

Functional Verification: *A Best-of-Breed Assessment*

Charles Moore

June 18, 2002

Table of Contents

1 DEFINITIONS AND SCOPE OF THIS ASSESSMENT..... 3

2 BEST-OF-BREED FUNCTIONAL VERIFICATION TECHNOLOGY..... 5

 2.1 VERIFICATION PHILOSOPHY..... 5

 2.2 VERIFICATION TECHNIQUES AND METHODOLOGY..... 7

 2.2.1 *Optimize the Methodology for Simulation Efficiency* 7

 2.2.2 *Early Development of a Comprehensive Verification Plan* 8

 2.2.3 *Hierarchical Attack* 8

 2.2.4 *Coverage Tracking* 14

 2.2.5 *Discipline, Control and Management*..... 15

 2.2.6 *Design for Verification (DFV)*..... 15

3 CONCLUSIONS 16

Revision History

Revision	Date	Author	Comment
0.1	14-Nov-01	Chuck Moore	First Draft (created while at Parthus)
0.4	18-Jun-02	Chuck Moore	Expanded sections, mode more general

1 Definitions and Scope of this Assessment

The term *verification* is used very broadly in IP development and chip design. In this broad context, each stage of the development process has an element of verification associated with it. Each effort is critical to the success of the development process as a whole, and to the quality of the final product. In reality though, the broad use of the word *verification* is an oversimplification, and each of these phases target very different goals and require very different methods.

In an effort to help clarify some of these verification-related efforts, the following definitions are offered to help factor out the different aspects:

Functional Verification is the process of verifying that a design operates correctly as described in the architectural and functional specifications. It targets the *logical design* of the product, and typically assumes that the *physical design* process is unrelated. As described further in this document, a good functional verification plan attacks potential problems at several levels of hierarchy and abstraction.

Equivalence Checking is the process that ensures that a design is logically equivalent between the various representations of the design. For example, equivalence checking is used to make sure that an RTL representation is logically equivalent to a gate level representation that may have come from automated synthesis tools. Another good example is back-end LVS checking, where the actual shapes layout of a circuit (actually, the *extracted logical function* from the shapes) is checked against the logic embodied in the schematics of that circuit.

It is important to note that it is not possible to do exhaustive equivalence checking of the RTL functionality versus the architectural specification (because, in reality, the specifications are neither formal, nor are they complete). Moreover, while formal verification tools can handle some smaller designs, they cannot handle moderately sized design. This increases the need for special methods to achieve appropriate functional verification.

Timing Analysis is the process of checking that a design meets the required timing constraints associated with the clocking structure and frequency goals of the product. *Dynamic timing analysis* attempts to perform the timing checks while concurrently simulating a set of functional vectors in a timing-driven simulator. *Static timing analysis* attempts to check that all possible paths in the cone of logic leading to a timing checkpoint meet the required goals. It is important to note that while dynamic timing analysis offers some functional coverage (as defined by the pattern set run during the analysis), static timing analysis does not offer any functional coverage.

Power Analysis is the process of checking that a design meets the specified power related goals. This breaks down further into the analysis of power delivery mechanisms (can current be supplied at the rate demanded by the device), the analysis of power consumption (how much power is consumed during operation), and the analysis of the thermal implications of power consumption. A poorly designed chip-level power system may not only violate power-related specifications, but it can also result in functional anomalies due to voltage droops and noise spikes.

It should be noted that power management functions are becoming an increasingly common and important function in many designs. In cases where specific state or event-based information is used, in a logical sense, to engage clock gating or special state machines, new verification challenges are surfaced. To the degree this involves logical operations on state information, the verification of these functions should be included in the topic of functional verification.

Design Rules Checking is the process of checking that a physical implementation of a circuit or interconnect between circuits meets the physical design rules of a particular semiconductor technology. In general, it assumes that the intended logical function is already embodied in the design, and only works to check that the physical design meets the physical constraints associated with the specific technology. It is a fundamental step in the back-end chip development flow.

Testability Checking is the process that checks a design's compliance with certain test related design rules (typically, to enable automated test vector generation). The resulting test vectors are intended to enable hardware testing (on expensive chip-level or package-level testers) that check whether or not the manufactured chip has physical defects that affect the intended functionality. Again, this type of testing assumes that the intended logical function is already embodied in the design, and only provides checking to find out if the manufactured chip matches the functionality specified in the gate-level design.

Hardware Validation refers to the overall process of checking that a completed chip design matches the models (and derived specifications) used during the design process. Typically, hardware validation (also called *silicon evaluation*) focuses on the electrical and timing aspects of the chip, but also includes making sure that key software functions (OS, drivers, key applications) can be demonstrated. It carries value in the functional verification effort, but typically targets only the baseline functionality (ie: *not a systematic attack* on the functionality).

This report is focused on *Functional Verification*. As technology has offered increasing chip integration opportunities, the associated complexity of the designs has also increased dramatically.

It is important to realize that the relationship between complexity and functional verification is not linear, and in fact, is **exponential** (as a quick reminder of the nature of exponential mathematics, recall that in a system with 100 state values (latches), there would be $2^{100} = 1,000,000,000,000,000,000,000,000,000$ possible cases to be considered). Many modern designs easily contain tens of thousands of latches!

To deal with this exponential growth, new methods are fundamentally required.

2 Best-of-Breed Functional Verification Technology

The issues, concerns and complexities of functional verification are not new. They are, on the other hand, growing at a rate that has caused the electronics industry to invent new ways of dealing with them. We have the opportunity to leverage an enormous amount of valuable work that has been done in other segments of the industry (microprocessor design, for example). This section attempts to describe some of the best-of-breed thinking and methods for functional verification.

2.1 Verification Philosophy

Best-of-breed verification results come from a fundamental passion about the challenges and realities of today's designs. This passion gives rise to a set of *verification philosophies* that must thread into the fabric of the entire design team. The following statements of philosophy represent common themes demonstrated by the most successful teams in the industry. They form the skeleton upon which all other decisions about verification (methodology, tools, resources, scope, etc) are based.

- ***The cost of finding functional problems increases, in dramatic steps, as the design and development proceeds.***

Problems found after RTL freeze cause rework on all back-end efforts. Problems found after tape-out cause new masks and new manufacturing. Problems found in the field can cause recalls and/or broad deployment of patches. In each of these three scenarios, the cost of managing late-breaking problems increases by an order of magnitude. Appropriate investment in verification technology is very cost effective.

>> *Optimize the verification process to find bugs as early as possible.*

- ***A systematic and hierarchical approach is required to find the most difficult (and costly) bugs.***

Verification of basic functionality is only a starting point. The difficult bugs are buried beneath the basic functionality. They require extraordinary attention to detailed corner cases and/or seemingly unrelated system activity to flush them out. The ability to abstract and stimulate this sort of system activity for use in block-level, subsystem-level, chip-level, and full system-level simulation is key to finding the most difficult bugs. This requires dedicated people with special skills to achieve. In microprocessor design, experience has shown that a 2:1 (or even 3:1) ratio of verification personnel to logic design personnel is what is required to do the job.

>> *Plan for substantial verification efforts at each level of the design.*

>> *Plan for dedicated resources to attack the functional verification challenge.*

- ***Fully deterministic verification plans are not possible.***

Quantitatively, the problem is on the order of 2^{**N} , where N is the sum total of all of the state and inputs into the system (ie: an astronomically large number). As a result, the number of deterministic tests (that is, tests targeting a specific combination of events) is far too large to handle one at a time. While deterministic testing is still a valuable part of the overall verification plan, other techniques must also be applied. For example, low-level protocol checkers, random test generation, directed random test generation, random tests seeded from coverage-based feedback and sliding window tests to name a few. Each of these new methods offer new coverage, but they also have an infrastructure (and resource) cost associated with them.

>> New methods and associated infrastructure are required.

- ***For modern designs, 100% coverage is not possible.***

To manage the problem, we apply rationale thinking to eliminate cases, categorize relationships, and organize the verification attack. Even with the best of intentions, important cases will be overlooked, and can surface as bugs later on. The architecture and the design must be prepared to help work around latent problems.

>> Aggressive risk management is an important part of functional verification.

- ***Every bug found is a symptom, and should be treated as such.***

At the heart of every bug, there is some reason why it got introduced. Furthermore, there is some reason why it got caught at a particular point in the verification attack. It is important to understand these reasons, and use them as a seed in exploring the possibility (probability) of more problems along the same line. In the case of a late breaking bug, it is particularly important to determine why it took so long to find it (and then go back and adjust the verification environment for that part of the design).

>> The verification environment is dynamic and iterative.

- ***The functional verification burden extends into the post-silicon phase of the design process***

The hardest bugs are not found in the simulation models of the design. They are found in the actual hardware (and then typically recreated in the simulation models to aid in the debug and regression testing). Actual hardware, although limited in the granularity of control and stimulus, is actually the fastest “simulator” possible. It offers unique opportunities for driving the design into corners and exploring the associated “neighborhood” by leveraging the raw speed of execution. It encourages a level of verification attack based on the power of higher-level programming that would be inconceivable in a simulation environment due to the number of cycles required. Instead of stumbling on problems while trying to bring up classic software, the hardware-based verification effort targets problem areas directly with specially written software.

>> Plan for the development of special software-based “exercisers” for use in post-silicon hardware verification efforts.

- ***Verification considerations are a fundamental constraint on the architecture and design process.***

Architects and designers should be keenly aware of the verification implications for any design trade-offs that they make. Verification representatives should participate in all technical design reviews. A design that cannot be verified (or even on that takes too long to verify) is not worth anything (in fact, it can be a huge liability). In the near future, it will be common for designers to actually add hardware to help aid the verification efforts.

>> Early and active involvement by verification personnel is key.

- ***Excellence in verification technology is an outstanding career opportunity.***

The philosophies, technologies, techniques and skills required for quality verification are a key asset. Experienced verification personnel are one of the most sought after skill groups in the industry today. Developing these skills in non-trivial, and represents an outstanding full-time career opportunity for people. Verification is not a part-time activity, and it is not a training ground for future designers.

>> *Recognize and grow this special discipline in the organization*

2.2 Verification Techniques and Methodology

Each designer involved in a portion of a project introduces innovative thought and structure into their part of the overall design. As the scope of a particular project increases, so does the number of designers. In the end, the overall design is inevitably a product of lots of individual styles and structure at the low level. The architecture and design principles of the project usually offer enough structure to allow the overall design to come together, but it is critical to note that they do not offer a complete specification for functional correctness. The devil is in the details, and a good verification strategy orchestrates a *systematic attack* on these details.

Every design is inherently different, and therefore the details of every verification plan will also be different. There are also many ways of attacking the verification problem. While there is arguably no proven best method, a survey of the most successful projects in the industry suggest some common techniques and offer a structure for the overall plan. The following sections describe some of these techniques and structure in use today by the best-of-breed functional verification teams.

2.2.1 Optimize the Methodology for Simulation Efficiency

The amount of functional verification coverage that can be achieved is strongly related to the simulation throughput that can be realized. As a result, given the scope of the problem, it makes sense to optimize the efficiency of the simulation engines.

One optimization involves the use of so-called *cycle-based simulation models*. These models are optimized to only simulate the logical function of the design, and defer all other aspects of the analysis problem to other dedicated methodological flows. Typically, the simulation engine is tuned to perform zero-delay simulation so that the simulator can focus all of the computational power on the logical aspect of the design. Experience has shown that this approach can yield simulation throughput that is an order of magnitude larger than approaches that attempt to combine both logic simulation and timing analysis in a single session (so-called, *assigned-delay dynamic simulation models*). While most CAD tool companies today offer special cycle-based simulation engines, it is also typically possible to configure traditional simulators to emulate this style of simulation.

There are some important side effects of this type of modeling. First, since it does not offer any coverage for intra-cycle timing relationships, it demands a separate process for doing timing analysis. In most cases, this is covered through the use of a static timing analysis methodology (which, as it turns out, is a much more robust method for doing timing analysis anyway). Second, it often requires special modeling for certain types of circuits such as latches, arrays, PLLs, asynchronous logic, and analog devices. Of course, this special modeling of some circuits also obligates the overall methodology to close the loop on the abstractions used in these models.

Another optimization involves the use of large banks of computers configured to accept batch-style simulation jobs around the clock (the so-called “*sim farms*”). These machines are typically inexpensive networked PC’s that have been optimized to run simulations. One issue that quickly arises with this scheme is the cost of the simulation licenses needed for the machines. Once again, the methodology and tool selection should be structured to take this issue into account. Using a large number of state-of-the-art dynamic simulation engines in this mode is so cost ineffective that it is virtually impractical. Another key problem, of course, is knowing what to feed these machines, but that is described later in this document.

2.2.2 Early Development of a Comprehensive Verification Plan

The early development of a comprehensive verification plan is key. The plan can and should be started during the architectural phase of the project (for that matter, verification considerations can and should affect the architecture of the design). The plan should recognize and leverage the logical, physical and architectural hierarchy that is natural to the design. At each level of the hierarchy, it should systematically describe the verification requirements for all key mechanisms, state machines, modes, protocols, and interactions inherent in the design. It should recognize both the tops-down architectural view of these things, as well as the bottoms-up logical implementation view of them. Over time, the plan should be expanded to include the specific techniques and approaches used to cover each item articulated in the systematic analysis. The plan development (and resulting quality of the plan) should be the joint responsibility of the architects, designers, and verification professionals on the project. It is not reasonable to expect a quality design without the development and fulfillment of a high quality, comprehensive verification plan.

To aid in the development of the verification plan, the early stages of the conceptual design process should encourage a thorough pre-engineering of the design hierarchy before the design is entered into RTL. Forcing the design team to effectively do the design on paper before committing it into RTL has many positive effects. First, it forces the designers to think through the less mainstream cases sooner, and it forces them to communicate more clearly with other designers on inter-block protocols (the best way to eliminate design bugs is to not introduce them at all). This enables much more effective design reviews. Second, it forces a level of documentation and specification of protocols that can be leveraged in the development of the verification plan. And, third, this same documentation can be leveraged to define block-level verification environments that stress the design at a very fundamental level.

2.2.3 Hierarchical Attack

Common sense tells us that a hierarchical verification approach is sensible, and the historical evidence has validated this insight. Hierarchical verification encourages a systematic attack at each level of the design hierarchy. It works to build up confidence in the lower-levels of the design, and then address higher-level interactions at the higher-levels of the design. Different techniques and methods are used at each hierarchical level. Collectively, and when properly implemented, these techniques offer the basis for a defensible and proven verification strategy.

It is important to note that although the initial passes of hierarchical verification are naturally serial (that is, they engage as the design elements are pulled together), later passes of the design allow a parallel application of the techniques. This allows much more control over the regression and promotion process, and more stability in the expensive full system models.

The following subsections describe the approaches used in each level of the design hierarchy.

Designer-level Simulation

The designer of a piece of logic is ultimately responsible for its correctness. This includes both the logical mechanisms inside the design and the interface protocols for communicating with neighboring designs. A robust design flow encourages an early representation of the design in written form (dataflow diagrams, state machines diagrams, protocol definitions, and functional walkthroughs of key operations), and often in a high-level programmatic form (C, C++ models). The development of these representations forces the designer to think through the design details more completely (minimizing the *introduction* of bugs), and allows others to understand the design earlier as well. All of this should occur before the designer enters the design into RTL.

Once the designer has entered the design into RTL and before they pass it onto others, they have an obligation to make sure that it basically works as expected. Initially, this typically involves the use of manual interactive simulation techniques. As the design matures, these manual techniques are typically replaced by relatively simple and low-level *testbenches* (written and maintained by the designer) that stimulate and check the design in a more automated manner. Later in the design cycle, these designer-level testbenches are a key link in the regression chain that enables quick turnaround of higher-level models after bugs have been found.

As a designer fixes bugs in their design, it is important that they don't "create on onion" by promoting an updated design with obvious new bugs (thereby disrupting downstream simulation and analysis efforts). ***The goal of designer-level simulation is to find these types of problems as early as possible without disrupting anyone but the designer themselves.***

Many high performance teams in the industry make this a condition of basic designer integrity.

Block-level Simulation

A *block of logic* implies different things to different people, but for the purpose of this discussion, it refers to one element of the lowest-level partitioning of the overall design. It might be a state machine, or it might be the BIST engine around an array, or it might be an adder with some special bypass. Typically, it contains only a few main functions, and the logic internal to it has a natural self-affinity.

Block-level simulation is a systematic attack on a block of logic. It has enormous leverage for several very fundamental reasons. First, since the scope of the logic is relatively contained, it is possible to get very high coverage on the internal functions (possibly even exhaustive coverage). Second, if the interface protocols have been well designed and documented, it allows the block-level testbench to actively push these protocols through their range to an extent that is very difficult at the next level of hierarchy (in fact, it is often desirable to push the protocols through sequences that might not even theoretically be possible from the next level of hierarchy). Third, the block-level design is usually reasonably complete before the overall design is complete, so that verification can start earlier in the design cycle.

Block-level simulation can be achieved using several different techniques. First, similar to designer-level methods, it can start out using manual interactive simulation, but usually requires more automated methods over time. In addition, since ***the goal of block-level simulation is to do nearly exhaustive simulation of the internal function, and to push the interface protocols through their full range***, more automated and background-oriented simulation is usually required. The testbench, which performs setup, stimulates the model each cycle, and checks that the expected result is achieved each cycle (or, sometimes, at discrete intervals of the simulation) can become relatively complex. As a result, the testbench often demands the attention of a skilled programmer to develop it. Moreover, the value of such dedicated verification professionals with programming skills becomes clear in the development of these testbenches.

Another technique that can be applied at the block-level makes use of *model-checking* formal verification methods. In model-checking formal verification, the device under test is compared to a detailed specification of the design usually represented in a high-level programming language. Formal methods are excellent verification tools, but require a very complete specification. Also, today's tools have limited design size capacity. In general, both of these factors limit the use of model checking formal methods, but are increasingly used successfully for block-level verification.

In IP-based platform development, some of the individual blocks may not have been designed in-house. While this makes block-level simulation more difficult, it is still possible to do, and is recommended whenever possible. Again, by pushing the interface protocols to their extremes, additional practical insights can be gained about the appropriate use (or inappropriate use) of the third-party IP block.

Unit-level and Core-level Simulation

The unit-level represents the next level of hierarchy in the design. It combines many blocks together to achieve some higher-level function that is a product of the interactions between the blocks and other inputs into the unit. In very large designs, there is sometimes the need for a sub-hierarchy within this level (so-called multi-unit-level and core-level), but they all share a similar set of verification techniques.

The goal of unit-level simulation is to create as much interaction between the blocks as possible. The basic assumption is that the block internals have already been verified, so the techniques applied at this level can focus primarily on the real interactions between the blocks and the range of input values for the specific unit. In the process, of course, additional block-level coverage is also achieved but this should not be the primary method for achieving block-level closure.

For even a moderately sized unit, the range of possible interactions is enormous, so an exhaustive sequencing through each case is not possible. Instead, unit-level verification works to specifically identify the types of possible interactions and then *slide* combinations of potentially related interactions across one another in windows of time. In general, designs tend to contain a set of mainline mechanisms and associated interactions, and then a vast set of so-called corner cases that arise when less frequent oddball events are interleaved between or on top of these mainline operations. Although these cases naturally occur less frequently during normal operation, they represent some of the most complex scenarios and typically give rise to the nastiest bugs. As a result, a primary goal of the unit-level simulation environment is to create situations where these occur much more frequently than one might see in normal operation.

To achieve this, the verification environment needs to engage a certain degree of randomization in the testing. Further, pushing the design into the corner cases requires a certain degree of directed randomization (that is, get the design into a particular state and then perturb the test in subtle ways to explore a region in the state space). There are several important ramifications of this type of randomized testing. First, the number of cycles needed for simulation is very large. This puts increased pressure on simulation throughput and justifies some of the optimizations in the simulator described earlier in this report. Second, these techniques benefit greatly from coverage-based feedback into the test generation process. In order to achieve this, an enhanced coverage tracking mechanism needs to be in place, and the test generation needs to be seeded from the results harvested from the tracking. And finally, the overall test generation, checking and reporting mechanics become quite involved and quite complex. Again, the need for verification professionals with outstanding programming skills becomes obvious. The development of these environments is time consuming and naturally iterative. One hazard of this approach is that, in some cases, the environment itself can present as many bugs as the design under test. In the end, however, history has demonstrated that these unit-level verification techniques are extremely high leverage and are worth the effort.

Another technique that is sometimes successfully used in unit-level testing is *theorem-proving* formal methods. Theorem-proving formal methods are different from the previously mentioned model-checking formal methods in that they do not require the availability a separate specification, and can usually operate on a larger portion of the design. The concept behind theorem-proving is that by reading the design into a formal verification database and making queries about the design, one can draw conclusions about the closed-loop nature of the design (for example, “given this range of input values, does state machine A always return back to it’s initial state?”). The details of this technique go beyond the scope of this document, but it is important to note that they are being successfully used to augment traditional verification methods in many verification teams around in the industry today.

Chip-level Simulation

In chip-level simulation, **the goal expands to include multiple unit interactions, and verification of product functions that only come together at the full chip level.** For multiple unit interactions, chip-level simulation starts, once again, by systematically checking that individual chip-level functions operate as expected. This checks that the expected interactions between the units (pushed to their expected limits during block-level and unit-level testing) are implemented as described in the unit-level specifications. In effect, it establishes closure to ensure the validity of all the lower-level simulation efforts.

Testing of basic individual functions, however, is not sufficient. The next phase of chip-level verification strives to create as much parallel activity as possible. The comprehensive verification plan, described earlier, should form a blueprint for identifying operations that share chip-level resources. This sharing gives rise to an exponential number of new complexities, and chip-level verification works to leverage large numbers of simulation cycles to hit as many of these as possible. To maximize the efficiency of the simulation cycles, several techniques are used. First, *directed-random testcases* are developed. This type of testing differs from pure random testing (which, while it has a role in chip-level verification, also has some serious limitations) in that it identifies specific areas of resource sharing, and then biases the test generation process to create large amounts of activity in these areas. Second, special *warm-up profiles* are used to put the model into states that allow it to quickly engage interactions that would otherwise take hundreds or thousands of cycles to achieve through normal means. For example, rather than programming a DMA controller through the use of normal memory mapped storage operations, the warm-up profiles allow the testcase developer to reliably place state information directly into

the DMA controller so that it starts doing the interesting functions as quickly as possible. Typically, these warm-up profiles utilize a set of pre-existing loaders that offer an abstracted way to easily load up correct, and appropriate state into the blocks. Once again, development of these loaders takes very specific skills, and requires time and patience in getting them correct. Once operational, they become extremely powerful tools for the chip-level verification engineers. A third technique for improving the efficiency of chip-level verification efforts involves the use of coverage-based feedback into the test generation process. Of course, the first step for coverage-based feedback requires coverage analysis. This is described in more detail later in the report, but idea is that chip-level testcases are biased towards areas of lesser coverage based on this feedback. The most sophisticated environments in the industry today rely on some degree of automated biasing and testcase generation. The development of such an environment is clearly a very large effort, so in many cases, manual biasing is still used.

As mentioned above, chip-level verification is also meant to address all the other aspects of chip support functionality. This includes functional verification of the DFT infrastructure, BIST infrastructure, debug hooks, trace capture, performance monitoring, dynamic power management, power-on reset, special modes of operation, work-around hooks, and more. It is easy to underestimate the amount of effort required in these areas, and indeed, many emergency chip tape-outs have resulted from a lack of focus on these areas. Several of these areas require special testbenches and environments. They also require special skills that understand these non-mainline functions, and can develop the appropriate test plans. In some sense, these are the most important functions to be verified because they form the skeleton of the chip, and enable visibility into all the other functions supported by the design. All of these, and any others that are relevant to the specific details of the product, should be included in the comprehensive verification plan and the execution of that plan.

Another aspect of chip-level simulation involves performance verification. The performance of the final product is the result of many lower-level functional trade-offs that relate to cycle counts of key operations, arbitration schemes, forward progress schemes, bandwidth balancing and more. Many designs today make very fundamental trade-offs in the name of performance. Clearly, establishing functionality is a first-order objective, but right behind this lies the potential for many serious design flaws related to performance. A complete verification plan recognizes this and takes special actions to verify performance. Some performance verification can, and should, be done at lower-levels of simulation, but it is only at the chip-level where enough functionality comes together to really enable monitoring of the full picture related to performance. One method for performance verification involves the use of performance verification programs (PVPs) that, in addition to functional checking, add the ability to check the timing relationships between events (in cycles). A simple PVP might specify an acceptable range of cycles for the entire testcase to complete within. More complete PVPs create an environment that monitors specific events, and the relationships between events, throughout the run of the testcase. While it is tempting to manually verify key aspects of performance by interactive (or post-simulation) analysis of the events trace, it is highly desirable to develop regression quality PVPs that can be run with each new model drop. Experience has shown, that as designers struggle with difficult bug fixes and timing closure, they often inadvertently change the event-based performance relationships. PVP regressions can help spot these induced problems as early as possible.

A final aspect of chip-level verification that should be highlighted is power verification. In this context, power verification involves checking that the functional conditions that cause the various mechanics of dynamic power management to engage and disengage do so as expected, and without affecting the normal functional correctness of the design. For example, if a particular clock is gated based on upstream state (in an attempt to save power by only switching downstream logic when it is needed), it is important to check that this only occurs when appropriate. Note that even if all of the surrounding mainline logic is 100% correct, a bad specification on clock gating can negate all of that functionality. As power management becomes more of an important differentiator, the need for complete verification of these design hooks is mandated.

System-level Simulation

In system-level simulation, a complete model of the system is developed. This includes the specific design (or designs) that have been the target of all of the lower-level simulation efforts, as well as other system elements that the design will ultimately interact with. These models are typically very large, and they are expensive to develop (that is, they require broad synchronization and stabilization of all elements of the design). Due to the size of the model, simulation throughput is diminished, so it is important to recognize the role of these models in the overall verification strategy.

The system-level model allows verification closure on how well the design interacts with external elements (memory, expansion slots, serial interfaces, graphics devices, etc). Accurate models of these elements are required, and while some are available from the manufacturers or other third-party modeling companies, some need to be written in house. In any case, the task of pulling all of these models together into an integrated system model is quite large and should not be minimized. As with each level of the hierarchical verification approach, each of these interfaces should be systematically attacked to make sure that the design can operate across the full range of the specified interface.

The system-level model also enables a limited amount of real software bring-up and debug. Ideally, a separate C-level system model is produced in parallel with the real design to enable more efficient software co-design, but it is still reassuring to be able to run sections of the real software on the simulation model. In addition, the system-level model enables the development of special exercisers that become extremely useful in post-silicon verification efforts (see below).

Post-Silicon Verification

Verification need not, and should not, stop at tape-out. The eventual availability of real silicon capable of running at speed opens up a huge new verification opportunity. In some sense, the real silicon is the ultimate simulation model of the design, and the verification plan should take advantage of it. The reality of today's chip design cycles is that multiple tape-outs, representing a convergence on the final product specifications, are a standard practice. As a result, although very late in the design cycle, problems found in post-silicon verification activities still have an opportunity to be fixed before the product is finalized.

Of course, the availability of silicon enables the testing of actual software written for the product, and this is a very important function. It is also important to recognize that most software relies on basic functions, and only occasionally stumbles into the more interesting, bug infested corner cases. In the specific area of microprocessor design, for example, bringing up an OS on silicon is a significant moral milestone, but it is not considered to be a significant stress of the design.

The silicon has the advantage of raw speed, but it also has the disadvantage of accessibility to stimulate and check internal node values. This combination gives rise to a very different form of verification that leverages high-level programming languages to produce *special design exercisers* that target specific areas of the design. In general, the goal of these exercisers is to massively create scenarios that might not ordinarily come up all that often. For example, an exerciser might bias the interrupt controller to cause exceptions much more often than a normal OS would. At the same time, it might work to sustain heavy DMA traffic or other events to ease the design into corners that might not be obvious from inspection of the design. The specifics of the appropriate exercisers are a function of the design itself, and usually require some very creative thinking to stimulate and check interesting scenarios.

2.2.4 Coverage Tracking

The age old question in functional verification is “How do you know when you are done?”. The state space in today’s designs is so huge that it is clearly not possible to cover every case. On the other hand, with every design there is an obligation to verify that it functions correctly across the full range of possible scenarios. The development of a comprehensive verification plan and use of the hierarchical methods described in this document attempt to offer a rationale approach for managing this paradox.

In addition, best-of-breed verification methodologies make use of coverage tracking to help quantify the amount of coverage achieved in the plan. Typically, this involves the development of some form of coverage model that attempts to itemize the required scenarios. As mentioned previously, a complete coverage model would consider all possible states and all possible inputs, but due to the raw size of the problem, this is not possible. Instead, many methodologies make use of one or more simplified coverage models. These simplified models represent a necessary but not sufficient set of scenarios. In the end, they do offer useful information about scenarios that have never been hit, but it is important to note that they do not offer much insight about the degree to which the total set of scenarios have been hit.

One coverage model, called *code coverage*, helps to illustrate this point. Code coverage leverages the structure of the RTL code itself, and works to show which RTL statements have been evaluated at some point, and which have not. More sophisticated code coverage models extend this to include a directional element to how the statement was reached. In either case however, simply hitting a section of the RTL code does not offer a comprehensive view of the total state space. Context (relative to the total state space) is important, and code coverage fails to offer insight into the machine context when each statement was hit.

Code coverage can be enhanced through the addition of *event-based counters*. The events monitored by these counters are typically manually specified and represent very important scenarios in the design. Event-based coverage recognizes that hitting a particular scenario once is not sufficient, and provides a means for counting the number of times a particular scenario has been hit. If an important scenario has been hit thousands of times with random or directed-random testing, then there is a reasonable chance that many of the interesting corner cases (the context) related to that scenario have also been hit.

The most sophisticated verification environments actually use the data from coverage analysis to automatically help direct future testcase development. The mechanics behind such an environment are complex, and require dedicated resources with a passion for automating such a process.

The bottom line is that coverage tracking can help show what has *not* been hit in the design. It is tempting, but dangerous, to associate high confidence in a design just because some (inherently incomplete) coverage model indicates a high hit rate

2.2.5 Discipline, Control and Management

As described above, modern verification methodologies can be come quite involved and quite complex. As a result, appropriate management of the verification effort is a very important part of the overall verification plan. The following paragraphs describe some of these key management functions.

Design Regression and Promotion Process

A well managed design regression and promotion process is key for achieving the maximum efficiency and throughput of the available simulation resources. The goal of this process is to ensure that every new model release is better than the previous model, and that no obvious new problems have been introduced by the fix of some other bug. This process starts at the lowest levels of simulation, and it is the goal of each level to minimize disruption imposed on the higher-level models. As mentioned previously, many best-of-breed design teams informally make this a condition of designer integrity.

Infrastructure Development

From the discussions throughout this report, it should be obvious that the verification environment itself can require a significant effort. It is, in fact, a moderately-sized software engineering project that should be managed as such.

Measurement

Measurement of key verification metrics is also key (“*you get what you measure!*”). Project managers should develop the means to monitor certain aspects of the verification environment, and track these across time. This provides insight into the behavior of the bug curve, the efficiency of resource usage, the impact of bad practices on overall progress, and many other management-like things. It also provides a historical reference that can be important for predicting how much time and resources will be needed on future projects.

Commitment to Philosophies

The project leadership needs to constantly remind themselves and the rest of the team about the verification philosophies accepted as part of the project. For example, if “*every bug is a symptom*”, then the management and leadership team should enforce a process that generates the appropriate discussion on every bug. Words are cheap, and falling short of best-of-breed is a common problem in the industry. It takes serious commitment and dedication from the leadership team to step up to a best-of-breed process.

2.2.6 Design for Verification (DFV)

In the microprocessor industry, design verification closure has become such a significant problem that verification-related considerations have become a primary constraint on the overall project. In many cases, the architecture of the design is actually changed based on feedback from the verification teams. This scenario is growing and should be encouraged.

Another aspect of *design for verification (DFV)* that is growing in importance recognizes the need to offer workarounds for functional problems that escape to silicon. Typically, these so-called *chicken-switches* are built into the design to allow a bypass of some mechanism featured in the design. Common examples include workarounds for performance optimizations, special modes, clock gating logic, and others. Design elements that have accumulated a lot of bugs over time, and/or ones that have seen late-breaking bugs should be considered good targets for adding workarounds. Although the addition of these workarounds complicates the design and verification slightly more, they tend to be invaluable later in the design cycle.

In the future, it is likely that more logic will be added to the design to help with verification (and/or workarounds). In the same way that scan-based testing has imposed constraints on the design, verification will likely impose new constraints as well.

3 Conclusions

As designs become more complex, they become more difficult to functionally verify. The problem is fundamentally exponential, and traditional methods are becoming inadequate. Given that time-to-market and design quality are key differentiators in the market, it is clear that new methods need to be invented and engaged to help manage the problem. Investment in these types of improvements is both practical and cost-effective in the long run.

This paper has described some of the best-of-breed functional verification techniques in use by the industry today. These techniques have spawned from a set of verification-focused philosophies. The first step towards improving functional verification is to accept these philosophies as fundamental maxims in the overall design and development process.