cadence®

Hardware Simulator Performance Scaling to Meet Advanced Node SoC Verification Requirements

By Amit Dua, Adam Sherer, and Umer Yousafzai—Cadence Design Systems

Because of its flexibility, hardware simulator–based verification is expected to scale with design complexity. Continuous innovation has enabled hardware simulators to reduce both memory and overall turnaround time. The Cadence® Incisive® Enterprise Simulator not only embodies these characteristics, but also provides the new controls, technologies, and methodologies necessary to meet SoC verification performance requirements at sub-40nm. This technical paper examines the Cadence systematic approach to optimizing verification performance and productivity.

Contents

Introduction

"The state-space of a chip will grow exponentially every 24 months" is the verification engineer's corollary to Moore's Law. The block or IP-level components are now as big as full systems on chip (SoCs) were five years ago. Today's SoCs are growing beyond multiple hundred-million gates. While the verification engineering community has created several new tools and methodologies, these are primarily designed on top of two technologies developed in the 1980s to simulate circuits expressed using a hardware description language (HDL)—the hardware simulator and the hardware accelerator/emulator.

Because of its flexibility, hardware simulator–based verification is more widely used for verification tasks and is therefore expected to scale with the exploding design complexity. This expectation is rational even after a generation of use because of the continuous innovation that has enabled the hardware simulator to reduce memory while speeding execution and overall turnaround time. The Cadence Incisive Enterprise Simulator not only embodies these characteristics, but it also provides the new controls, technologies, and methodologies necessary to meet the SoC verification performance requirements at 40nm and finer geometries.

While the number of knobs available to tune performance has grown with verification complexity, the approach to addressing project-specific requirements remains systematic. The first step to doing so is obviously to analyze the design and verification requirements; a careful examination will identify the areas that can be improved and to what extent those improvements will affect overall turnaround time. If it is a derivative testbench and design, then simply moving to the latest simulator release is all that's needed to improve productivity. However, if there are verification environment features, or if the deployment of a new technology/methodology causes a significant change to the design and/or testbench, then a more detailed profile

is needed to identify performance bottlenecks due to coding styles, engine optimizations, or both. If the project is undergoing a larger design/verification expansion, then a more structured approach is needed for performance and productivity.

This technical paper is aligned with this systematic approach to verification, each section focusing on a project-specific method for improving verification performance and productivity.

Innovation in the Core Simulator

The assigned task may be to "simulate this design" and the good news is that verification language standards enable interoperability, so project teams can choose. After a generation of development, what sets the simulator providers apart is the ability of their engineering teams to innovate both core performance and structured solutions. What makes the core performance so important is that it pervades all verification tasks—design, subsystem verification, SoC integration, and gate-level simulation of the synthesis output—and does so with minimal methodology impact.

Better simulation through better engineering

Functional verification at the register-transfer level (RTL) is expected to continue being the predominant verification technique. One way to increase performance and productivity is to move up to higher level of abstractions like transaction-level modeling (TLM). But in the other direction, gate-level simulation continues to be the primary signoff criteria. As a result, simulation-based verification holds the key at all level of abstraction. The requirements on RTL performance have evolved dramatically in the last 25 years of the commercial simulation industry, driven by the dramatic expansion of new languages and extensions to them compounded by the amount of code that is required to express the increased complexity of design and verification.

"Dramatic expansion" must have metrics to understand how this led to a new generation of performance innovation. For years, simulation engineers have created reference examples to validate the scalability of the simulator. The Incisive Verification Kit is a simple SoC developed in early 2000 equaling about 1.5M gates. At the same time, the Incisive Enterprise Simulator is being applied to actual designs approaching 200M gates and more, and will certainly be applied to larger designs in the future. This growing gap between contrived and actual designs led the Cadence Incisive team to form close working relationships with several large electronics firms to access their leading-edge designs. The result is a wave of optimizations born of real-world test cases and delivered to the broad user community in each new release of the Incisive Enterprise Simulator.

All the examples of Incisive simulation innovation are too numerous to cover in this paper, but select examples can provide credence to the new opportunities and head-room for performance gains in the core. Here are a few of the improvements made during 2011:

- **Assertion-based verification:** Assertion-based verification (ABV) has matured from a scattering of checker instances to a core design requirement with some project teams defining "assertion density" metrics as much as one assertion per ten lines of design code or more. A single finish for each cover property, optimizations focused on SystemVerilog Assertions (SVA) sequence operators, and performance controls (such as limiting the failures/finish per assertion and user control of assertion evaluation at each stage of the compile/elaborate/simulation cycle) optimize performance for both subsystem and full SoC verification.
- **Gate-level simulation:** Gate-level timing checks became significantly more complex beginning with the 40nm node. A careful examination of the design and library found a much greater use of complex expressions in timing outputs, creating an opportunity for optimization. Specifically, the complex expressions can be in the continuous assignments or through multi-gates that then drive the timing check conditions. While optimizations like this one have some benefit in smaller designs, most of the memory and runtime benefits occur in designs that elaborate larger than 8GB, which is becoming the norm at 40nm and below.
- **Coverage:** In the span of a few years, mainstream coverage collection has moved from simple code coverage to complex functional coverage. As a result, coverage data has grown in two dimensions—the number of bins tracking the design plus testbench and the number of coverage data files required to merge from multiple tests. The former includes designs up through the 100M gate equivalent range and larger, while the latter includes test suites that range up through the hundreds of thousands of merges and more.

Optimizations for mixed-language dumping, dynamic SystemVerilog objects, toggle coverage, and more all contribute to runtime improvements while union merge, storage of evaluated expression types, auto bin range allocation/de-allocation, and others all contribute to memory improvements. As the metric-driven verification methodology pioneered by Cadence in the mid 2000s and Accellera's Universal Verification Methodology continue to proliferate, additional innovation opportunities will appear for coverage.

- **Power-aware design:** Power-aware design is transitioning from a feature that's needed for specific applications to one that's a basic physical requirement to enable advanced node SoCs to work. At the same time, the complexity of the power modes and their transitions is growing to the point where directed tests are no longer sufficient to verify that the SoC will operate properly under all power conditions. This implies that the simulator needs to be fast enough to run every functional test as a power-aware functional test. Incisive technology offers this capability with a native low-power solution that has marginal overhead during both elaboration and runtime.

The data in Table 1 provides insight into how the Incisive engineering team measures itself. As members of the team build optimizations such as the ones described for assertions, gate-level simulation, and coverage, these features are rolled into specific software releases and measured against multiple test cases. For a project team that starts with a specific software release, they will experience some of the improvements in performance in that release and will look to upgrade to the subsequent release for additional improvements.

	Representative Simulation Features	Gains from 9.2 Base to 10.2 Latest Release	Gains from 10.2 Base to 11.1 Latest Release	Cumulative Gains
Turn-Around Time (TAT)	RTL elaboration	1.6 – 2.8x	1.0 – 1.2x	1.6 – 3.4x
	SystemVerilog elaboration memory	1.0 – 1.2x	1.0x	1.0 – 1.2x
Simulation Run Time	Verilog RTL	1.0 – 1.3x	1.0 – 1.3x	1.0 – 1.7x
	ABV performance	1.6 – 7x	1.0 – 1.2x	1.0 – 8.4x
	SystemVerilog performance	1.0 – 1.2x	1.0 – 1.3x	1.0 – 1.3x
	Constraint solver performance	1.5 – 2.0x	1.2 – 1.7x	1.8 - 3.4x
	Coverage write performance	1.6 – 2.7x	1.1 – 7x	1.8 – 19x
	GLS performance	1.0 – 5.7x	1.0 – 1.4x	1.0 – 8.0x
Simulation Memory	Verilog RTL memory	1.0 – 1.2x	1.0 – 1.2x	1.0 – 1.4x
	SystemVerilog memory	1.1 – 1.3x	1.0 – 1.2x	1.1 – 1.6x

Table 1: Representative Incisive Enterprise Simulator release-to-release performance gains

The data in Table 1 measures the statistical mode for the tests in each category and uses a single standard deviation to create the range. Since performance of a software application is often a tradeoff between memory and runtime, a specific optimization may reduce one at the expense of the other. The test suite helps Cadence identify such performance tradeoffs and make good engineering judgments that help improve performance for the majority of the use cases. The Incisive engineering team also measures the number of tests that appear above and below the mode to ensure that the performance of the overall suite improves. Subsequent updates of this paper will include updates to this table.

Configuring the simulator for speed

The points in the previous section outline the forces of change that are acting on the simulator as well as the subsequent changes to the simulator itself. With respect to enhancements like those to gate-level simulation for advanced node libraries such as 40nm and below can benefit by simply upgrading to the latest simulator. However, the enhancement may be overshadowed by other factors that reduce perceived benefit of the enhancement. For example, if only select instance hierarchies of the design are open for debug access, the effects will be much greater in terms of overall runtime improvement compared to running the design with complete access. There, users may want to consider modifying their run scripts, where appropriate, to maximize the improvements in the core simulator.

By default (no options) the simulator runs in a fast mode with minimal debugging capability. To externally refer to design objects (e.g. through PLI/VPI/VHPI or tcl) or to single-step through lines of code during simulation requires using debug options. These options provide visibility into different parts of a design in exchange for speed, but judicious use of them provides a balanced approach to optimizing performance.

When global debug options are applied, like "–linedebug" or "–access rw[c]," the optimizations get switched off on the entire design. In general, these options should not be used for regression. For debug and on large designs, they should be applied selectively rather than globally. "-linedebug" is a compile-time switch, so it can be selectively applied to source files that need step debugging. Running without "-linedebug" mode is generally 2x faster, or more, when applied to all sources.

The external access to design objects can be limited by passing an access file to the elaborator, which contains access permissions for specific instances or objects in the design. It is important to note that the access permission is required only for external references from PLI and tcl; it is not required for any access from Verilog, SystemVerilog, or VHDL source code, nc_mirror, or for out-of-module references. The access file option ("-afile filename") lets the user specify only those objects that need access. As a result, the simulator can optimize objects that do not need to be accessed. The elaboration option "-genafile" is used to automatically generate an access file. The impact of using an access file over "-access rw" depends on the number of objects in the access file. Fewer objects in the access file yields more gain. The typical range is a 1.05–1.5x improvement by reducing the access level.

CPU, cache, memory, storage, network, and operating system can also affect performance. For example, large simulations, including top-level SoC tests and gate-level simulations, use large amounts of memory whereas smaller subsystem regression simulations may be dominated by turnaround time in both elaboration and runtime. Regardless, the optimal performance is achieved when the simulation fits within physical memory. The -status flag in the Incisive simulation environment will report the memory requested by the Incisive Enterprise Simulator. In the situation where the simulator executable, or snapshot, is very large, it will load faster if it is copied to local storage rather than accessed across a network. Regardless of the size of the design, larger caches are better with at least 8M of cache being the lower limit when a project team upgrades their regression farm.

There are many more options and methodology changes that can be used to extract more performance. The simplest and fastest way to configure the simulator for a project's performance needs is to work with a Cadence expert to conduct a performance audit. The expert will work closely with the project team onsite and suggest performance improvement techniques and methodology changes.

An example of the benefit of such an audit would be to improve performance during the gate-level simulation phase of a project. To start with, there are a few options including running with different SDF precision, disabling negative timing checks, or even running in zero or unit delay mode. All timing values (including those less than 10ps) and all negative timing checks are used by default in the Incisive Enterprise Simulator. This results in more accurate timing simulation, but higher memory consumption and slower simulation speed. Verilog timing simulation can be run faster by using a larger timescale and without negative timing checks. A performance audit will yield many more suggestions, such as how to make optimal use of farm machines for better regression performance, the management of compiled libraries, and so on.

Using a profiler to tune performance

Software engineers have used profiling tools for years to improve algorithm performance, and that technology is increasingly important to verification engineers as the environments increasingly resemble large software projects. The profiler is a tool that measures where time is spent during simulation. It helps the project team understand and optimize the environment for improved performance.

It is recommended that project teams profile the simulation environment regularly. The best practice is to run it at every major change in environment. When writing code, engineers tend to focus first on functional accuracy—and that is the right thing to do. In some cases, the initial algorithm tradeoffs include coding styles that are easier to understand but may be inefficient. Profiling the environment can help identify those bottlenecks as the code is maintained and developed. The value of the profiler is that it indicates to the user exactly where the runtime is spent and, in many cases, clearly points to the unexpected effects of a particular coding style. In this way, the profiler points out an opportunity to improve performance.

The Incisive Enterprise Simulator has a built-in profiler that provides basic information in a text file. It can be enabled by simply adding the -profile option at runtime. There is very low overhead on runtime performance and memory for collecting the statistics in a simulation.

The Incisive profiler report is divided into several sections, each of which provides an aspect of the overall performance picture. The header section contains information about the machine details, elaboration options, simulation time, CPU, and memory usage. The stream counts section reports the blocks or the tasks that consume most of the time. It provides information about the time spent in individual code streams such as always blocks, initial blocks, VHDL processes, tasks and functions, continuous assignments, non-blocking assignments, and other language features. If an HDL construct appears unexpectedly high in the list, it may have been written inefficiently. For example, in Figure 1, the stream counts section reports that the highest percentage of total time—31% —is spent in one VHDL process block.

```
---------------------------------------------------------------
Stream Counts (230526 hits total)
---------------------------------------------------------------
%hits #hits #inst name
31.4 72380 [ ] Process statement: $PROCESS_000 (line: 64, in design
unit
COMN_LIB.C_LCBCTL:C_LCBCTL)
3.2 7407 [ ] Process statement: $PROCESS_002 (line: 82, in design unit
COMN_LIB.C_LCBBAS:C_LCBBAS)
2.6 6070 [ ] Method SSS_MT_RETURN_BYTE (method)
1.0 2245 [ ] Process statement: $PROCESS_001 (line: 69, in design unit
COMN_LIB.C_LCBBASE:C_LCBBASE)
0.9 2105 [ ] Process statement: $PROCESS_000 (line: 64, in design unit
COMN_LIB.C_LCBCTL:C_LCBCTL)
…
```

Figure 1: Language features in the stream counts section of the profiler report

There is also a summary section that makes it easier to identify widespread inefficiencies in the simulation. For example, large amounts of time spent on always blocks, probing, file I/O, and PLI applications will show up most clearly in this section. The example in Figure 2 shows the summary section for a typical Verilog and VHDL mixed-language RTL design.

```
---------------------------------------------------------------
Stream Type Summary Counts (19154 hits total)
---------------------------------------------------------------
%hits  #hits  #inst name
35.0   6708 [ ] Always statements
21.7  4153 [14474] Verilog functions
15.6  2997 [14971] VHDL process
11.7  2245 [ ] Non-blocking assignments
11.4  2176 [ 6452] Simulation Engine Time
0.9   164 [ 7528] Engine support
0.9   163 [ ] Verilog tasks
0.8   150 [ 139] VCD/SHM variable dumping
0.7   140 [ 7549] Continuous assignments
0.5   96 [ 331] Parallel block sub-processes
0.5   93 [ ] User-defined primitives
0.3   59 [ ] Logic primitives
0.1   23 [ 4152] Initial statements
0.1   18 [ ] VHDL subprogram
0.0   7 [ ] User defined system tasks/functions
0.0   1 [ ] Support for VPI callbacks (or UI)
```

Figure 2: Summary section of the profiler report

As design sizes become large and verification environments become more complex, there is a need for more advanced profiling. An advanced profiler has to not only provide a means for managing the greater amount of data, but it also has to accommodate the allocation of time to objects that can be created and destroyed during simulation (hardware descriptions can't do either) and a flow of control that moves across threads of execution rather than by hierarchy.

The Incisive advanced profiler supports features including instance-based profiling, finer basic block-level information, and a call graph for object-oriented code. The instance-level information is useful to know the time spend in a sub-scope within an IP block. This can be used to replace the time-consuming IP blocks with a fast model. The call graph is very useful for class-based verification environments such as those written with the Open Verification Methodology (OVM) or Universal Verification Methodology (UVM) libraries. An example of this more advanced profiler information is shown in Figure 3.

```
iprof> report -detail -metrics callgraph
```

```
Index   Cumm                  Self                  No. Calls   Name
-------------------------------------------------------------------------------
        1.16% (162/13933)     0.00% (0/13933)       158         scoreboard.do_func_phase [189]
        3.22% (448/13933)     0.00% (0/13933)       435         snooper.do_func_phase [84]
        3.52% (490/13933)     0.00% (0/13933)       479         observer.do_func_phase [128]
[9]    16.34% (2277/13933)    0.01% (1/13933)             build_phase:call_func [9]
        0.05% (7/13933)       0.31% (43/13933)      7           ovm_phase:set_executed [299]
        3.42% (477/13933)     0.02% (3/13933)       478         observer.build [129]
        0.01% (2/13933)       0.01% (2/13933)       2           ovm_phase:call_func [1732]
        0.06% (9/13933)       0.00% (0/13933)       9           ovm_component:build [295]
        0.12% (17/13933)      0.00% (0/13933)       17          score_imp.build [1009]
        3.12% (435/13933)     0.00% (0/13933)       435         snooper.build [85]
-------------------------------------------------------------------------------
```

callers

func node

callees

`cumm` - cumulative profile grade of the entire callgraph rooted at the function

`self` – profile grade of only the function

`numbers in parenthesis` – # of samples in the instance / total no. of samples

`No. calls` – number of times caller called func node or func node called callee

Figure 3: Output from an advanced profiler

Advanced Node SoC Verification Requirements

While tuning the engine can provide single-digit performance multiples, advanced node SoC designs at 40nm and below demand structured approaches to simulation technologies and a methodology to achieve greater gains. The first requirement encountered by all project teams is to reduce turnaround time (TAT) at each stage of verification, as shown in Figure 4. To do this, we need to look at the other steps in the simulation process that consume significant time. Quite often that means speeding the assembly, or elaboration, stage. Incremental elaboration comes over from the software world, where subsystems can be compiled into linkable objects and then those linkable objects are linked together. Then, only those subsystems that have changed are re-compiled. This reduces TAT by orders of magnitude when minor changes are made.

A similar process can be applied at runtime by executing the common portion of a set of tests once and then restoring and reseeding each of those tests from that common point to reduce debug and regression time significantly. In each of these steps, some minimal changes in design and verification methodology are needed to achieve the large performance improvements. The same is true as simulation engines shift from single- to multi-core execution. Taken together with a tuned core engine, these new technologies will enable simulation to meet the performance requirements of advanced node SoC verification.

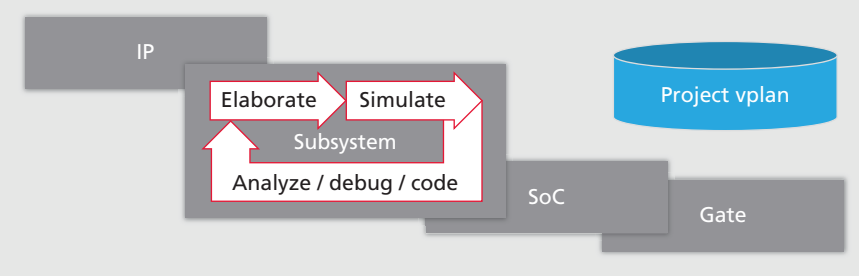

Figure 4: Turnaround time affects each verification stage for advanced node SoCs

Incremental elaboration

SoC projects typically distribute the development across multiple teams, creating natural lines of division in the SoC. Verification engineers usually don't make changes to the design under test (DUT) as they verify it with various test scenarios. That allows the test or verification engineers generally to work on a fixed version of the DUT and verify it with various test scenarios. They typically change only the verification environment (testbench or just the tests). This division worked naturally in the past as the common verification languages (like ***e***, C/C++, and Vera) were compiled outside the design model build by HDL simulators.

However, simulators have upgraded significantly in the last few years and have incorporated the verification languages as an integral part of their engines. The verification languages became compiled natively into a unified environment for both design and verification. In the early use of SystemVerilog, the environments were generally small enough that there was little additional overhead. However, with the wide acceptance of the OVM, and the UVM built from it, the SystemVerilog content grew so quickly that it is now a common requirement to elaborate the testbench separately from DUT to reduce TAT.

The Incisive Enterprise Simulator provides the ability to split the environment into parts that are elaborated separately and combined at the simulation step. This capability enables fast elaboration of the design and verification environment in many common situations, including the following:

- Changes local to the testbench or the verification environment, but not in the design
- Linking different testbench hierarchies to a design that has been elaborated once
- Changes local to a design when the rest of the verification environment is unchanged

This feature is most useful during the debugging phase as it enables very fast TAT after small modifications. It significantly boosts productivity and performance by separately elaborating the small changing portion from the once-elaborated or large fixed-portion of the design and verification environment. SoC project teams are deploying this technology to reduce TAT from hours to minutes and are driving additional flexibility into the solution as it becomes a mainstream approach to structured verification.

Dynamic load and reseeding

Regression simulation, the execution of waves of tests on a farm of servers, is a common verification practice. In regression simulation, the randomized tests are written to validate specific features and complete in a fixed period of time between a few seconds and a few hours. In many cases, a large number of tests will require a common design initialization or interface protocol training phase. While a single test can verify this startup, all of the tests in the group run the same phase, resulting in wasted farm cycles.

Dynamic reseeding eliminates this redundancy, reducing overall regression time. Working on real designs, gains of as much as 3x have been observed. The key is being able to save the state of simulation, including any attached PLI/VPI/VIP, restore a simulation from this saved state, and then reseed the restored environment to start the unique test. By doing so, each test that shares the same initialization is now that much shorter. This means many more waves of regression can execute on the farm in the same amount of time, resulting in a reduction of the total regression time. Many groups leverage this reduction in regression time to run many more simulations, resulting in a gain in functional coverage results. This effort does require a small amount of methodology change to understand which tests share the same initialization routine, but the design and testbench do not have to change. Once set up for regression, reseeding also can improve TAT for debugging failing simulations.

An example of this structured approach is shown in Figure 5. The first step is to identify all of the tests that share a common reset, protocol programming, or other type of initial sequence. Then a single test is run through that sequence and the state is saved. From that point, each of the individual tests can be executed with a much shorter run. Tests, such as test D in the example, can even be reseeded. The structured aspect is just to identify the tests that meet these criteria and to schedule them together with the load-balancing software to improve regression farm performance.

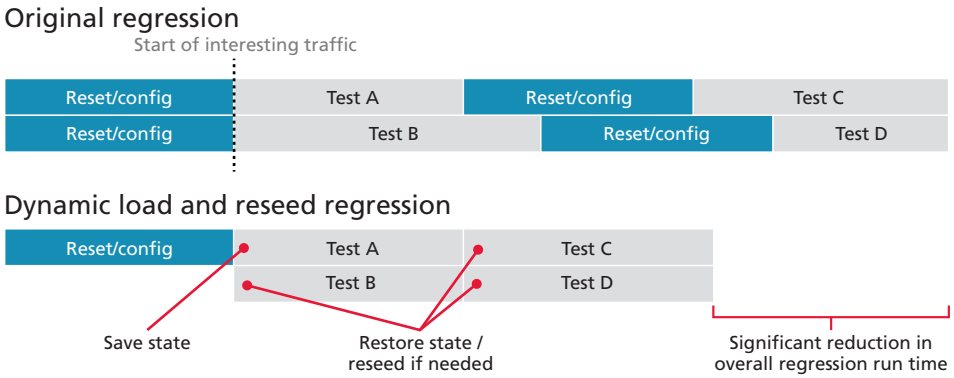

Figure 5: Dynamic load and reseeding

Multi-core simulation

Multi-core simulation can improve verification performance by distributing the regression runs among multiple cores. Almost all machines running simulation today are multi-core. There are different ways to use them for faster performance.

The most common way to exploit them is by running multiple tests on one machine with each simulation using a single core. This technique is commonly used to boost the performance of regressions using Incisive Enterprise Manager or with simple load-balancing tools. However, there are cases when there are specific simulation jobs that are very long and the profile data attributes much of the execution time to applications like waveform dumping, generation of code streams, and more. Depending on the application, the Incisive Enterprise Simulator can provide 1.1-1.4x additional performance using multiple cores in this situation.

The other mechanism is to partition a large design and simulate it in parts on the multiple cores available on a machine. The overall verification environment is subject to improvements because the event distribution in an SoC is very specific to the test being run and may vary greatly from one test to the next. For example, in a gate-level simulation without timing, long simulation threads can be found that are then loaded onto different cores for execution for very significant gains. When timing is added, those threads can fracture into very short events that lose their parallel nature. By doing so, performance gains of 1.5x on two cores and 2.9x on four cores have been observed on a representative SoC design.

The best way to determine if a design can benefit from multi-core support is to use a special thread profiler to determine the level of parallelism in the environment. Doing so may lead to tuning in the testbench to inject more parallel traffic into the environment so that the functional coverage goals are met and the environment is more suited for multi-core simulation. Regardless, the application of multiple cores to improve simulation execution performance shows promise for advanced node SoCs.

Project-level performance optimization

While compilation, elaboration, and simulation performance is very important for users, verification performance needs to take into account the "high value engineer performance (HVEP)" in achieving verification goals. Cadence is constantly innovating to provide better debug, coverage analysis, and automation capabilities to improve HVEP. These innovations to improve project-level performance can be organized into two categories: novel verification technologies and novel methodologies.

Some examples of novel verification technologies include leveraging formal engines and hardware-based acceleration. Formal engines, such as the Incisive Enterprise Verifier, can be applied to signal connectivity checking. Doing so is an efficient way to check that buses and I/Os are hooked up correctly without having to simulate the full SoC or without even having an SoC-level testbench. That translates into overall project time reduction because designers have a way to debug low-hanging problems early. This particular application of the formal engine also requires no knowledge of assertions or even formal verification.

Another way to apply formal techniques to reduce the project cycle is to identify design code that is unreachable to adjust code coverage goals. This coverage unreachability flow provides automation to verification engineers so their valuable time is focused on hitting coverage goals that are actually achievable. Similarly, verification engineers use coverage ranking to identify test runs that are not adding unique coverage hits to assess if the test has a bug or is redundant.

Hardware-based acceleration is a way for project teams that are trying to find 2–3x gains from simulation to break through to 10–50x gains or more. For complex SoCs that include mixed-signal blocks, acceleration can still be part of a structured approach that leverages abstract models for the analog blocks. With additional testbench tuning, the acceleration can achieve 500–1,000x gains with no degradation associated with larger design size

Some examples of novel methodologies include low-power simulation at RTL, digital/mixed-signal modeling, and metric-driven verification. With power being the bane of many SoC designs, more and more designs include power-aware domains. These domains depend on physical characteristics of the chip, such as power rails and isolation cell location, which have not been traditionally modeled in either gate or RTL abstractions. The Incisive Enterprise Simulator, with its direct connection to Cadence Encounter® Conformal® Low Power, provides the accuracy and native simulation performance needed to execute every regression test as a power-aware test. Another aspect of SoC designs is that they all have analog blocks. Modeling those using transistor or traditional analog behavioral modeling slows the execution of the SoC tests to that of the analog model. By modeling those blocks using wreals, or the emerging SystemVerilog modeling standard, the entire SoC can execute at digital speeds while maintaining much of the analog accuracy required for integration verification.

These methodologies enable the simulation to maintain digital speeds, but they don't provide project-level information on the completeness of the verification. For that, teams should turn to the project-level, executable plans associated with the Cadence metric-driven verification (MDV) methodology and the associated Incisive Enterprise Manager. With that pairing, project teams can increase the performance, productivity, and quality of the overall project, which is critical for advanced node SoC complexity.

Summary

Performance for functional verification is undergoing the largest change since commercial simulators debuted a generation ago. Designers experience this change in the doubling of their work every 24 months, as Gordon Moore observed, but verification engineers experience exponential growth in the same time period.

In response, Cadence engineers are innovating performance improvements in the core simulator, forming close relationships with leading electronics companies, and delivering continuous improvements that benefit the entire use community. That helps the simulator keep pace with advanced node SoC performance needs, but the scale of the integration for these SoCs demands further innovation. As such, Cadence also provides structured approaches to verification—including new technologies and methodologies—that address bottlenecks unique to SoC verification, go beyond RTL simulation, and include virtual prototyping, high-level synthesis, mixed-language behavioral and timed simulation, hardware acceleration, emulation, and rapid prototyping solutions. Delivered by a worldwide expert field team, Cadence is scaling verification performance to meet your projects needs for today and tomorrow.

cadence®

Cadence is transforming the global electronics industry through a vision called EDA360. With an application-driven approach to design, our software, hardware, IP, and services help customers realize silicon, SoCs, and complete systems efficiently and profitably. www.cadence.com

© 2012 Cadence Design Systems, Inc. All rights reserved. Cadence, the Cadence logo, Conformal, Encounter, and Incisive are registered trademarks of Cadence Design Systems, Inc. All others are properties of their respective holders. 22551 01/12 MK/DM/PDF