic test pattern generation (ATPG) pattern would activate almost all the logic in the DUT at every clock cycle, which makes GLS DFT simulation (ATPG pattern) a good candidate for the Incisive Enterprise Simulator multi-core solution.

The table below shows one of the designs with timing having good performance improvement.

Machine/SW	Intel E5-4650 0@2.70GHz, 20M/IES13.1s09					
	Ncvlog	**ncelab**	**ncsim**	**SimTime Gain**	**Total Time**	**Overall Time Gain**
Single Core	30.5s	1278.4s	2816.8s	1x	4125.7	1X
Memory usage	328.5MB	8492.7MB	7347MB			
2 Cores	30.6	1349.7	1702.6	1.65X	3082.9	1.34X
Memory usage	328.5MB	8493.1MB	7509.2MB			
4 Cores	30.5	1340.7	1318.4	2.13X	2689.6	1.53X

STIL File

A STIL file is a standard way to describe the scan data. Some of the ATPG tools, such as Cadence Encounter® Test, could generate the STIL with scan-chain information. The scan-chain information would list the scan flip-flops on each scan chain. Partial STIL fill contents are shown below. You can extract the scan cells information to find out the flip-flop numbers in each instance.

STIL.FULLSCAN.full_all.signals.stil

```
ScanStructures {

   ScanChain "Control_Observe_Reg_1_FULLSCAN" {
     ScanLength 741;
     ScanIn "port_pad_data_in[0]";
     ScanCells
       "COMPACTOR.compressor.mreg2_reg_0_.__i0.dff_primitive"
       "COMPACTOR.compressor.mreg2_reg_1_.__i0.dff_primitive"
       "COMPACTOR.compressor.mreg2_reg_2_.__i0.dff_primitive"
       "COMPACTOR.compressor.mreg2_reg_3_.__i0.dff_primitive"
       "COMPACTOR.compressor.mreg2_reg_4_.__i0.dff_primitive"
       "COMPACTOR.compressor.mreg2_reg_5_.__i0.dff_primitive"
       …
    ScanOut "port_pad_data_out[0]";
    ScanInversion 0; }

   ScanChain "Control_Observe_Reg_2_FULLSCAN" {
     ScanLength 740;
     ScanIn "port_pad_data_in[1]";
     ScanCells
       "COMPACTOR.compressor.mreg1_reg_0_.__i0.dff_primitive"
       "COMPACTOR.compressor.mreg1_reg_1_.__i0.dff_primitive"
       "COMPACTOR.compressor.mreg1_reg_2_.__i0.dff_primitive"
       "COMPACTOR.compressor.mreg1_reg_3_.__i0.dff_primitive"
            …
              "DTMF_REG1_digit_out_reg_14_.__i2.dff_primitive"
              "DTMF_REG1_digit_out_reg_15_.__i2.dff_primitive" ;
              ScanOut "port_pad_data_in[3]";
              ScanInversion 0; }
}
```

The Limitation of Using STIL

Not every tool provides the capability to write out the scan chain information.

Parallel Scan Testbench

In order to reduce the runtime of scan patterns, the ATPG tools usually generate parallel scan patterns which will directly load the scan data to scan flip-flops by using Verilog force statement. It is expected to have the scan flip-flop information in the Verilog testbench. It is possible to extract the flip-flop numbers in each instance from the testbench.

VER.FULLSCAN.full_all.mainsim.v

```
assign ( supply0, supply1 ) // CR = 1

   dtmf_chip_inst.port_pad_data_in[0] = part_SLs_1[0741]!==1'bZ ?  part_SLs_1[0741] : 1'bZ
,   // pinName = port_pad_data_in[0]:    CR = 1:  pos = 1:  index = 21:  nodeID = 22:  latchEntry
= 34:  :

   dtmf_chip_inst.COMPACTOR.compressor.mreg2_reg_1_.D = part_SLs_1[0740]!==1'bZ ?
part_SLs_1[0740] : 1'bZ ,   // pinName = COMPACTOR.compressor.mreg2_reg_1_.D:    CR = 1:
pos = 2:  index = 2061:  nodeID = 376:  latchEntry = 36:  :

   dtmf_chip_inst.COMPACTOR.compressor.mreg2_reg_2_.D = part_SLs_1[0739]!==1'bZ ?
part_SLs_1[0739] : 1'bZ ,   // pinName = COMPACTOR.compressor.mreg2_reg_2_.D:    CR = 1:
pos = 3:  index = 2120:  nodeID = 384:  latchEntry = 38:  :

   dtmf_chip_inst.COMPACTOR.compressor.mreg2_reg_3_.D = part_SLs_1[0738]!==1'bZ ?
part_SLs_1[0738] : 1'bZ ,   // pinName = COMPACTOR.compressor.mreg2_reg_3_.D:    CR = 1:
pos = 4:  index = 2179:  nodeID = 392:  latchEntry = 40:  :

   dtmf_chip_inst.COMPACTOR.compressor.mreg2_reg_4_.D = part_SLs_1[0737]!==1'bZ ?
part_SLs_1[0737] : 1'bZ ,   // pinName = COMPACTOR.compressor.mreg2_reg_4_.D:    CR = 1:
pos = 5:  index = 2238:  nodeID = 400:  latchEntry = 42:  :

   dtmf_chip_inst.COMPACTOR.compressor.mreg2_reg_5_.D = part_SLs_1[0736]!==1'bZ ?
part_SLs_1[0736] : 1'bZ ,   // pinName = COMPACTOR.compressor.mreg2_reg_5_.D:    CR = 1:
pos = 6:  index = 2297:  nodeID = 408:  latchEntry = 44:  :

   dtmf_chip_inst.COMPACTOR.compressor.mreg2_reg_6_.D = part_SLs_1[0735]!==1'bZ ?
part_SLs_1[0735] : 1'bZ ,   // pinName = COMPACTOR.compressor.mreg2_reg_6_.D:    CR = 1:
pos = 7:  index = 2356:  nodeID = 416:  latchEntry = 46:  :

   dtmf_chip_inst.COMPACTOR.compressor.mreg2_reg_7_.D = part_SLs_1[0734]!==1'bZ ?
part_SLs_1[0734] : 1'bZ ,   // pinName = COMPACTOR.compressor.mreg2_reg_7_.D:    CR = 1:
pos = 8:  index = 2415:  nodeID = 424:  latchEntry = 48:  :

   …..;
```

Limitation

Different ATPG tools have different testbench coding styles. You need to understand the format of these testbenches in order to extract the information. For details on the multi-core solution, please refer to the Multi-Core Simulation in the Incisive Enterprise Simulator document in Cadence Help.

4. Improving Performance in Debugging Mode

4.1 Wave Dumping

Waveform dumping negatively impacts simulation performance, so it should be used only if required. If the waveform dumping activity is high, you can use the parallel waveform `mcdump` feature of *ncsim*. Waveform dumps inside cells/primitives should be turned off (using probe … *-to_cells*).

Enabling Multi-Process Waveform Dumping

If you are running the simulator on a machine with multiple CPUs, you can improve the performance of waveform dumping, and decrease peak memory consumption, by using the `–mcdump` option (for example, `ncsim –mcdump` or `irun –mcdump`). In multi-process mode, the simulator forks off a separate executable called *ncdump*, which performs some of the processing for waveform dumping. The ncdump process runs in parallel on a separate CPU until the main process exits. You can see a 5% to 2X performance improvement depending upon the number of objects probed.

4.2 Access

By default, the simulator runs in a fast mode with minimal debugging capability. To access design objects and lines of code during simulation, you must use *ncelab* command-line debug options. These options provide visibility into different parts of a design, but disable several optimizations performed inside the simulator.

For example, Incisive Enterprise Simulator includes an optimization that prevents code that does not contribute in any way to the output of the model from running. Because this dead code does not run, any runtime errors, such as constraint errors or null access de-references, that would be generated by the code are not generated. Other simulation differences (for example, with delta cycle-counts and active time points) can also occur. This dead code optimization is turned off if you use the `ncvlog –linedebug` option, or if read access is turned on with the `ncelab –access +r` option.

The following line shows the `ncelab –access` debug command and command-line options that provide additional information and access to objects, but reduce simulator speed. Because these options slow down performance, you should try to apply them selectively, rather than globally.

```
ncelab –access +[r][w][c]
```

`r` provides read access to objects

`w` gives write access to objects

`c` enables access to connectivity (load and driver) information

To apply global access to the design, the following command can be used for read, write, and connectivity access:

```
% ncelab –access +rwc options top _ level _ module
```

Try to specify only the type(s) of access required for your debugging purposes. In general:

- Read (`r`) access is required for waveform dumping and for code coverage.
- Write (`w`) access is required for modifying values through PLI or Tcl.
- Connectivity (`c`) access is required for querying drivers and loads in C or Tcl, and for features like tracing signals in the SimVision *Trace Signals* sidebar. In regression mode, connectivity access must be always turned off.

For example, if the only reason you need access is to save waveform data, use `ncelab –access +r`.

To maximize performance, consider using an access file with the `ncelab –afile` option instead of specifying global access using the `–access` option. Any performance improvement is proportional to the amount of access that is reduced. The maximum improvement from specifying `–access rwc` to using no access can be >2X. Even removing connectivity (c) access can result in a 20-30% improvement. The typical gain of fine-tuning access in large environments is 15-40%

You can also write an *afile* to increase performance at the gate level.

Sample *afile*:

```
DEFAULT +rw

CELLINST –rwc
```

Note: A new switch `–nocellaccess` is also available in release 12.1 and later to turn off cell access.

Multiple Snapshots for Access

You can also maintain two snapshots—one with more or global access for debug mode, and the other with limited or no access, for regressions. For performance, you can always use the snapshot with limited or no access and, in the case of debugging, use the global access snapshot.

5. Other Useful Incisive Enterprise Simulator Gate-Level Simulation Features

5.1 Initialization of Variables

The `–NCInitialize` option provides you with a convenient way to initialize Verilog variables in the design when you invoke the simulator, instead of writing code in an `initial` block, using Tcl `deposit` commands at time zero, or writing a VPI application to do the initialization. This option works for Verilog designs only, and you can enable the initialization of all Verilog variables to a specified value. To initialize Verilog variables, you use `–ncinitialize` with both the `ncelab` and the `ncsim` commands as follows:

- Use `ncelab –ncinitialize` to enable the initialization of Verilog variables.
- Use `ncsim –ncinitialize` to set the value to which the variables are to be initialized when you invoke the simulator.

For example:

```
% ncvlog test.v

% ncelab –ncinitialize worklib.top // Enable initialization

% ncsim –ncinitialize 0 worklib top // Initialize all variables to 0
```

When you invoke the simulator, all variables can be initialized to `0`, `1`, `x`, or `z`. For example, the following line initializes all variables to 0:

```
ncsim –ncinitialize 0
```

Different variables can be initialized to `0`, `1`, `x`, or `z` randomly using `rand:`*`n`*, where n is a 32-bit integer used as a randomization seed. For example:

```
ncsim –ncinitialize rand:56
```

Different variables can be initialized to `0` or `1` randomly using `rand _ 2state:`*`n`*. For example:

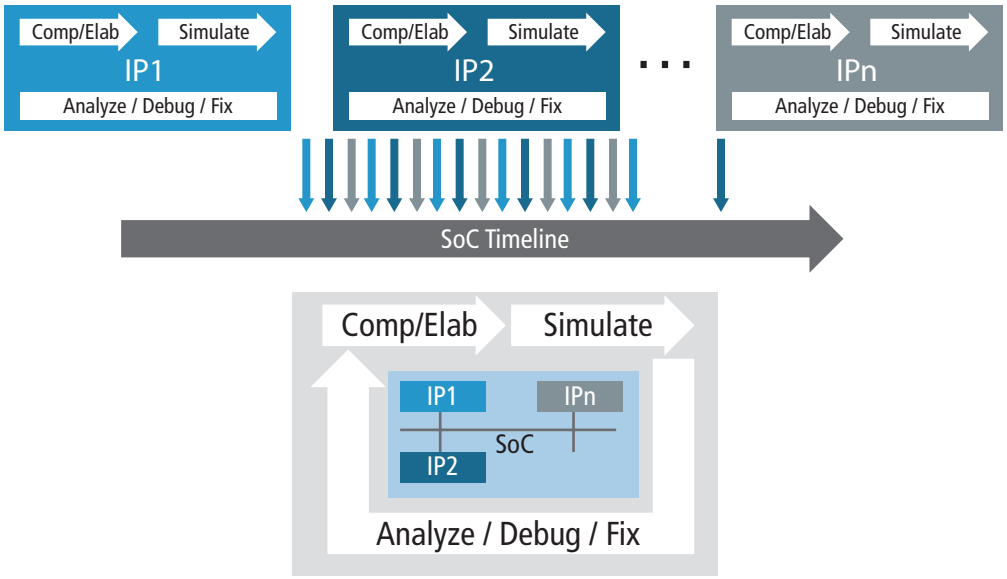
```
ncsim –ncinitialize rand _ 2state:56

See ncsim –ncinitialize in Cadence Help for details.
```

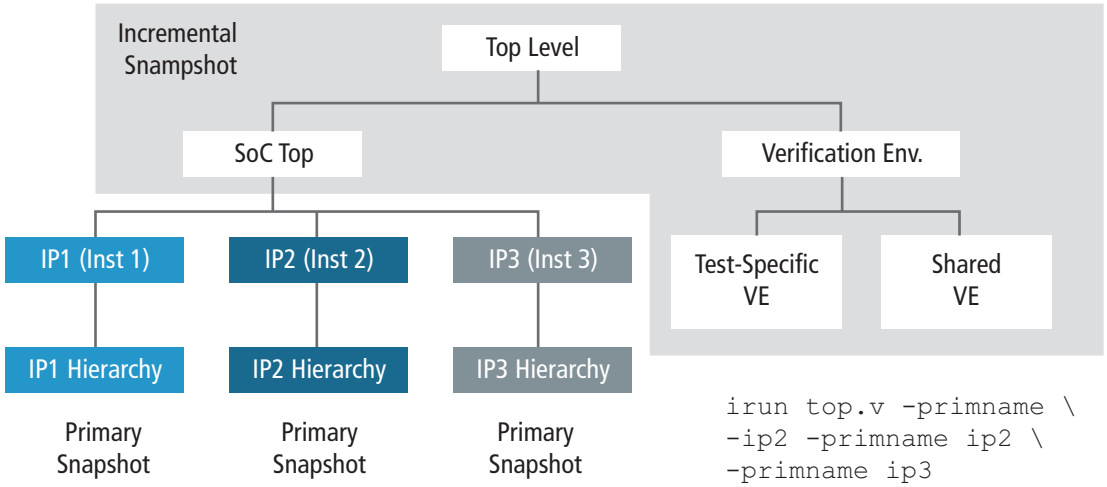
Note: You can also enable initialization by specifying global write access to all simulation objects with the `–access +w` option. However, this option provides both read and write access to all simulation objects in the design, which can negatively affect performance. The `–ncinitialize` option provides read and write access only to Verilog variables.

6. Improving Elaboration Performance Using Multi-Snapshot Incremental Elaboration (MSIE)

Incremental elaboration technology can be used in any case where turn-around time (time for newer versions to simulate) is an overriding concern. In an SoC environment, there can be multiple IP used and newer revisions might come in at different intervals in the verification process. MSIE technology can help the designer save build time and start gate-level simulations sooner.



MSIE can be applied as shown in the figure below, where there are multiple IP cores in an SoC top module. The different primary snapshots can be created for each IP and top, and the testbench environment can be in the incremental portion. Even in the case where a complete elaboration is required at once, all the primaries can be built in parallel and, finally, the incremental portion can be built, including all primaries. This saves a lot of elaboration time.



MSIE Use Model

1. Apply to GLS without timing (without SDF)—The technology can be directly applied and design partitioning can be done as shown in the figure above.

2. Apply to GLS with timing (SDF)—Requires a separate SDF for each partition and IP.

MSIE can also be applied for DFT-style tests (serial or parallel scan). If the tests are set up so that each test has a separate top-level HDL file containing the patterns (which is somewhat common), then building the DUT as a primary snapshot with the SDF file, and then making the incremental snapshot the *test*+DUT, works very well.

7. Accelerating Gate Level Simulations

The Cadence Palladium® XP platform accepts design descriptions in both synthesizable RTL and gate-level representations or mixed abstractions equally well. Gate-level designs are compiled using the UXE compiler targeting all verification modes, including simulation acceleration and in-circuit emulation with minimal to no compromise on capacity or speed. Generally speaking, as design sizes increase (or the number of objects to simulate increase), the performance of a simulator subsides whilst the performance of the Palladium XP platform remains fairly constant thereby achieving relatively higher verification speeds. For simulation acceleration, speed-ups of 100x or higher are typical, whereas for in-circuit emulation, speed-ups of 10,000x or higher are achievable, reaching MHz speeds. For large designs where the simulation speeds for gate-level designs are in the range of 1 to 10Hz, simulation acceleration can be as high as 10,000x, and in-circuit emulation could be as high as 100,000x over the GLS.

With the Palladium XP platform, gate level designs can be used for both pre-silicon verification and post-silicon validation. Users can validate gate-level netlists synthesized by standard synthesis tools targeting silicon vendor libraries, including DFT and scan overlay structures. Gate level netlists can also be used to identify and analyze peak and average power at sub-system and system level.

A Methodology for Improving Gate-Level Simulation

This section shows some methodologies that can help you reduce overall GLS verification time and make the process more effective.

1. Effectively Use Static Tools Before Starting Gate-Level Simulation

Using static tools like linting and static timing analysis (STA) tools can effectively reduce the gate-level verification time.

1.1 Linting Tool

It is recommended that you use static *linting* tools such as the Cadence HDL Analysis and Lint (HAL) tool before starting zero-delay simulations. This can help you identify the issues or potential areas that can lead to unnecessary issues in gate-level simulation.

Since simulation runs at gate level can take a lot of time, and based on the issues reported by the linting tools, updates can be done in the gate-level environment up front, rather than waiting for long gate-level simulations to complete and then fixing the issues. Some of the potential issues that can be identified by linting tools are as follows:

- Detecting zero-delay loops
- Identifying possible design areas that can lead to race conditions

If race issues are detected, you can use the techniques listed in section 1.3 Handling Zero-Delay Race Conditions to fix them.

1.2 Static Timing Analysis Tool

Static timing analysis tool information and reports can be used to start gate-level simulations with timing early in the design cycle. The information from STA reports can help you run meaningful gate-level simulations with timing, by focusing on the design areas that have met the timing requirements. Also, you should temporarily fix timing for other portions of the design that require changes or updates for timing closure.

Some techniques that can be used for this are presented in this section.

1.2.1 Reducing Gate-Level Timing Simulation Errors and Debug Effort Based on STA Tool Reports, Even If Timing Closure Is Incomplete for Some Parts of the Design

Since GLS runs much slower than simulations without timing, starting GLS timing verification early can be helpful. However, this can lead to a lot of unnecessary effort in debugging the issues that are already reported by STA tools like Cadence Encounter Timing System, as the design is still in the process of timing closure. As the timing issues that have been reported by STA tools require fixes to the design, using the SDF from this phase directly in the simulation will show failures in the simulation as well.

The designer can temporarily fix these violations in SDF through STA tools and start GLS in parallel. And at the same time, timing issues in the design can be fixed by the timing closure team. This technique can help you:

- Reduce GLS errors based on STA (Encounter Timing System reports)
- Improve the design cycle/debug time during GLS:
  - You can focus on gate-level issues rather than known STA (Encounter Timing System) errors
  - Debugging and fixing issues in parallel during design phase:
    - » STA (Encounter Timing System) timing violations
    - » GLS (other than Encounter Timing System errors)
  - Running functionally correct GLS by ignoring Encounter Timing System errors

The following section gives an overview of the typical STA GLS flow and also describes techniques to temporarily fix setup and hold violations.

Static Timing Analysis and Gate-Level Simulation Flow

The following figure illustrates an overview of a typical STA tool, such as Cadence Encounter Timing System and a gate-level simulator, such as Incisive Enterprise Simulator.

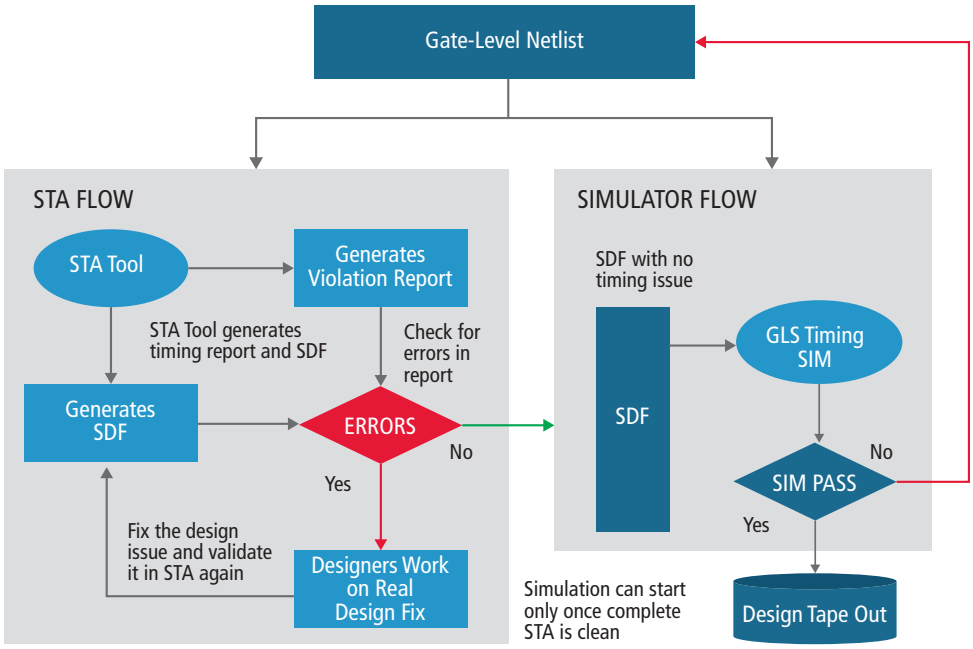
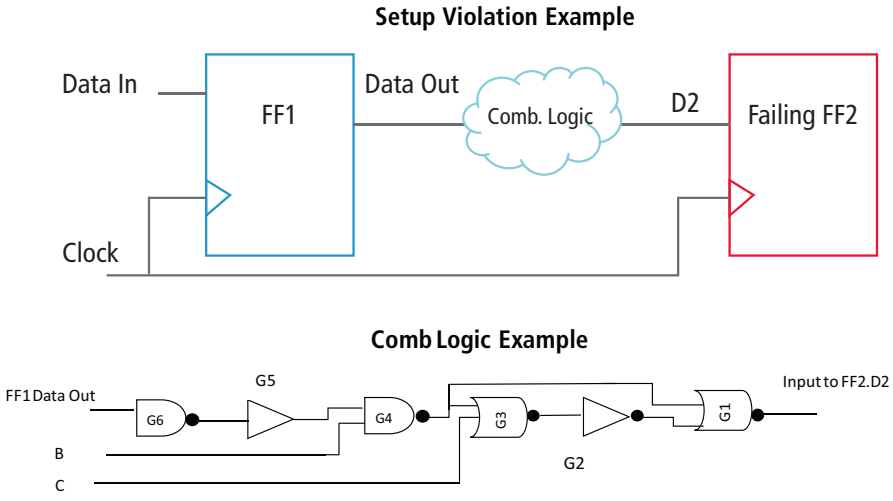


Figure 5: Typical Static Timing Analysis Tool and Gate-Level Simulator

Handling STA Setup Violations

This technique can help designers temporarily fix setup timing violations reported by STA tools so that they do not appear in the GLS. Consider the following simple example for a setup violation:



End Point	Slack (ns)	Cause
Top/core/ctrl/FF2/D2	-0.56	Violated

Figure 6: Setup Violation

In the example illustrated above, a setup violation is reported at `FF2` and the slack is -0.56. The sample combinational logic is also shown in the figure. If the SDF is generated by an STA tool (in this case, Encounter Timing System) directly, then it will not only report timing violations during GLS, but there will also potentially be a functionality mismatch, as the data for D2 might not get latched at `FF2` at the next clock cycle. This might impact the functionality of other parts of the design as well.

Since this is a known issue and requires a fix in the combinational logic from `FF1` to `FF2`, the GLS can temporarily ignore this path, or the timing needs to be temporarily fixed. This can be done in the STA tool itself by setting a few gate delays to zero and compensating for the slack. Once the slack is compensated for, then the SDF can be generated and used by the GLS.

To help you understand this approach in detail, the following table shows the delays for each gate.

Gate	Delay (ns)	Cumulative Delay (ns)
G1	0.32	0.32
G2	0.25	0.57
G3	0.4	0.97
G4	0.35	1.32
G5	0.28	1.60
G6	0.33	1.99

Referring to Figure 6, within Encounter Timing System, starting from node FF2.D2, a traversal is done and the slack at each node is checked. Looking at the table above, it can be seen that setting delays for G1 and G2 to zero will compensate the slack of -0.56.

Similarly, all the setup violations can be temporarily fixed. STA can be run again on the modified timing to check if there are any new violations added by assigning zero delays to the gates.

This approach can be helpful only when the violations reported by STA tools are limited in number and are present only in some small part of the design. Using this approach will not work in cases where there are a lot of violations, and where they affect almost the complete design, as it would change the timing information in the SDF file completely and make the simulation very optimistic.

Adding Negative Delays in SDF (Limited Cases)

The effects of the above case can also be mitigated by adding negative delays in the STA environment at gate G1, if the negative delay value in the above example is less than or equal to the G1 delay (which is 0.32ns). This feature of adjusting negative delays by the simulator is available only in the Incisive 13.2 release and onwards. This new feature in Incisive Enterprise Simulator can only adjust delays for one level. If the delays cannot be adjusted for more than level 1, the remaining negative values are set to zero. By default, the elaborator zeros out all negative delays and issues a warning. To enable negative interconnect and module path delay adjustments, run the elaborator with the `-negdelay` option (`ncelab -negdelay` or `irun -negdelay`). For example:

```
% ncvlog -messages test.v

% ncelab -messages tb -negdelay
```

or

```
% irun test.v -negdelay
```

Use the `-neg _ verbose` option (`ncelab -neg _ verbose` or `irun -neg _ verbose`) to print out the negative delay adjustments and save them to the default log file, as shown:

```
% irun test.v -negdelay -neg _ verbose
...
file: test.v
...

       Elaborating the design hierarchy:
       Top level design units:
               tb
       ...
       Building instance-specific data structures.
       Processing Negative Interconnect Delays...

       Adjusting driver of net tb.out1...
       Driver delay adjusted successfully...
       Adjusting driver of net tb.out3...
       Driver delay adjusted successfully...

       Number of negative interconnect delays adjusted successfully: 2
       2 Negative interconnect delays have been adjusted successfully.
       ...
       Writing initial simulation snapshot: worklib.tb:v
```

Adjustment Rules for Negative Delays

Incisive Enterprise Simulator now supports negative delays on single output gates without the need to zero their values. When enabling negative delay adjustments, the adjustments are applied in the following sequence:

1. Other interconnect delays at the port

2. Driver delays at the input

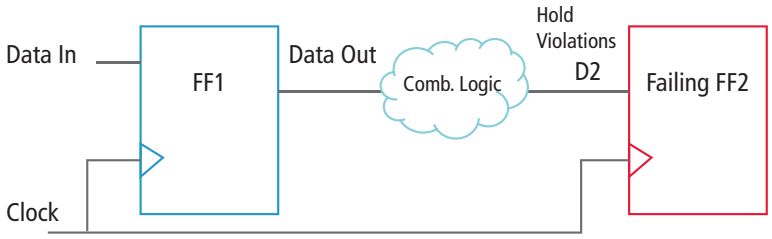
3. Load delays at the output

Note: When comparing the negative delay values to the total delays in a design, adjustments to negative delay values will introduce inconsistencies with the delays specified in the SDF file and may affect elaborator performance.

Please Note: For this approach Cadence is evaluating different designs to test corner scenarios and would like to partner with design teams to review and deploy it. Contact your field AE if you are interested.

Handling STA Hold Violations

Similar to set-up violations, for hold violations consider the example below:



End Point	Slack (ns)	Cause
Top/core/ctrl/FF2/D2	-0.50	Violated

Figure 7: Hold Violation

A hold violation is reported at `FF2` and the slack is -0.50. In order to temporarily fix the issue, an extra interconnect delay of 0.5ns can be added at D2.

The following are the advantages, assumptions, and limitations for this flow.

Advantages

- Encounter Timing System violations will not appear in the GLS
- Improvements in design cycle time, as the designer can focus on GLS issues, other than issues reported by Encounter Timing System
- Adding 0 delays for gates can improve GLS performance

Assumptions

- Most of the design is STA clean and has limited known violations
- Adding 0 delays will not add other STA timing errors
- Needs to be checked by re-running STA

Limitations

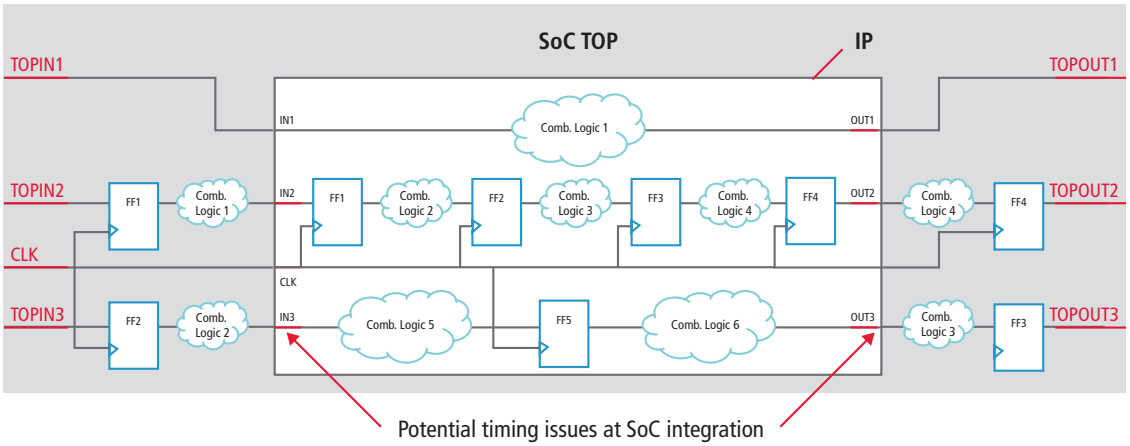
- New errors added by the above updates might not get corrected.

Generating and Using Smart SDF with Timing Abstractions Running Timing Simulations

STA tools have a capability for doing hierarchical timing analysis and generating SDF hierarchically. This feature is included in STA tools because flat STA for a complete chip takes a lot of time.

Typically, there are IP cores in, or in portions of, the design that are re-used across different SoCs and are already silicon-proven. Turning on complete timing for an SoC enables timing of internal cells of all the blocks and IP cores, leading to a lot of redundant simulation overhead. Currently, the designer can turn off the timing of the complete IP/module using the tool features, but this adds optimism at the SoC level since the delay at output ports of the IP become zero. Also, the timing checks at the input pins of the IP are ignored.

IP that is timing-clean and integrated at the SoC level, if run with full timing enabled, requires timing for all flops and combination logic for accurate timing results. Since the IP is already timing-clean, the only set of timing issues that can come are at the SoC level, or at the integration of the IP as shown in the figure below.



The following types of timing models can be used to generate SDF for gate-level simulation: Interface logic models

- Extracted timing models

Please Note: For this approach Cadence is evaluating different designs to test corner scenarios and would like to partner with design teams to review and deploy it. Contact your field AE if you are interested.

2. Interface Logic Models

An interface logic timing model (ILM) has partial timing of the block that includes the timing at boundary logic only, but hides most of the internal register-to-register logic.

Consider the following simple example.

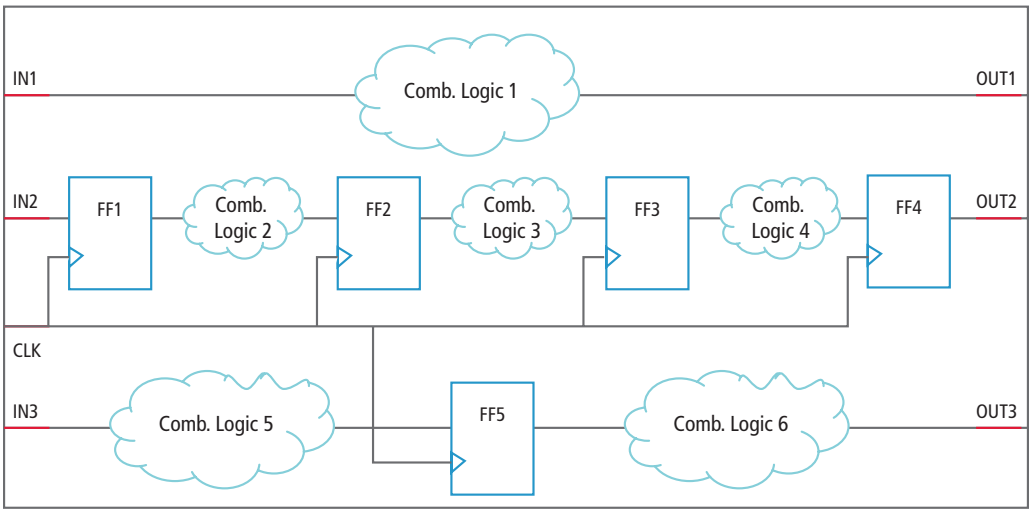


Figure 8: Full Timing Model Example Design

Flop-to-flop internal paths are not required, as they do not affect the interface path timing and we are interested in the interface path timing only. In the figure below, the timing of internal gates is set to zero (marked in green)

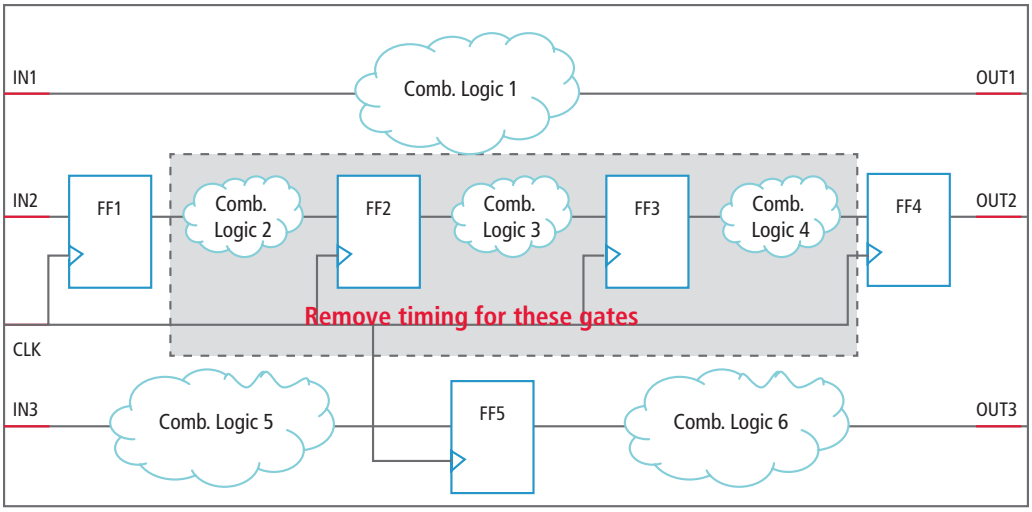


Figure 9: Interface Logic Model

2.1 Advantages of Using Interface Logic Models

The advantages of using interface logic models include:

- Good performance improvements, both in STA and GLS, since only a partial design is used
- High accuracy as interface logic and interface cells are preserved

Please Note: This is a patented flow (beta version) and is available in Encounter Timing System (ETS) to dump abstracted interface timing models for running effective GLS. Cadence would like to partner with design teams to review and deploy this flow. Contact your field AE if you are interested.

The figure below shows the results for an IP core with full timing versus interface logic timing. As the waveforms show, both behave exactly the same. The performance gains are expected to be in the range of 1.5X – 2X, depending upon the number of IP cores considered for timing abstraction or the overall level of abstraction.

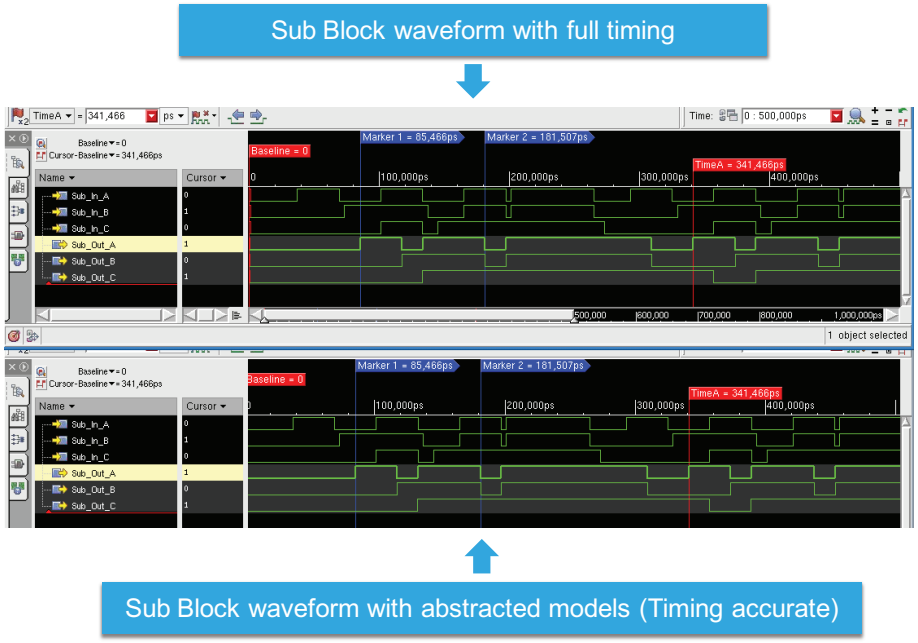


Figure 10: Functionality and Timing of an IP Core with Full Timing and with Interface Logic Timing

2.2 Extracted Timing Models

An extracted timing model (ETM) creates a timing representation for the interface paths, which are the timing arcs created for each input-to-flop/gate, input-to-output, and flop/clock-to-output paths.

If there are multiple clocks capturing data from an input port, then an arc, with respect to each input port, is extracted.

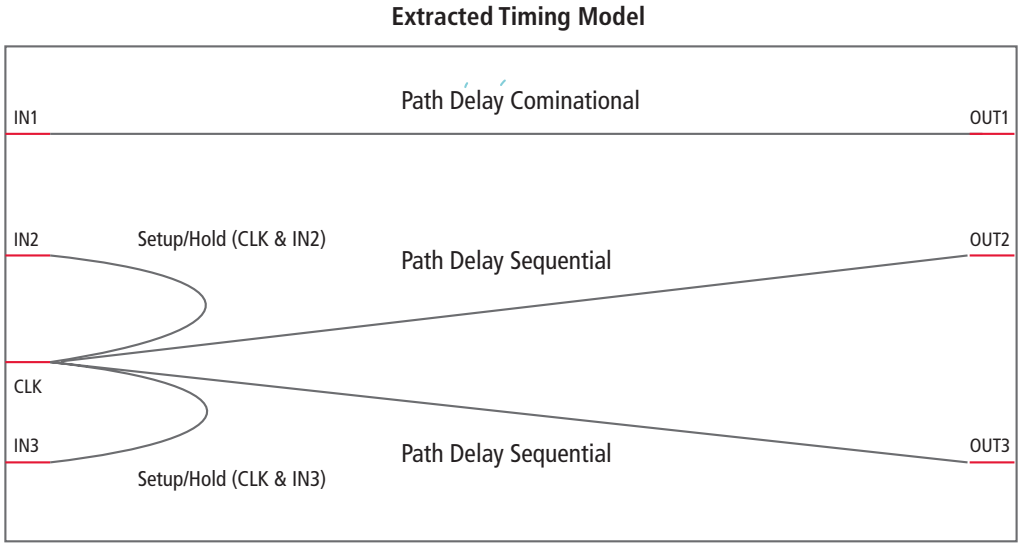


Figure 11: Timing Model Details

In the example, the timing model that is extracted is context-independent and does not contain timing details for the logical gates and flops inside it. It just contains timing information for path delays from input to output, and for setup and hold violations at inputs.

There is also no need to re-extract the model if some of the context gets changed at a later stage of development. This is because none of the boundary conditions or constraints (input_transitions, output loads, input_delays, and neither output_delays nor clock periods) are taken into consideration for extracting the model.

However, the model depends upon the operating conditions, wireload models, annotated delays/loads, and RC data present on internal nets defined in the original design. If these elements change at a development stage of design, then you need to re-extract the model for correlation with the changed scenario.

Advantages of Using Extracted Timing Models

The advantages of using extracted timing models include:

- Huge performance improvements both in STA and GLS because detailed timing is removed
- IP reuse and interchange of timing models
- IP protection in black boxing the design

3. Controlling or Handling Timing Checks Based on STA Reports

Since STA does the complete timing analysis, and in cases where timing for the complete or a portion of the design is already met, the timing checks during simulation might not be required for this portion of the design, specifically the internal flops.

As mentioned above in section 4, "Timing Checks" of 1.3.4 Incisive Enterprise Simulator Race Condition Correcting Features, timing checks have a significant impact on performance, so they can be turned off either completely or selectively, based on the requirement.

The STA and simulator flow is illustrated in the following figure.

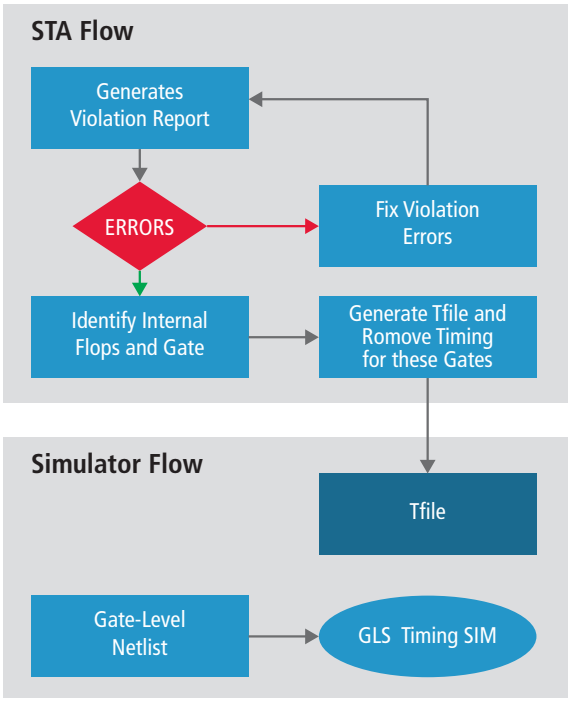


Figure 12: Timing File Example

Consider the example in Figure 8 above (titled Original Design). Based on STA, timing information for internal gates and flops is not required. In the figure below, an Incisive Enterprise Simulator timing file can be generated without timing information for gates marked in green.

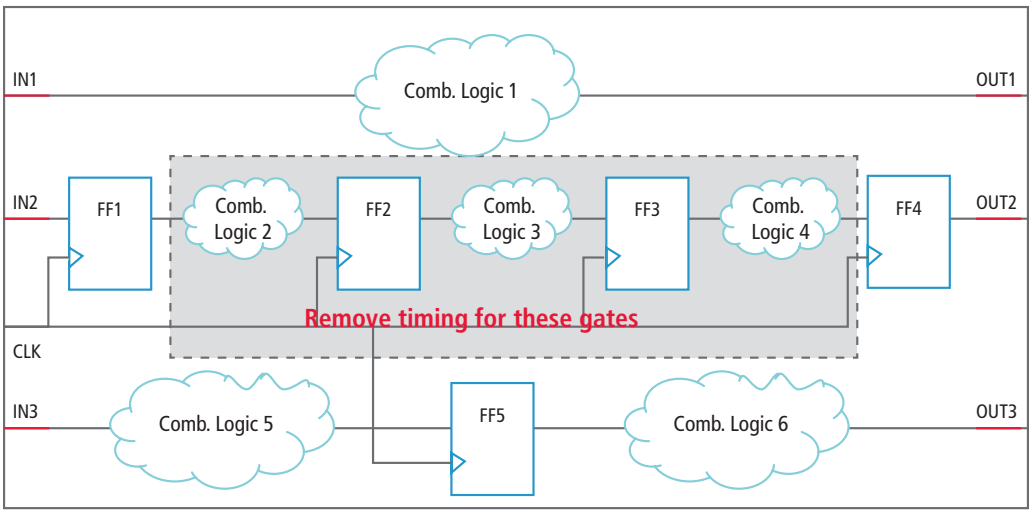


Figure 13: A Timing File Can Be Generated Without Timing Information for Gates

4. Focusing on Limitations of Static Timing Analysis and Logical Equivalence Tools

STA, or even logical equivalence tools, are not able to catch all the issues that can only be seen in a GLS run. You can define coverage tests around these limitations so that any functionality that is not handled by the static tools can be tested.

Limitations of STA include:

1. Inability to identify asynchronous interfaces

2. Static timing constraints like false paths and multi-cycle paths

Based on constraints, GLS can define the coverage points. Simulation must focus on these areas as they constitute the majority of issues in GLS.

5. Using DFT Verification

Gate-level DFT simulation is performed for the verification of test structures inserted by specialized DFT tools, such as Encounter Test. As designs have exploded in size and complexity, great advancements have taken place in automated test tools over the last decade in the area of scan chain insertion, compression logic to save I/Os and speed up testing, removal of hotspots, Built-in Self Test (BIST) logic, and so on.

However, these techniques and technologies are not the focus of this document. This section summarizes the motivations for DFT simulations, and offers you best practices that could optimize the running and debugging of long DFT simulations.

Simulation with Netlists

The motivations behind simulating netlists with test structures include:

- Functional equivalency checks
- Meeting timing requirements
- Fault coverage in the design
- BIST

Let's examine them one by one for better understanding.

Functional Equivalency Checks

Functional equivalency checks answer the very basic question: Did I change something in my design functionally by inserting additional scan or BIST logic? The following figure shows a simplified view of a scan chain (scan-in to scan-out) highlighted by the red path.

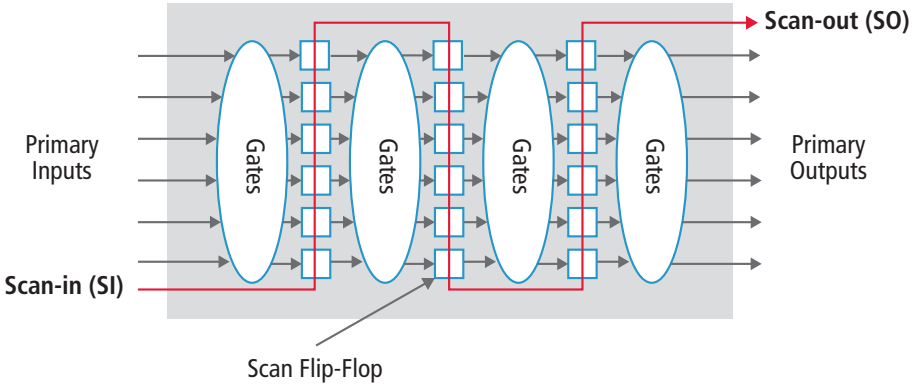


Figure 14: Simplified Scan Chain Representation in a Design

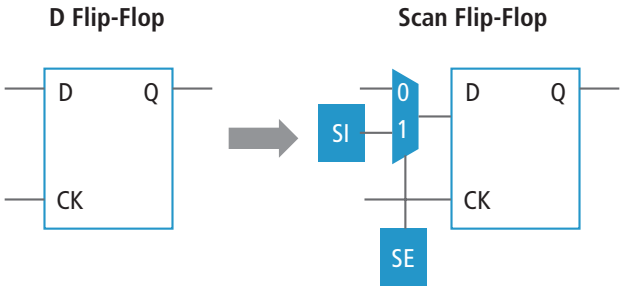


Figure 15: Basic Logical Representation of a Scan Flip-Flop

Each scan flip-flop, as shown in the figure above, introduces additional controls such as scan enable, scan input, and scan mux into the design.

Inserting these scan elements into a complex design with asynchronous I/Os, gated clocks, and latches is automated by test tools, and may introduce functional errors. The primary concerns here are to verify that all nets are connected with scan flip-flops, and that no unwanted functional effects are introduced.

Static equivalency checking is the primary method for catching functional errors introduced in the scan insertion phase. However, designers find GLS to be an additional insurance, and so they run simulations to find functional errors, as well as to verify logic inserted by the test tool. We call this effort Functional Integrity Simulation Testing (FIST).

Meeting Timing Requirements

"Have I met the timing requirements?" This is often the most challenging question because meeting timing requirements on SoCs with multiple asynchronous I/O blocks, multi-cycle paths/false paths, and clock speeds pushing into multi-GHz ranges makes timing closure a tremendous challenge for teams. And, including scan multiplexers adds additional gates in the critical path for functional operation, which increases flip-flop fan-out as shown in the figure above.

This step is done with a post-layout netlist, since routing and placement effects must be used to validate the timing of the chip. Design teams use constraint-based static timing delay checkers to validate that timing closure is met with test logic. However, multi-cycle paths and false paths are excluded from such checks. Therefore, timing annotated GLS is required to verify timing on multi-cycle paths. Often, simulation-based timing checkers are used as additional insurance against static timing tools to make sure human errors were not introduced in constraints that were specified statically.

Fault Coverage in the Design

Test tools provide input vectors and expected outputs, which are targeted to be run on automated test equipment to catch manufacturing defects. Some design teams run these vectors in simulation with toggle coverage enabled to measure how many wires in the design toggle. Using these vectors can give an independent gauge of testability.

BIST

For high-speed memory tests, a BIST controller is placed in the design to generate pseudo-random patterns that write to different memory banks. The memory is read back and compared against expected values to detect defects in memory arrays. Typically, BIST controllers are verified at an IP level and basic tests are performed to make sure a subset of all addressable memories can be accessed by the BIST controller.

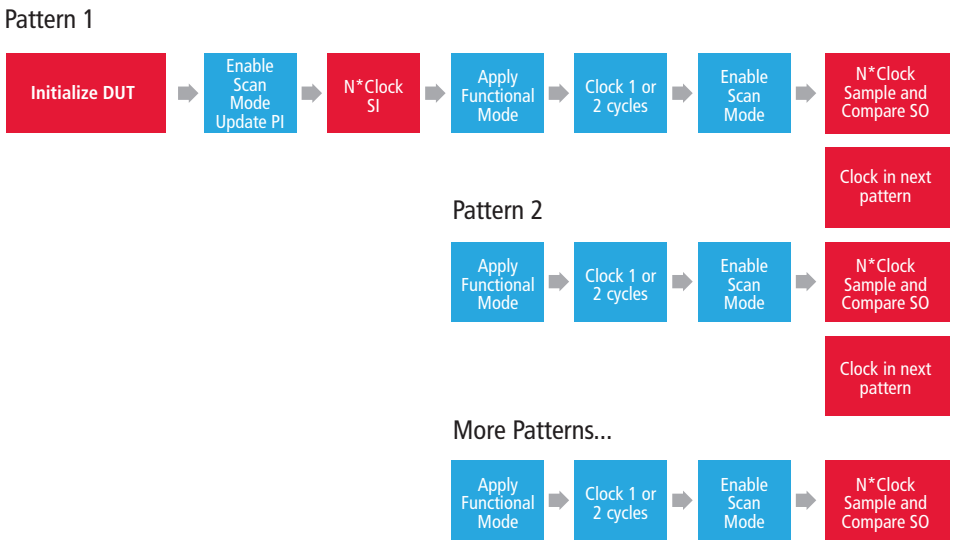


Figure 16: Typical ATPG Simulation Flow Using Serial Mode

Tests that run in seconds of real time at GHz speeds can take several hours to several days in simulation, depending upon a few important factors. The figure above shows a typical ATPG test flow.

The red boxes highlighted in the graphic indicate qualitatively where the majority of simulation time is spent. These time-consuming scenarios include:

a. Scan patterns clocked in serially. These patterns take "n" clock cycles to scan in and "n" cycles to scan out. Even with optimized scan chains, this could be thousands of clock cycles per pattern. Parallel load and unload techniques can be used to drastically reduce the simulation time in these cases.

b. SDF annotated timing simulations with advanced node libraries. These could take 4X-5X more wall-clock time and compute farm memory resources to simulate. Several verification concerns could be handled with a pre-layout unit delay netlist.

c. Design initialization time versus pattern simulation time. The simulation flow in Figure 15 shows a typical case.

Let's look at each of the above scenarios to examine trade-offs for different methods for the task at hand.

Functional Integrity Simulation Testing (FIST)

FIST should be done in pre-layout mode, without timing, to catch any errors introduced in the scan-chain insertion step. Two kinds of tests are required for this verification. First, a few functional tests in zero-delay mode should be simulated to build confidence in the test insertion. Second, a single ATPG pattern should be run in serial model that exercises all scan chains to verify scan chain integrity.

Depending upon the amount of compute resources available for the job at the time, a few additional, top-ranking patterns for simulation should be selected, based on coverage grading produced by the test tool. A hardware emulation solution, such as the Cadence Palladium® XP platform, can be used very effectively to verify the functional integrity between RTL and pre-layout netlists.

Timing Verification

Typically, this activity takes the bulk of the DFT simulation effort since simulations with timing annotated on large netlists are very slow, usually in the range of a few cycles per second to a few seconds per cycle, on the latest workstations. Simulating a single pattern could take several hours in serial mode. If 10-12 patterns are chained together in a single simulation run, a simulation run could take days.

A few techniques can be applied to improve simulation runtime and, therefore, the debug time associated with these simulations:

a. Use a compute-farm to break down long serial runs into several shorter simulation runs.

This involves trading-off compute efficiency versus turn-around time for validating late design changes and debug time. A calculation can be made to make the right trade-off. To do so, measure the DUT initialization time and call it $T_{initDUT}$. Now measure simulation times associated with a few patterns and take an average to compute average time per pattern, and call this $T_{pattern\_av}$.

Note that pattern simulation time will vary due to different event densities produced by different patterns.

Calculate the amount of time it takes to start a simulation on the farm and call it $T_{SimStart}$. Total simulation time for running "m" patterns serially should be:

Total simulation time on a single machine = $T_{SimStart} + T_{initDUT} + m \times T_{pattern\_av}$

By segmenting patterns into multiple parallel simulation runs, designers can reduce turn-around time significantly. We can apply constraint solving to identify the optimal solution for the minimum number of machines required to achieve a target regression time, as shown in the table below (all numbers are in minutes).

$T_{SimStart}$	$T_{DutInit}$	$T_{pattern\_av}$	Number of patterns	Desired regression time	1 sim run-time	Number of machines	Average single test time	Patterns per machine
30	60	50	80	500	4090	10	490	8
30	60	100	80	500	8090	20	490	4
30	60	150	80	500	12090	40	390	2
30	60	200	80	500	16090	40	490	2
30	60	250	80	500	20090	80	340	1

The table shows that partitioning long single-pattern simulations into multiple shorter simulations can achieve faster regression turn-around time. This step must be planned up-front so this step does not become the critical constraint right before tape-out.

b Use the parallel load and unload technique

Modern test tools, such as Encounter Test, provide an option to create testbenches and test patterns that use a simulator API, such as the Verilog Programming Interface (VPI) to back-door load and unload scan flip-flops. This technique eliminates the time-consuming activity of consuming processor cycles to *shift* data in and out, and to clock data in and out.

This technique should be used in combination with a few patterns tested in serial mode to ensure there are no timing issues in shifting data in and out of scan flops. Once that is done, the majority of the timing verification task can be achieved through the parallel load and unload technique. This technique checks various timing scenarios that could be exposed by different patterns, which can cause transitions to propagate by way of different logic paths in the circuit.

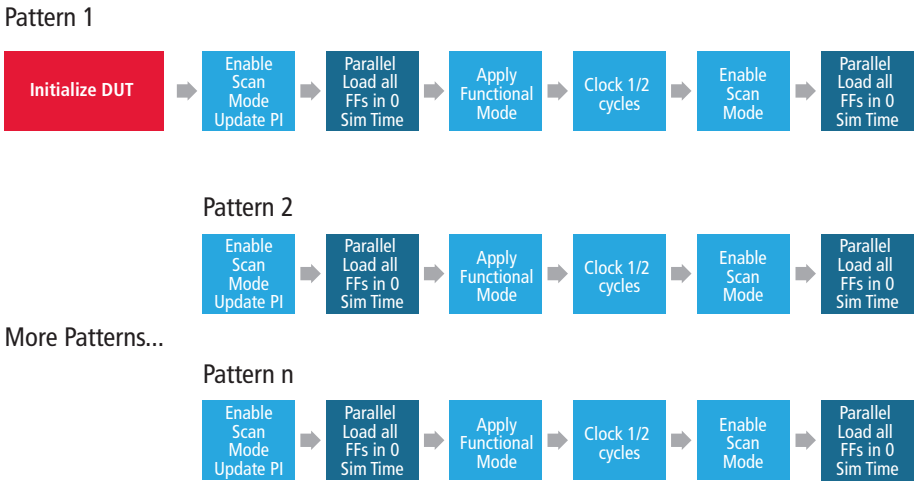


Figure 17: Typical ATPG Simulation Flow Using Parallel Mode

Fault Coverage

New DFT tools use sophisticated algorithms to insert test structures and report accurate fault coverage results. There is no need to use simulation tools to cross-check those DFT tools. If a cross-check must be done, simulations can be run without timing, and in parallel mode, as a confidence-building verification.

BIST Testing

The BIST technique is used to test memory faults in the device without using expensive ATE time. A stimulus generator and a response collector are placed in the design and hooked up to memory structures with a multi-plexer to select between BIST-generated stimulus and system stimulus, as illustrated in the following figure.

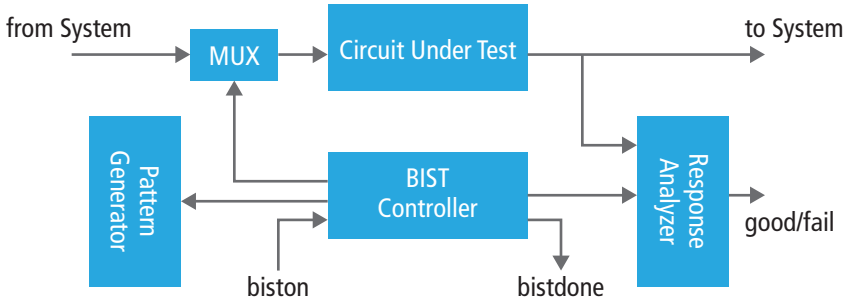


Figure 18: Typical BIST Architecture for Memory Verification

Functionally, the BIST controller and the response analyzer are exhaustively tested at the IP level. Therefore, the objective at the gate-level is to ensure connectivity and timing. A subset of BIST tests should be simulated with a post-layout netlist and with timing turned on selectively for components involved in the BIST path. Then, black-box test the rest of the DUT.

6. Catching Gate-Level Simulation X Mismatches at RTL Using Incisive Enterprise Simulator X-Propagation Solution

Standard Verilog semantics define X as an unknown value. This value appears when simulations cannot acutely resolve signals to 1 or 0. Unfortunately, RTL simulations often mask X values by converting unknown values into known values, while GLS show these hidden Xs. These X issues can be costly, adding complications as you familiarize yourself with abstract synthesized logic, while consuming time as you exhaustively simulate and synthesize designs for verification purposes.

The Incisive Enterprise Simulator x-propagation feature enables the same behavior as that of hardware at the RTL level to identify the 'X' issues much earlier in the design phase, saving costly GLS and debugging effort.

X-Optimism Problem at RTL

To understand the 'X' optimism issue, let's consider a simple "if" block in the figure below. In the case of RTL, if the select signal is 'X' then, according to LRM schematics, else block of the "if" statement is executed. In case of real hardware depending upon the values of input signals 'A' and 'B', the output will be computed. In this case, if both 'A' and 'B' have the same value, then the output will be the same. But if the values of input signals are different, then the output will be non-deterministic.

The truth table below shows the difference between the two.

This can also be seen in for case statement and if the select goes to 'X', there is a difference between the behaviors of RTL vs hardware.

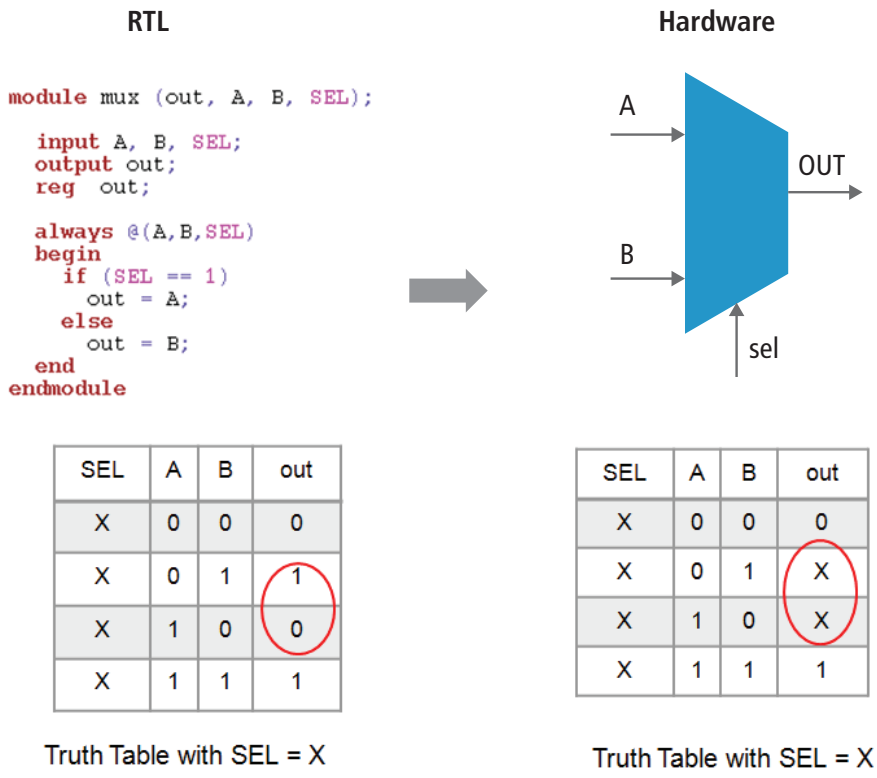


Figure 19: X-optimism

Similarly, the 'X' optimism problem can be seen in a sequential-edge triggered block where the edge goes from '0' or '1' to 'X' or 'X' to 1 or '0'. Support is available in Incisive Enterprise Simulator to handle such edge-triggered 'X' issues at the RTL level.

To learn how to enable the x-propagation function, please refer to the user's guide.

7. Black-Boxing Modules Based on the Test Activity

Black-boxing some of the modules can be a good approach to improving GLS performance, as typically each test verifies only a certain portion of the design. Also, the IP cores that are already verified can be black-boxed completely and only their interface-level details are required for verifying the other portions of the design.

Black-boxing can have a positive performance impact on:

1. Reducing elaboration and simulation memory footprint because the details of the module are missing

2. Reducing elaboration time

8. Saving and Restarting Simulations

Typically, much GLS time is spent in the initialization phase. This time can sometimes be very significant. As a result, a single simulation should be saved and all other (n-1) simulations should be run from the saved checkpoint snapshot. The figure below describes this flow.

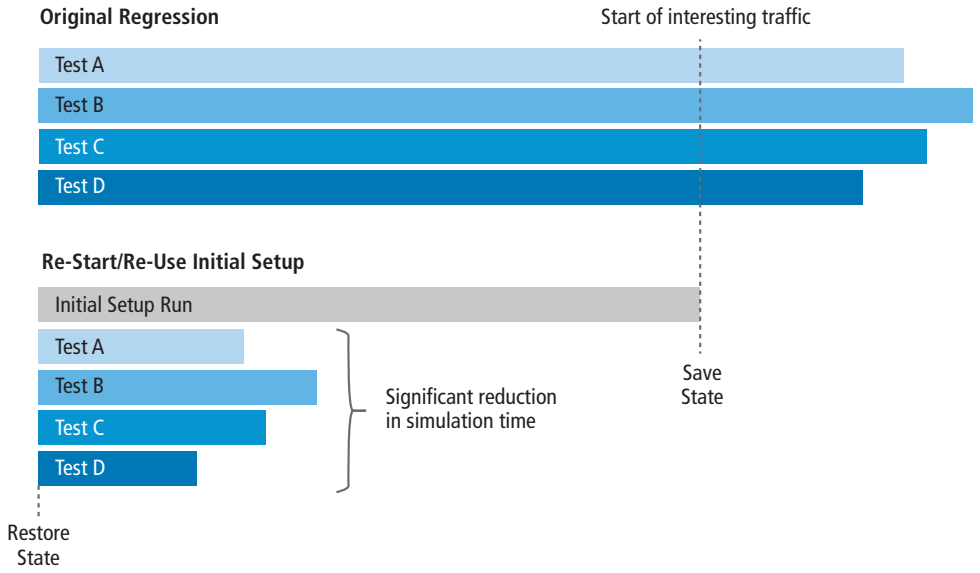


Figure 20: Saving the State After Initialization Saves Simulation Time

Library Modeling Recommendation

Complex timing checks in 40nm technology nodes and below are responsible for longer run times and high memory requirements. Writing library models efficiently can help improve the performance of GLS with and without timing.

Let's consider a simple example of a finer technology node, where the timing checks in specific blocks not only have conditional logic (marked in red), but also the conditions require extra logic (marked in blue). Similarly, the I/O path delays have conditional paths (marked in green).

```
module FF1 ( Q, TE, CPN, E );
    input TE, CPN, E;
    output Q;
    ……..
    ……..
```

```
    not (nTE, TE);
    and (E_TE, E, TE);
    mybuf (E_TE_SDFCHK, E_TE, 1'b1);
    mybuf (nTE_SDFCHK, nTE, 1'b1);
  ……
 ……..……

 specify
    if (E == 1'b0 && TE == 1'b0)
    (posedge CPN => (Q+:1'b1)) = (delay1, delay2);
    if (E == 1'b1 && TE == 1'b1)
    (CPN => Q) = (delay3, delay4);
    $width (posedge CPN &&& E_TE_SDFCHK, 0, 0, notifier);
    $setuphold (negedge CPN &&& nTE_SDFCHK, posedge E , 0, 0, notifier);
endspecify
```

Due to these additional conditional statements and additional logic, the events on the simulator increase significantly, as shown in Figure 21 below.

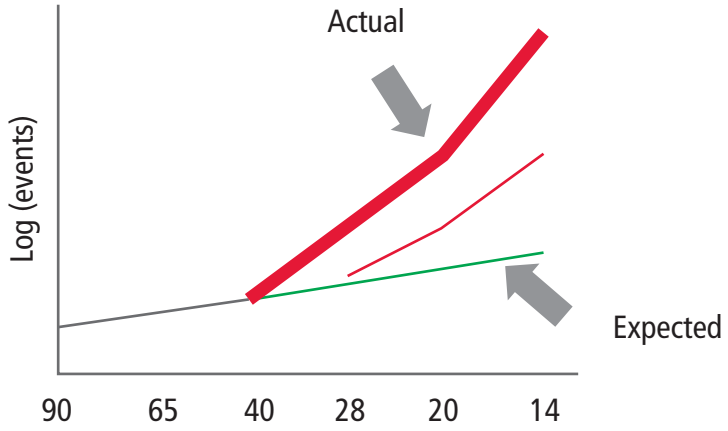


Figure 21: How Additional Conditional Statements and Logic Impact Simulation Events

In order to reduce the number of events in the simulation, consider the following while writing a library:

1. Since conditional timing checks require extra logic and are more complex, check the following:

   a. Whether the conditions can be reduced by removing unnecessary, insignificant conditions

   b. Whether the conditional logic can re-used or merged for multiple cases

2. Use of # delays in libraries should be avoided if the simulation is running with timing mode as using the absolute # delays turns off simulator optimizations. These delays can be used only for functional mode, i.e. zero-delay GLS. The example below shows the library models having functional mode for zero-delay simulations and pure timing mode.

```
module FF (clk , d, q ) ;

   input clk;
   input d;
   output q;
……..

`ifdef FUNCTIONAL
//For pure zero delay simulations, models have #1 delays in sequential cells to avoid race
conditions.

        not     #1          NOT1( q, q1) ;

`else
```

```
//For timing simulations, models do not have # delays in sequential cells.

        not         NOT1( q, q1) ;

`endif
```

1. Use the minimum number of gates to represent the functionality. A smaller number of gates results in a smaller number of events during the simulation.For example, use a three-input and gate instead of two two-input and gates.

```
// Use of 3 input and gate
  and and1(q, a,b, c);

// Instead of two 2 input and gates
   and and1(out1, a, b);
   and and2(q, out1, c);
```

2. Logic on the CLK signal should be minimized; also, bear in mind that a heavier load on the CLK signal slows simulation.

Summary

We have shown that in order to improve GLS performance, both the simulator runtime performance as well as the simulation methodology need to be improved.

The runtime performance of Incisive Enterprise Simulator has been continuously improved for

GLS. In addition, some recent enhancements in the simulator can now deliver from 1.2X to 8X performance improvements for a particular design.

There are also continuous improvement efforts going on in the simulator development area to improve

GLS performance even further. However, in order to match the verification requirements for newer, larger designs with increased complexities, a combined simulation and methodology approach has to be taken in order to achieve an effective and efficient verification process.

Contacts

Gagandeep Singh gagans@cadence.com

Amit Dua adua@cadence.com