IT’S ABOUT TIME: REQUIREMENTS FOR THE FUNCTIONAL VERIFICATION OF NANOMETER-SCALE ICS
OVERVIEW

Functional verification of nanometer-scale ICs is all about effective use of time, in terms of speed and efficiency. Yet today’s fragmented functional verification processes make it impossible to optimize either. Every design task has its own separate verification stage, resulting in numerous unique verification stages. Each stage has its own methodology, environment, software, languages, models, user interfaces, APIs and, often, even its own specialized verification engineers. Engineers create almost everything from scratch at every stage, leaving the preceding stage to rot. The result is an expensive, slow, inefficient process that all too often allows critical bugs to reach silicon. Techniques such as automatic test generation and assertions during RTL simulation, sometimes known as “smart verification,” apply to only a single verification stage and thus cannot even begin to address fragmentation.

This paper describes the requirements for high-speed, high-efficiency functional verification of nanometer-scale ICs. The paper first examines the primary verification drivers—massive digital logic, massive embedded software, and critical on-chip analog circuitry. It then describes a unified verification methodology—one that supports verification from system design to system design-in—across all design domains.

The unified methodology delivers unmatched speed and efficiency by eliminating fragmentation. It uses only proven technologies and techniques, and supports evolutionary migration from existing methodologies. The heart of the methodology is a functional virtual prototype (FVP), which begins as a transaction-level golden reference model of the design and verification environment, then serves as unifying vehicle throughout the verification process. The methodology also addresses the two critical paths in IC design-to-volume—embedded software development and system design-in.

The paper also describes the primary requirements for a verification platform that optimally supports the unified methodology. The first requirement is a heterogeneous single-kernel architecture with native support of Verilog®, VHDL, SystemC®, PSL/Sugar assertions, analog/mixed-signal (AMS), and algorithm development. This type of architecture is necessary to eliminate fragmentation. The second requirement is exceptional performance through full transaction-level support; high-speed, unified test generation; and hardware-based acceleration.

Functionally verifying nanometer-scale ICs requires incredible speed and efficiency. Utilizing a unified methodology based on a unified verification platform provides exactly that. It’s about time.

FUNCTIONAL VERIFICATION DRIVERS

The factors driving IC functional verification are the same as those behind IC design—increasingly complex digital logic, increasing embedded software content, and critical on-chip analog circuitry. Any unified methodology must directly address all three areas.

MASSIVE DIGITAL LOGIC

Exponentially increasing silicon capacity is the major force fueling the entire electronics industry, and functional verification is no exception. At 130 nanometers, designs can contain nearly 50M gates of digital logic, and that increases to 100M gates at 90 nanometers (see Figure 1). While design teams try to keep up with the exponential growth in complexity known as Moore’s Law, verification teams are chasing a curve that rises exponentially faster.
From a theoretical point of view, completely verifying a digital design requires checking the entire state-space, i.e., the entire set of possible internal states and transitions between them. The maximum state-space is roughly a function of the square of the number of registers in a design, which is growing exponentially. Even very simple designs by today’s standards have many thousands of registers, giving them a state-space so large that it is impossible to simulate it within one’s lifetime.

From a more pragmatic point of view, verification complexity is a function of the number and complexity of a design’s interfaces and their protocols. Again, the sequential combinations of potential interface activity are astronomical and not possible to fully simulate. Given this, the verification team’s responsibility is to cover as much of the most important behavior as possible within the target design cycle. In order to do so, they must maximize speed and efficiency.

MASSIVE EMBEDDED SOFTWARE

Increased silicon capacity enables and encourages design teams to integrate a wider variety of functions on the same chip, and most important, processors with their associated software and analog circuitry. These trends also make verification much more challenging.

SoCs are integrated circuits with one or more processors, memory, and application-specific logic. They include embedded software, from drivers and protocol stacks to operating systems and application software, all of which must work flawlessly with the hardware. The amount of software embedded in SoCs is growing rapidly. In fact, at 130 nanometers embedded software development costs equal the hardware development costs for SoCs, and by 90 nanometers fully 60% of all SoC development costs will be embedded software related. See Figure 2.
More important, embedded software has become the critical path in completing most SoC designs. Design teams must thoroughly verify the hardware-dependent software prior to silicon tapeout or risk re-spins. This hardware-software co-verification adds a new dimension to the already complex task of functional verification, and requires software and hardware teams to work closely together.

Given the complexity of today’s hardware and software, it is infeasible to verify their interaction at the register-transfer level. Verifying low-level, hardware-dependent software requires at least 100x faster execution. Verifying application-level hardware-software interactions generally requires very long sequential execution, making it impractical without emulation or prototyping. Moreover, SoCs increasingly contain more than one processor, such as a microprocessor with a digital signal processor (DSP), which makes software development and hardware-software co-verification all the more difficult.

CRITICAL ON-CHIP ANALOG

Digital logic and analog circuitry historically have resided on separate ICs, and their functions have been relatively independent. However there is a strong trend to integrate the two on the same piece of silicon, creating a new generation of ICs, the digital/mixed-signal (D/MS) ICs. These devices include relatively small, but critical analog logic that interacts tightly with the surrounding digital logic. This functionality is needed for high-performance wireline and wireless communication. Figure 3 shows the growth in the number of SoCs that have critical analog/mixed-signal content. According to industry analyst IBS Corporation, digital/mixed-signal SoCs accounted for approximately 20% of worldwide SoCs in 2001, and that percentage will rise to nearly 75% by 2006. In fact, more than half of all 90 nanometer designs will be D/MS ICs.

Not surprisingly, adding analog circuitry on ICs requires additional functional verification. This is especially crucial in designs that contain tight interaction between the digital logic and analog circuitry. Approximately 50% of all D/MS design re-spins are due to the analog portion, and many of those are functional errors. Additionally, IC design teams over-design mixed-signal interfaces to avoid functional problems, thereby sacrificing performance and area.

KEY ISSUES

With increasing design complexity, functional verification requires ever more elaborate verification environments that drive ever more vectors through the design. Many complex designs today are verified with billions of vectors before tapeout, yet as pointed out above, these cover only a small fraction of the potential state-space. Thus, not surprisingly there is no shortage of verification issues. Table 1 below contains an unprioritized list of some of the most critical verification issues that IC teams cite. These issues are described briefly in the appendix on page 21.
While each of these issues can be addressed separately, a fundamental issue underlies all of them—fragmentation. Moving to a unified methodology substantially addresses all of these issues, resulting in far superior speed and efficiency.

**TODAY’S FRAGMENTED VERIFICATION METHODOLOGY**

Today’s verification processes are not optimized for speed or efficiency. In fact a single word captures the essence of today’s verification methodologies—fragmented. *Figure 4* shows a typical verification “flow” for an SoC design. The verification stages are connected with dashed lines to indicate where the connections should reside. In fact little if any verification IP transfers between stages today.

**FRAGMENTATION WITHIN A PROJECT**

Each design task, from system modeling to prototyping, has its own verification stage to go along with it. Each verification stage has its own isolated environment including its own language support, software, testbench, models, user interface, and debug environment. These environments can take many staff-years to create. Because each environment is unique, teams reuse few models,
tests, monitors, or other forms of verification IP from one stage to the next; there is limited “vertical reuse.” Instead, they re-create the same information again and again throughout the project. The result is a slow, grossly inefficient overall methodology.

Once a design stage is complete, the verification environment is left unmaintained while the team creates a whole new verification environment for the next design task. Design changes in later verification stages are never incorporated into earlier completed verification stages. If and when the team needs to incorporate major late-stage changes, it can use only slow late-stage verification approaches because earlier verification stages no longer match the design. Moreover, this also makes it impractical to reuse the verification IP and methodology for subsequent derivative designs.

Verification methodologies have grown in an ad hoc manner, reacting to the expanding design methodologies. So it is not surprising that today’s verification process simply mirrors today’s multi-stage design process. Verification software vendors have followed suit, creating increasingly specialized software that are useful during one or a few verification stages. Thus the lack of a comprehensive verification platform has created an artificial barrier for verification teams to create a unified methodology. However, the fragmentation problem is much more far reaching than a given project. There is substantial fragmentation between projects within companies.

FRAGMENTATION BETWEEN PROJECTS

Few companies have anything like a common verification methodology across projects—even for derivative designs. While for years companies designing semiconductors have standardized on implementation flows, they have not given the same consideration to standardizing on verification methodologies.

IC design projects often use completely different verification methodologies. Only a small fraction of the differences are due to inherent design differences. Different projects often use completely different approaches with different software programs that have different limitations and require different types of modeling. The projects rely on different types of metrics and set up different types of environments. Thus while the design IP might be transportable between projects, the verification IP almost never is transportable. Even the verification engineers generally require significant ramp-up time, specifically to learn the new verification environment in order to transition from one project to another.

Even when the top-level methodologies are the same, as is the case with derivative designs, the details are invariably different enough to require a substantially new effort. Figure 5 illustrates the same top-level methodologies, with different environments for each stage. With more and more derivative designs, directly addressing this fragmentation is critical.

Figure 5: Fragmentation occurs even between derivative IC designs.
For large companies, fragmentation between projects is incredibly expensive. It creates small pools of tools and infrastructure that cannot be shared readily, including verification engineers who have expertise in only specific verification environments. Fragmentation between projects greatly increases the cost of integrating and maintaining verification environments. It results in vast redundancy in developing various forms of verification IP including modeling, monitors, and test suites. It even makes it difficult to evaluate the relative effectiveness of various approaches, identify best practices, and increase their use.

FRAGMENTATION ACROSS A DESIGN CHAIN

Fragmentation also causes major inefficiencies between design chain partners. Most of today’s IC designs include many IP blocks from third parties, each of which performed extensive verification. Unfortunately, the IP integrator is all too often left with little if any useful verification IP because of incompatible verification environments and approaches. At the other end of the design process, IC design teams provide little if any useful verification IP their system customers can use to integrate their IC.

Given that the majority of overall design time is spent in functional verification, these have become major problems. There are signs that this is changing. Leading system companies are beginning to demand complete functional verification environments from their IC suppliers, and leading IC providers are beginning to provide them.

UNIFIED VERIFICATION METHODOLOGY

This section describes a unified verification methodology based on proven technologies and techniques. The methodology directly addresses the problem of fragmentation by unifying verification from system design to system design-in, and across all design domains. In doing so it addresses today’s most critical verification issues. It also supports evolutionary migration from existing verification methodologies. While it may be impractical for many verification teams to move directly to a unified methodology from a highly fragmented methodology, this methodology provides an important standard that verification teams should work toward.

GUIDING PRINCIPLES: SPEED AND EFFICIENCY

The unified verification methodology is based on a set of basic guiding principles that maximize speed and efficiency. Verification speed refers to the raw throughput of stimulus, most conveniently measured in equivalent cycles per second. Treating the design and the verification environment as distinct, there are four key factors contributing to speed as shown in Table 2.

<table>
<thead>
<tr>
<th>#</th>
<th>Performance Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Design model abstraction</td>
</tr>
<tr>
<td>2</td>
<td>Environment model abstraction (including test generation)</td>
</tr>
<tr>
<td>3</td>
<td>Design verification engine performance</td>
</tr>
<tr>
<td>4</td>
<td>Environment verification engine performance (including test generation)</td>
</tr>
</tbody>
</table>

Table 2: Principles of verification speed

Each step up in abstraction—from gate to RTL to transaction to behavioral—can increase speed by one or more orders of magnitude simply because there is less data, less computation, and less frequent computation. On the verification engine front, hardware-based verification can provide 100x to an incredible 100,000x speed-up versus RTL simulation. The unified methodology leverages these principles by enabling verification teams to work at the highest possible level of abstraction and to migrate to hardware-based engines as quickly as possible.
The unified methodology is built upon the principles for verification efficiency in Table 3, where efficiency is defined as the return per unit of input.

<table>
<thead>
<tr>
<th>#</th>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Utilize an executable specification</td>
<td>Create, maintain, and use a golden executable specification throughout the process to remove all functional ambiguity</td>
</tr>
<tr>
<td>2</td>
<td>Optimize for the critical path</td>
<td>Optimize for the overall critical paths to volume SoC production — embedded software development and system design-in</td>
</tr>
<tr>
<td>3</td>
<td>Increase abstraction</td>
<td>Work at higher levels of abstraction to create verification IP that executes much faster and takes much less time to write</td>
</tr>
<tr>
<td>4</td>
<td>Create once</td>
<td>Avoid creating the same information multiple times or in multiple forms</td>
</tr>
<tr>
<td>5</td>
<td>Reuse often</td>
<td>Once verification IP of any form is available, use it whenever and wherever applicable</td>
</tr>
<tr>
<td>6</td>
<td>Increase observability</td>
<td>Use assertions on all internal interfaces and complex logic to catch more bugs in less time and to debug them faster</td>
</tr>
<tr>
<td>7</td>
<td>Measure to optimize</td>
<td>Measure coverage at the structural (i.e., logic and logic structure), transaction, and application levels in order to direct verification</td>
</tr>
<tr>
<td>8</td>
<td>Leverage everything</td>
<td>Eliminate all barriers to utilizing verification IP, software, infrastructure, and engineers at any stage of any project at any time</td>
</tr>
</tbody>
</table>

Table 3: Principles of verification efficiency

THE FUNCTIONAL VIRTUAL PROTOTYPE—THE UNIFYING VEHICLE

The unified verification methodology is a top-down methodology specifically designed for complex ICs in which architectural performance analysis, embedded software development, system design-in, or any combination of these is critical. All of these cases greatly profit by having an early, fast, accurate design representation. Properly created, the representation can serve as a unifying vehicle throughout the rest of the verification process. Such a representation—the functional virtual prototype (FVP)—is the centerpiece of the unified verification methodology.

Unified methodologies are less advantageous for less complex designs, such as all digital designs, simply because such designs have less fragmentation and less need for an early design representation. However, the vast majority of nanometer-scale ICs will be complex SoCs for which a unified methodology is critical. Moreover, this approach is also valuable for any processor-based system — systems within which most all digital designs, for example, must eventually be verified.

FVP overview

The FVP is a golden representation of the entire design and verification environment (see Figure 6). It encompasses all aspects of the design including embedded software, control logic, datapath, and analog/mixed-signal/RF circuitry.

Figure 6: Functional virtual prototype (FVP)
The initial FVP uses a transaction level of abstraction for all design models. Creating transaction-level models takes a fraction of the time it takes to create the equivalent RTL model, and transaction-level models run approximately 100x faster than equivalent RTL. IC teams may tradeoff top-level architectural detail and accuracy to reduce the FVP development and maintenance effort; however, the FVP partitioning should match the partitioning of the intended implementation. All stimulus generator, response generator, and application checkers also are written at a transaction-level in which creation time and run times are very fast.

The critical role of the transaction-level FVP

A transaction-level FVP should run approximately 100x faster than the RTL equivalent. This speed provides a number of distinct advantages, making it important to have transaction-level models for all top-level blocks, including previously implemented blocks. The transaction-level FVP serves a number of critical roles throughout the design and verification process, most notably those listed in Table 4.

<table>
<thead>
<tr>
<th>#</th>
<th>FVP role</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>An unambiguous executable specification</td>
<td>Eliminates the ambiguity and out-of-date nature of document-based specifications</td>
</tr>
<tr>
<td>2</td>
<td>An accurate model for architectural performance analysis</td>
<td>Enables micro Architects to evaluate and optimize the overall architecture’s performance</td>
</tr>
<tr>
<td>3</td>
<td>A fast executable model for early embedded software development</td>
<td>At 100x faster than RTL, enables verification of hardware-dependent software such as drivers and protocol stacks many months earlier</td>
</tr>
<tr>
<td>4</td>
<td>An early handoff vehicle to system integration teams</td>
<td>Enables system integration teams to evaluate, direct IC architecture optimization, and begin functional design-in</td>
</tr>
<tr>
<td>5</td>
<td>The source for block-level golden reference models</td>
<td>Eliminates the need for block-level verification teams to create a reference model or embedded checks in the tests</td>
</tr>
<tr>
<td>6</td>
<td>The full-chip verification vehicle for blocks</td>
<td>Provides block-level teams a high-speed, full-chip verification environment with a full test suite for in-context verification prior to integration</td>
</tr>
<tr>
<td>7</td>
<td>The contract between the top-level and block-level teams</td>
<td>Ensures block-level implementation teams know what they need to deliver and top-level integration teams know what they will receive</td>
</tr>
<tr>
<td>8</td>
<td>The verification integration vehicle</td>
<td>Provides a ready, high-speed vehicle for integrating block-level implementations into the top-level design</td>
</tr>
<tr>
<td>9</td>
<td>The reference for transaction coverage requirements</td>
<td>Through interface monitor coverage, implicitly defines the top-level transaction coverage requirements</td>
</tr>
</tbody>
</table>

Table 4: Transaction-level FVP roles

**METHODOLOGY OVERVIEW**

The unified methodology begins with creating and verifying a transaction-level FVP. Once complete, the FVP is an executable specification that replaces the paper specification, and embedded software development and system integration teams begin using it to accelerate their design process.
Next, while designers are implementing and verifying their individual units, verification engineers create block-level test environments using the transaction-level FVP block models as reference models and extending the FVP interface monitors to the signal level for that block. Verification engineers perform functional directed testing to bring up the block, then use a combination of extended testing in the block-level test environment and verification in the original transaction-level FVP environment to meet the required transaction and structural coverage. Structural coverage, to be clear, is implementation-specific coverage of the logic and important logic structure functions, e.g., determining that FIFOs are filled.

As soon as a set of blocks that provides meaningful top-level functionality is verified, the verification team populates the FVP with those blocks, using the original transaction-level models for the rest of the design. The team verifies these blocks together, adding each remaining verified block as it becomes available. When verification becomes performance- and capacity-limited, the team uses acceleration on the more thoroughly verified blocks. When the FVP is fully populated with the block-level implementations, the verification team focuses on completing the application and transaction coverage requirements. At this point the implementation-level FVP is ready for final software integration and final system design-in, generally using emulation or prototypes.

Once a verification team has established the FVP and unified methodology for a given design, it can easily re-apply them for derivative designs to again reap the benefits of speed and efficiency.

**TRANSACTION-LEVEL FVP CREATION AND VERIFICATION**

**Definition**

An FVP begins as a fully transaction-level model, where transactions are defined as single-cycle or multiple-cycle autonomous interface events in which signal-level activity is abstracted to operations, such as a bus read or a bus write. For many applications it is beneficial to define hierarchical transactions, especially at the external interfaces where top-level transactions represent application functions such as instruction commands, packets, or data streams.
Having interface monitors on all key external and internal interfaces is critical. These monitors are sets of assertions that check the transaction-level protocol for each block’s interfaces. They track interface specific transaction coverage, including transaction sequences and combinations, and they form the framework for the signal-level interface monitors used in implementation testing.

The remaining verification environment—stimulus generator, response generator, and application checks—are also at the transaction level. The stimulus generator supports directed, directed-random (constraint-based random), and random test generation. The response checker provides appropriate application-level responses. The application checker ideally would be a full behavioral model for the IC. However, as that is often impractical, it may be a collection of application-level checks.

Creation

Verification teams should create the FVP with the help of the design’s micro-architects. Doing so facilitates knowledge transfer from the micro-architects to the verification team. Once created, the micro-architects have a golden executable specification which they can use for transaction-level performance analysis, and the verification team is far ahead of where it would be otherwise. The verification team is intimately familiar with the design requirements and has the specification in an unambiguous executable form, a top-level verification environment, and a set of reference models for block-level verification.

It is important to create transaction-level models for pre-existing IP that appears at the top level. RTL models will degrade performance to the point where it eliminates many of the transaction-level FVP benefits. The models can be a behavioral model with a transaction-level shell and need be only as detailed as necessary to support overall FVP utilization (see Figure 8).

![Figure 8: Transaction-level modeling of existing IP](image-url)

It may be more efficient to develop some blocks, such as those for signal processing and analog/mixed-signal/RF circuitry, in specialized design and verification environments. The transaction-level FVP models for such blocks should contain behavioral cores with transaction-level shells to interface to the rest of the FVP. The team verifies these behavioral blocks as part of the transaction-level FVP.

Verification

Of course the team must verify the FVP itself. If a behavioral model exists for the entire IC, teams should obviously use it to verify the FVP functionality. Often there is no such behavioral model, so it is easiest to use traditional verification methods such as checkpointing, write/read-back sequences, and loop-back models. In fact, verifying the FVP is very similar to the lab bring-up of a final system, making similar methodologies appropriate. Creating tests at this early stage accelerates later implementation-level block verification in the FVP. The verification team should define initial application and transaction coverage requirements for the transaction-level FVP, then create the tests necessary to achieve this coverage prior to signing off on the executable model.
**BLOCK-LEVEL VERIFICATION**

Designers begin unit-level implementation while the verification team verifies the transaction-level FVP. The digital designers should use “structural assertions” to specify the intended behavior for all complex logic structures such as—finite state machines, FIFOs, stacks, and low-level interfaces between units. These assertions will detect bugs that might otherwise be missed, shorten debug by detecting bugs much nearer their source, and serve as important structural coverage monitors.

Below the block-level, design and verification teams often will choose to use bottom-up verification beginning at the individual unit level. Digital designers generally verify their own units. Most use an HDL as the verification language, although a growing number also use hardware verification languages (HVLs). Superior controllability makes deep verification easiest at this level. All designers should meet structural coverage criteria, including code coverage, for their units.

Designers creating units in specialized environments, such as signal processing and analog/mixed-signal/RF, will perform implementation and verification within those environments. They will be able to use their block-level behavioral core from the transaction-level FVP as a starting point for implementation and as a reference model for implementation verification.

The block-level environment

While designers are completing and verifying their units, the verification team creates block-level test environments as shown in Figure 9.

---

**Figure 9: Block-level verification environment**

The stimulus generator generates transaction-level test sequences. All tests are at the transaction-level where they are fast and easy to write, debug, maintain, and reuse. Transaction-level tests also enable directed random testing, on verification using randomly generated inputs that are intelligently constrained to stress a certain area of logic.
The master and slave transactors are transaction-to-signal and signal-to-transaction converters that bridge between transaction-level traffic generation and the block’s signal-level interface. The interface monitors check all operations on the bus, translate data into transactions, record transactions, and report transactions to the response checker. These monitors consist of the transaction interface monitors from the FVP with signal-to-transaction converters to interpret the signal-level bus activity.

The response checker contains the block model from transaction-level FVP and a compare function for self checking. Reusing the block model eliminates the redundant, error-prone, and high-maintenance practice of embedding checks in tests.

Hardening blocks in the block-level environment

The goal of block-level testing is functionally accurate blocks with no bugs. Thorough verification of the transaction-level FVP and the individual blocks leaves the top-level implementation verification to test block-to-block interconnects and the target application.

The most efficient way to bring up blocks is to start with directed tests in the block-level environment. One function at a time is verified first, then increasingly complex combinations of functions are verified. Debug time is critical at this phase. Thus, while it is possible to generate higher coverage in less time with random or directed random techniques early on, the additional debug time needed to decode the stimulus makes doing so much less efficient. Basic directed tests, unlike random tests, also provide a solid baseline of regression tests which are fairly insensitive to design changes. Once all basic functions are verified, it is time to “harden” the unit through extensive testing that meets all of the transaction and structural coverage requirements. The verification team accomplishes this through a combination of block-level stress testing and realistic top-level testing in the transaction-level FVP.

The verification team uses random, directed random, and directed tests in the block-level environment to stress-test the block. Random and directed random pick up most of the remaining coverage holes, but the team is likely to have to create directed tests to verify the difficult-to-reach corner cases. Verification teams can clearly trade off running more random or directed-random cycles versus handcrafting directed tests. Using simulation farms or an accelerator, teams can run orders-of-magnitude more random or directed random cycles to hit more of the corner cases, as well as cover more of the obscure state-space beyond the specified coverage metrics. If engineer time is at a premium—as it usually is—this can be an excellent tradeoff. It should be noted that formal and semi-formal techniques may also be useful to verify properties or conditions that are extremely difficult to simulate or even predict.

Verifying blocks in context

The transaction-level FVP environment includes the tests that verified the original transaction-level FVP. The verification team can take advantage of these tests by integrating the block implementation, the signal-level interface monitors, transactors, and structural assertions into the transaction-level FVP. Doing so tests the block in the context of transaction-level models for the surrounding blocks, thus directly testing its interfaces to those blocks. Since the rest of the FVP is at a transaction level, simulation is very fast, and it is also possible to use any available embedded software as tests. While this provides tremendous benefits for little incremental work, this environment does not have the necessary controllability to stress test the block.
When a core set of “hardened” blocks is available, such as a set that provides meaningful top-level functionality, the verification team integrates those blocks into the original transaction-level FVP. The FVP contains interface monitors from the transaction-level FVP and the structural assertions within the logic in the blocks. As it did during block-level bring-up, the verification team begins by exercising one function at a time, then verifying increasingly complex combinations of functions. This is the first verification of signal-level interfaces between blocks; prior to this all interface interactions were at a transaction level. Once the blocks are functioning, the verification team can run the complete set of top-level tests—including any available from the software team. It should also run top-level random and directed-random testing.

As each new block becomes available, the team brings it up in the same controlled manner to facilitate easy debug and to create a clean baseline set of top-level regression tests. Top-level verification quickly becomes performance and capacity limited, so it is best to migrate blocks that are running clean into an accelerator. Doing so leaves only a small number of blocks running in RTL simulation; the rest of the implemented blocks run in an accelerator and the non-implemented blocks run at a transaction-level (see Figure 11). Thus, this approach maximizes overall speed.

Figure 10: Verifying block in context of the original FVP

THE IMPLEMENTATION-LEVEL FVP

Figure 11: Integrated new block into implementation-level FVP
FULL SYSTEM VERIFICATION

When the FVP is fully populated with the block-level implementations, the verification team focuses on application coverage and any remaining transaction coverage requirements. Application coverage typically requires long, realistic test sequences such as transferring packets, booting an operating system, rendering images, or processing audio signals. Raw speed is essential. Complex ICs with long latency operations require emulation or prototyping to deliver near-real-time performance.

Emulation and prototyping

Full system verification generally requires one or more of the following:

- Extensive application-level software verification, including the RTOS
- Extensive verification of the full implementation
- Verification in the context of the rest of the system, such as other ICs
- Verification in real-time or near-real time for human interface evaluation
- Verification with real world stimulus or realistic test equipment

Full system verification requires either emulation or prototyping (generally via FPGAs and bonded-out processor cores). In the case of emulation, the verification team loads the implementation-level FVP with the structural assertions into the emulator. The process is straightforward if the team already has accelerated the design. In fact, in some cases it may make sense to combine simulation, acceleration, and emulation, such as when part of the design or environment is available only in software and part is available only in hardware. Prototyping is a straightforward process if all of the new logic fits into a single FPGA or easily partitions into a small number of FPGAs. Otherwise, it can be an onerous manual process.

Emulation and prototyping enable teams to test the design at near real-time speed in the context of its actual physical environment. This enables verification of application software against the design. Generally design teams emulate to test application software, increase application-level coverage, or produce design output for human sensory review—such as video and audio.

Verification hub: the design chain connection

In almost all cases today, system integration teams do not start their design-in in earnest until they get prototype boards or silicon from their semiconductor supplier—whether it’s an internal or external group. Both companies lose huge value in terms of time-to-market. With the unified methodology, as described above, embedded software developers and system integration teams can begin development based on the fast transaction-level FVP. While this has great advantages, it is limited by the same factors itemized in the preceding section. However, once the verification team has an acceleration, emulation, or prototype FVP, it can provide access to this much higher-performance, fully implemented model to customers via a verification hub.

Figure 12: Verification hub
Verification hubs enable the semiconductor company to maintain full control and possession of the detailed implementation and give their customers access to high-speed design-in, verification, and software development environments. They also provide high-quality, realistic application-level testing from actual customers prior to silicon which could be valuable to the SoC design team, internal embedded software developers, applications engineers, and others involved in technical IC deployment.

**UNIFIED VERIFICATION PLATFORM REQUIREMENTS**

Unified verification methodologies such as the one described above require a unified verification platform. The platform should be based on a heterogeneous single-kernel simulation architecture in order to optimize speed and efficiency. While a step in the right direction, integrating different verification engines—simulators, test generators, semi-formal tools, and hardware accelerators—and their environments, is no longer good enough. Without a single-kernel implementation, critical performance and capacity is lost, ambiguity is introduced and, most important, fragmentation remains.

The architecture must be heterogeneous, supporting all design domains—embedded software, control, datapath, and analog/mixed-signal/RF circuitry—and supporting everything from system design to system design-in. It must include a comprehensive set of high-performance verification engines and analysis capabilities. Lastly, the architecture must have a common user interface, common test generation, common debug environment, common models, common APIs, and must support all standard design and verification languages.

Only a verification platform meeting these requirements will support a unified verification methodology such as that described in the preceding section. By doing so, it will dramatically increase verification speed and efficiency within projects, across projects, and even across design chains.

**SINGLE-KERNEL ARCHITECTURE**

The single-kernel architecture must be heterogeneous with native support for all standard design and verification languages.

**Native Verilog and VHDL**

For digital design and verification the single-kernel architecture must natively support Verilog and VHDL. The number of mixed-language designs grows as IC teams use more and more IP in their designs, making a single-kernel increasingly important. Most designers will continue using HDLs for specifying their designs and performing unit-level verification for the foreseeable future. Verification engineers with strong RTL backgrounds may also prefer using these languages, especially for any block-level and unit-level work.

The platform should also include support for standardized verification extensions to these languages if and when they become viable. While adding verification extensions to current HDLs will give designers more powerful verification capabilities, they will have limited applicability at the top-level, or even the block-level, for SoCs with significant software content. With the vast majority of embedded software being written in C/C++, standard C/C++-based languages will be a requirement.

**Native SystemC**

The architecture must also provide native SystemC support. SystemC is a relatively new C/C++-based industry standard language that supports highly efficient transaction-level modeling, RTL specification, and verification extensions for test generation. SystemC is implemented as an open-source C++ library and the Open SystemC Initiative® (OSCI) provides a free SystemC reference simulator. These characteristics make it an ideal language for a transaction-level FVP, especially when the design includes significant embedded software.
Leading system, semiconductor, and IP companies are rapidly adopting SystemC both to natively support embedded software and hardware modeling in the same environment and to move off proprietary in-house C/C++ verification environments. This makes it likely that SystemC will become the standard exchange language for IC designs and IP. However, it is unlikely that SystemC will displace HDLs for unit-level design and verification in the foreseeable future; hardware designers strongly prefer HDLs and only the HDLs have a rich design support infrastructure and environment in place.

Native Verilog-AMS and VHDL-AMS

Since most nanometer-scale ICs will include analog/mixed-signal/RF blocks, the verification environment must include native kernel support for Verilog-AMS and VHDL-AMS. Having a single-kernel that combines logic simulation with analog/RF simulation ensures high-performance, high-accuracy simulation where it is needed most—in complex mixed-signal interfaces. A single-kernel implementation enables true top-down, analog/mixed-signal/RF design based on high-speed behavioral models verified at a transaction-level in the functional virtual prototype long before committing to actual silicon. It also supports verifying traditional bottom-up designs in the context of the implementation-level FVP.

Native PSL/Sugar

The architecture also needs native support for the new industry standard assertion language, the Accellera Property Specification Language (PSL) which is based on the IBM Sugar assertion language. Sugar is a powerful, concise language for assertion specification and complex modeling. A single-kernel implementation ensures the lowest possible overhead by providing maximum possible performance with minimum increase in verification capacity. It also enables a single-compilation process which guarantees that simulation always contains the right assertions for the design under verification. PSL/Sugar works with Verilog and VHDL only at this time. Verification teams creating a transaction-level FVP in any other language can include interface monitors written in PSL/Sugar by adding Verilog or VHDL modules with the necessary PSL/Sugar code.

PERFORMANCE, PERFORMANCE, PERFORMANCE

Needless to say, performance is more critical than ever at the nanometer level. As described above, speed depends on design abstraction, environment abstraction, design performance, and environment performance. The unified verification platform must maximize all of these dimensions.

Transaction-level support

With the 100x performance advantage transaction-level modeling has over RTL, verification teams need to use this level of abstraction whenever and wherever possible. In order for verification teams to take full advantage, their verification platform must fully support transaction-level abstraction. In many cases, verification teams will want to use SystemC for their transaction-level FVP, making native SystemC mandatory. Interface monitors must track transactions and ensure that they are legal, requiring native PSL/Sugar support as well as transaction recording.

Transaction recording in particular should be easy to retrofit to the extensive set of existing IP that recognizes transactions. The platform must support transaction recording throughout the entire verification process, and it must be able to aggregate results from all verification runs including those that the embedded software development team runs. Transaction visualization and transaction-level exploration is critical to assessing the design performance, understanding its actual functionality, and debugging errors. From a debug perspective, the platform needs to support transaction analysis, for example the ability to identify specific transaction combinations, and transaction debug, such as within the waveform viewer.
High-speed, unified test generation

At the block-level and top-level of the design, testbench performance is often just as important as the performance of the design itself. Test generation must be fast and the test environment must include a rich set of capabilities for constraint-based generation, monitoring, and analysis. Verification teams using C/C++ test generation environments have enjoyed high performance, but at the expense of having to develop and maintain a proprietary system. Verification teams using HDL also have enjoyed high performance, but at the expense of working within a hardware-only environment with a limited feature set.

In recent years, many verification teams have moved to commercially available proprietary hardware verification languages (HVLS) that mix hardware with software constructs to provide a rich set of verification-centric capabilities. These systems have distinct advantages versus standard C/C++ and standard HDLs. However they can be slow and expensive. Being proprietary, the environments are not readily transportable and often support only a small part of the overall verification process—generally simulation-based digital verification. Standardizing proprietary HVLS will certainly make them more attractive. However, many current HVL users have identified significant performance issues, especially at the top level, and are seeking alternatives.

For engineers interested in using C/C++-based verification environments, the SystemC verification standard provides an excellent alternative. It is very high-performance and has a rich set of standard capabilities in the same open source C++ language that supports system modeling and even RTL specification. Just as important, it is a standard, ensuring test environments are transportable and any number of tool vendors can compete to create the best products. For designers and verification engineers interested in working in an HDL-oriented environment, new open, standard extensions to Verilog may provide a strong alternative. Unified verification platforms must support the SystemC verification standard, as well as new HDL-based language extensions as they become viable.

Hardware-based acceleration

The single-kernel simulation engines must be fast, and continue getting faster. However, hardware acceleration and emulation provide typically two to four orders-of-magnitude higher performance. Hardware acceleration is required to keep top-level performance from grinding down as the team brings up the implementation-level FVP (see Figure 13). Acceleration has a very wide performance range depending both on how much of the design and verification environment is accelerated and on the simulator-to-accelerator interface which should be kept at a transaction level if possible.

Historically hardware-based systems have been prohibitively expensive for most verification teams. Moreover, it has taken weeks to months to get a design into the hardware-based system. Unified verification platforms must provide hardware-based acceleration that addresses both of these issues.
Hardware-based systems have inherent design and manufacturing costs. To effectively reduce these costs, the platform should include the ability to simultaneously share systems among a number of users. This capability can provide many verification engineers orders of magnitude performance advantages simultaneously at a fraction of the cost. In addition, the hardware accelerator should double as a multiuser in-circuit emulator to be valuable throughout the design cycle. In fact, in either mode the accelerator/emulator should have the ability to serve as a multiuser verification hub for internal or external software development and system integration teams.

Reducing time-to-acceleration is just as critical. Using the unified methodology can help, while also increasing the overall performance. For example, verification teams that know they are going to use acceleration should create their transactors so that the communication between the simulator and the accelerator will be at a transaction level rather than a signal level. From a platform standpoint, early design policy checking and co-simulation with a common debugging environment is a good first step; however, ultimately hardware acceleration needs to be an extension of the single-kernel architecture.

**EVOLUTIONARY MIGRATION**

When considering the ideal characteristics of a unified methodology it is critically important to recognize the realities of migrating to such a methodology. Migration must be evolutionary, not revolutionary. To the extent that existing resources are optimized, they are optimized for an existing fragmented flow and for specific stages of verification, as is the existing infrastructure. Verification engineers are certain to lack some skills necessary for the new methodology. Perhaps most important is the existing intellectual property (IP) in all of its various forms—specifically models and legacy tests.

A unified verification platform must support all third-party models and other verification IP written in standard languages. Native standard language support ensures high performance. Also important, it should be easy to incrementally update existing IP to take advantage of key unified methodology requirements, in particular transaction-level recording and assertions. Much existing IP is written to operate on transactions and/or to flag protocol errors. The platform should make it very easy to incrementally update the IP to record this information in its standard transaction and assertion databases. Similarly, it should make it easy to incrementally add reusable verification IP compatible with the unified methodology. On the test side, the platform must also provide support for any existing in-house or third-party test generation tools and related tests. This should be accommodated via production-proven links via standard APIs wherever possible.

Of course a unified platform should enable third-party and in-house tool integration. In large part this is accommodated via standard languages and APIs. However, the platform should provide third-party and in-house tools the ability to interact with any databases and via the standard user interface. Also, having a powerful ability to program coding style checks can help engineers migrate to a new methodology, as well as enforcing good design and verification coding policy.

**CONCLUSIONS**

Functional verification of complex ICs is in a disastrous state. Exponential increases in digital logic and embedded software, along with critical mixed-signal circuitry will only exacerbate today’s issues. While verification teams and EDA companies have focused on the symptoms—performance, capacity, languages, coverage, advanced techniques, hardware-software co-verification, etc.—fragmentation is the root cause. By simply mirroring today’s fragmented design methodologies, verification teams have wound up with a slow, inefficient overall process. It is time to fundamentally rethink this approach.
The unified methodology described in this paper eliminates the root cause. Utilizing a transaction-level functional virtual prototype, verification teams create a high-speed executable model useful for embedded software development and initial design—many months before the implementation is complete. The FVP provides reference models for block-level testing, serves as a top-level verification environment for each block, and becomes the integration vehicle for the final implementation—even during hardware acceleration. Verification engineers also use the FVP to define application-level and transaction-level coverage, and use its interface monitors to track progress throughout the process. When ready, the team migrates the design into emulation or prototyping for near real-time application-level verification in a realistic environment.

This unified methodology requires a unified verification platform. For efficiency, the platform must be based on a heterogeneous, single-kernel architecture that natively supports Verilog, VHDL, SystemC, Verilog-AMS, VHDL-AMS, PSL/Sugar, and all other viable standard languages. Otherwise, the methodology will remain fragmented. The platform also should support transaction-level abstraction, hardware-based acceleration, and high-speed test generation for maximum performance.

A unified verification platform can provide the incredible speed and efficiency needed to verify nanometer-scale ICs. It’s about time.
APPENDIX: CRITICAL FUNCTIONAL VERIFICATION ISSUES

The table below lists some of the most critical verification issues cited by SoC teams.

<table>
<thead>
<tr>
<th>#</th>
<th>Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Performance</td>
</tr>
<tr>
<td>2</td>
<td>Capacity</td>
</tr>
<tr>
<td>3</td>
<td>System, verification, and design languages</td>
</tr>
<tr>
<td>4</td>
<td>Testbench development</td>
</tr>
<tr>
<td>5</td>
<td>Coverage</td>
</tr>
<tr>
<td>6</td>
<td>Models and verification IP</td>
</tr>
<tr>
<td>7</td>
<td>Reuse</td>
</tr>
<tr>
<td>8</td>
<td>Use of advanced verification techniques</td>
</tr>
<tr>
<td>9</td>
<td>Hardware/software co-verification</td>
</tr>
<tr>
<td>10</td>
<td>Analog/mixed-signal verification</td>
</tr>
</tbody>
</table>

PERFORMANCE AND CAPACITY

Performance and capacity are perennial issues dating back to gate-level simulation. Despite substantial improvements in simulation engine performance and exponential increases in workstation performance, software-based verification is slowly losing ground to the growing complexity of IC designs — especially with increasing time-to-market pressures. Many nanometer ICs will require hardware-assisted verification in order to be verified thoroughly enough in a timeframe of relevance to decide it’s time for a >$1M tapeout.

LANGUAGE SELECTION

Language selection is becoming a critical issue. Life was simple when design and verification both used Verilog or VHDL. With nanometer ICs, there are new and newly extended languages for system design, test generation, assertions and modeling, analog/mixed-signal design, and digital design. Language decisions are so critical because they typically have very long-term effects. Once system, verification, or design IP is created in a given language, it becomes increasingly difficult to switch directions. Proprietary languages, including those controlled by a single company, lock users into a particular vendor and its limitations. Since proprietary languages rarely garner ubiquitous support, it also often means incompatibilities with partners and customers. If universally supported standard languages are available, they are clearly the best alternative.

TESTBENCH DEVELOPMENT, COVERAGE, MODELS, VERIFICATION IP, AND REUSE

The next four issues—testbench development, coverage, models and verification IP, and reuse—all have to do with efficiently creating effective test environments. Verification teams want to be able to create directed, directed-random, and random tests as easily as possible. Doing so requires having a rich set of models and verification IP as a starting point. When one has to create a new model or verification IP, they ideally want to do so in a manner such that they can reuse it throughout the project, and ideally on other projects. Lastly, since verification is an exercise in risk management, teams have a keen interest in coverage capabilities that identify the need for specific additional tests, as well as indicate the comprehensiveness of the verification to date.
ADVANCED VERIFICATION TECHNIQUES

Advanced verification techniques includes the use of increased abstraction, assertions, formal, and semi-formal techniques to verify designs more completely and in less time. Working at a higher level of abstraction requires much less specification and the resultant models run much faster because they carry much less detail. Assertions enable designers and verification engineers to specify design intent in a form that increases observability for error detection and coverage analysis, as well as serve as inputs for formal and semi-formal verification techniques. Formal techniques include modeling checking and static formal analysis that ensure various assertions hold true throughout the entire state-space. Again the practicality of this has been limited to relatively small units within designs. Semi-formal verification attempts to relax these limitations by using formal techniques to expand the state-space explored through select simulations.

HARDWARE-SOFTWARE AND ANALOG/MIXED-SIGNAL VERIFICATION

The last two issues listed are a function of design convergence at the nanometer scale. With embedded software engineering costs surpassing the hardware development costs for nanometer ICs, it is not surprising that hardware/software co-verification is becoming increasingly important. Likewise with analog invading what would otherwise have been pure digital ICs. With today's high-speed communications, verifying the complex interactions between digital and analog interfaces is critical to ensuring first silicon success.