Practical Approaches to SOC Verification

Guy Mosensoson
Verisity Design, Inc.

Abstract

This paper provides some guidelines on how to approach System On a Chip (SOC) verification, and how to create effective SOC testbenches. It surveys the challenges in SOC verification and some of the traditional verification techniques, and then focuses on showing preferred practical approaches to the problem.

1. Introduction

This paper starts by giving an introduction to SOC verification and its hardships. We look at typical SOC designs and the traditional verification techniques applied to them, commenting on their benefits and inherent limitations. As part of that, we mention the components of an SOC and some effect they have on the design of the testbench.

The rest of this paper focuses on showing practical and novel approaches to the verification of SOC designs. Experience shows that many SOC bugs result from incomplete integration of IP cores and complex scenarios and corner cases between the software and hardware components of the system. Techniques for dealing with both of these are discussed.

All material presented is based on the combined experience of many verification experts in many application domains. The technology mentioned is mature and available, and all that remains to be done is to develop "verification environments as they should be".

This paper discusses simulation-based verification, or in other words, verification done by running well-targeted simulations in self checking verification environments. This paper does not discuss any formal techniques, which are considered complementary.

It is our belief that creating good and capable verification environments involves some degree of programming. People write testbenches using a large variety of languages and tools: Verilog/VHDL, C/C++/PERL, and some verification languages and tools customized for verification.

2. Why are SOC Designs Special

Let us open by defining what an SOC is and is not. A System On a Chip (SOC) is an implementation technology, not a market segment or application domain. SOCs may have many shapes and many different variants, but a typical SOC may contain the following components: a processor or processor sub-system, a processor bus, a peripheral bus, a bridge between the two buses, and many peripheral devices such as data transformation engines, data ports (e.g., UARTs, MACs) and controllers (e.g., DMA).

In many ways, the verification of an SOC is similar to the verification of any ASIC: you need to stimulate it, check that it adheres to the specification and exercise it through a wide set of scenarios. However, SOC verification is special and it presents some special challenges:

Integration: The primary focus in SOC verification is on checking the integration between the various components. The underlying assumption is that each component was already checked by itself. This special focus implies a need for special techniques.

Complexity: The combined complexity of the multiple sub-systems can be huge, and there are many seemingly independent activities that need to be closely correlated. As a result, we need a way to define complicated test scenarios as well as measure how well we exercise such scenarios and corner cases. [1]

Reuse of IP blocks: The reuse of many hardware IP blocks in a mix-and-match style suggests reuse of the verification components as well. Many companies treat their verification IP as a valuable asset (sometimes valued even more than the hardware IP). Typically, there are independent groups working on the subsystems, thus both the challenges and the possible benefits of creating reusable verification components are magnified. [4]

HW / SW co verification: The software or firmware running on the processor can be verified only in relation to the hardware. But even more than that, we should consider the software and hardware together as the full Device Under Test (DUT), and check for scenarios that involve the combined state of both hardware and software. Thus we need a way to capture hardware and software dependencies in the tests we write and in the coverage measurements we collect.

Unique bugs: Here are some typical areas where bugs appear in SOC designs:

- Interactions between blocks that were assumed verified
- Conflicts in accessing shared resource
- Arbitration problems and dead locks
- Priority conflicts in exception handling
- Unexpected HW/SW sequencing

All the challenges above indicate the need for rigorous verification of each of the SOC components separately, and for very solid methodologies and tools for the verification of the full system. This requirement for extensive verification indicates the need for a high level of automation, otherwise the task of verification will simply become impractical.

3. Trends in Traditional SOC Verification

Companies and design groups around the world have many different approaches to verification and specifically to SOC verification. Thus it is quite impossible to say "this is how things are typically done". However, there are several characteristics that can be seen quite often. Lets take a look at some of the current trends and their limitations.

Test plans: Many companies apply the same techniques they used in ASIC verification to SOC verification. These typically involve writing a detailed test plan, with several hundred directed tests, and describing all sorts of activities and scenarios the designers and architects deem important. While these test plans are important and useful, their effectiveness is limited by two main factors:

- The complexity of the SOC is such that many important scenarios are never thought of
- As complexity of systems grow, it becomes harder to write directed tests that reach the goals

Test generators: Each directed test hits a required scenario only once, yet there is a real need to exercise those scenarios vigorously in many different combinations. This indicates the need for ways to describe generic tests, that can exhaustively exercise areas of interest. Many companies write random tests, but those are usually used only at the end of the verification cycle. While these tests can "spray wildly" and reach unexpected corner cases, they still tend to miss a lot of bugs. The problem is that these tests spray "blindly" in all directions, just like the sun floods the whole earth with its light. What we actually need is to focus most of our light into the dark and unexplored allies… we need generic test generators, that can be easily directed into areas of interest, and can adjust their light beams to any required width.

Checking integration: Many SOC verification testbenches have no special means for verifying correct integration. Instead, the system is exercised as a whole, as well as possible, under the assumption that any failure can be detected by some false side effect it will create (e.g., one of the data packets passing through a switch will be corrupted). The main draw back to this approach is that finding the source of the problems by tracing the corrupted data all the way back to where it originated from consumes too much time. This points out the need for integration monitors that could identify integration problems at the source.

HW/SW co verification: there are several commercial tools and in-house solutions that enable HW/SW co verification. By running the real software on the simulated hardware, one can debug the hardware and software together before the final production. However, these tools do not typically have the capabilities to look at the hardware and software as one single DUT. They may control the stimuli to the hardware, and may allow modifying software tables or variables, but it is usually impossible to describe scenarios that capture hardware and software dependencies. There is usually no way to describe scenarios such as sending a specific input to the hardware while the software is in a specific state or in a specific interrupt service routine. To exercise an SOC design to its limits, there needs to be a way to capture HW/SW dependencies as part of the test description, the checking rules and the coverage metrics.

When are we ready for tape out?: Every design group ultimately needs to answer this question. The means for answering are always insufficient, as verification quality is so hard to measure. Code coverage, toggle or fault coverage and bug rates are all useful measures, but they are very far from complete, and fail to identify many of the complex combined scenarios that need to be exercised in an SOC. To solve this dilemma, there is need for coverage metrics that will measure progress in a more precise way.

To summarize, there is always an element of "spray and pray" in verification, hoping you will hit and identify most bugs. In SOCs, where so many independent components are integrated, the uncertainty in results is even greater. There are new technologies and methodologies available today that offer a more dependable process, with less "praying" and less time that needs to be invested. The following sections will discuss some unique approaches in that direction.

4. Preferred Approaches in SOC Verification

Many questions come to mind when approaching the task of SOC verification: What does it mean to verify the SOC and in what way is this different than just verifying the parts? Should we verify the hardware, the software, or both? How can we reuse the verification of the blocks when verifying the SOC?

Indeed, we need to address all these questions. We need to take advantage of the fact that the SOC has IP and pre-

verified blocks in it. We need to remember that there are indeed two DUTs in the SOC: the hardware is the first DUT and the full SOC with both hardware and software is the second. In the following sections, we first focus on the verification of the hardware. Verification of the hardware and software together is addressed in section 4.5.

A theme that runs through all the previous sections is the need for capturing the complexity of an SOC into an executable verification environment. Verification always starts from the specifications: the design specs and the test plans. The goal for the verification engineers is to create, based on these specs, a full verification system, and do that as fast as possible, with minimal effort. Thus the enablers must be:

- Means to capture all SOC specifications and complexity in an executable form
- Automation of all verification activities
- Reusability of verification components

An important component in accomplishing the level of reuse and automation we seek is a high level verification tool, as discussed in [2, 3].

4.1 Looking At The Big Picture

When verifying an SOC, there is a need to look at the full system, and to integrate all details into a full coherent picture. One practical guideline is to take the programmers view of the system, and focus on it. Often the SW interface spec (or programmer's guide) is the document that best describes the SOC as a whole.

Another useful guideline is to define the verification in terms of high-level transactions, preferably look at end-to-end transactions. For example, for a packet router, packets might be received through an 8-bit interface. In this case, the test scenarios should be defined in terms of the end-to-end packets, not in terms of sequences of 8-bit vectors.

As trivial as it may sound to "look at the big picture," many verification teams do not do so. Out of habit or lack of experience, they take the low-level interface definition (MII, PCI, EUTOPIA, etc.) and create a testbench that "speaks" in the same terms as the pin interface. Instead of dealing with packets, GIF images, or printer commands, the tests deal with bit vectors or low-level primitive actions.

4.2 From Unit Level To Full System

SOC designs are built bottom up, from many units (i.e., hardware blocks) that are assumed to be verified, and are often reused in multiple designs. The fact that the basic units, often IP blocks, might be used in so many different contexts imposes a need to verify these units in any possible scenario that their spec may allow. This can be achieved using spec-based verification, where the rules and properties in the specs are captured in an executable form, allowing tests to span all possible points in the problem-space. This relates to IP verification more than SOC verification, so we will not expand on it here.

One of the things that can easily boost the full system verification is using some of the unit verification components. In many cases, it can be very straightforward to use them. Checkers of internal block properties can be integrated into the full verification system. For example, these checkers could check that all transitions in a state machine are legal, or that some mutually-exclusive signals are never asserted together. Coverage metrics for the "stand alone" core can still be useful in the full system. For example, a CPU coverage report may show if all possible opcodes were exercised with all possible operands.

There are other verification components that may be reused, provided that the unit verification is planned correctly. For example, some of the unit level test generators can be reused. As the units are assembled into a full system, many of the unit interfaces become internal interfaces between adjacent blocks of the SOC, and there is no longer need to drive their inputs. However, other interfaces remain external interfaces in the SOC. If test generators for external interfaces are maintained as independent components, then most system-level stimuli can be taken as is, from the various unit environments. For this to be practical, the test generators need to be flexible and configurable. With some of the existing verification tools and languages, it is very easy to create such generic test generators, and even later configure them for unexpected needs without modifying the core code. We will speak more about reuse in the next section.

Another useful technique, which we will just mention briefly, is building verification "shadows" for the actual units. These shadows can really help building the verification "bottom up". The shadows may be very high level reference models of the blocks. They may be interface compatible but can be very abstract in their internal implementation. These models can be assembled together into a "shadow system" for early prototyping of the SOC, before all actual HW blocks are ready. Later, as the HDL for the blocks become available, they can provide partially shadowed systems in which various sub-sets of the block can be exercised. Even when all blocks are ready and integrated, the shadow models can serve as reference models for the checking of each block.

4.3 Creating Reusable And Flexible Testbenches

Separating reusable verification components (such as external interface stimuli, checkers and coverage) is a nice and easy start for collecting reusable verification components. But there is much more to say on verification

reuse. The key to having good reusable components is the way we model the verification system. Lets take a look at some of the aspects we need to consider:

Verification environment vs. test scenarios: The *verification environment* is the infrastructure, it should be generic and developed for long-term use. This will make the *tests* easy to write and easy to maintain. The benefits of this can be sweeping.

On the side of "long term," the verification environment should include a description of all data structures, SOC interfaces, protocol rules and SOC properties. It should include generic test generators, checkers and coverage metrics. As a result, the environment will be self-checking and the effort in creating tests is focused just on describing the scenarios of interest. Even complicated tests can be written in a few lines of code if the infrastructure and primitives are well defined.

This approach saves a lot of redundant and repetitive descriptions found in many lower-level testbenches, and saves significantly in development and maintenance time. It contributes to verification reuse in several areas:

- The verification environment is kept generic, its components can be easily ported to work in other SOC designs and the full environment can be easily adapted to modifications in the SOC.
- The tests are short and descriptive, focus on the scenario described, and can be unaffected by implementation changes in the SOC. In case of changes or modifications of the SOC behavior, tests typically do not need to change, because the changes can be done in the infrastructure of the verification environment.
- Even the regression suite itself can be reused in different contexts. For example, if the tests are defined as sequences of Ethernet packets, they can be run on two systems that have different Ethernet interfaces (e.g., MII nibble interface vs. Twisted Pair single bit interface)

Orthogonal SOC components: Many components in the SOC can work independently and in parallel. In order to exercise the SOC to its limits, the tests should be able to describe parallel streams of activity for each component separately. On the other hand, sometimes these components sit on the same bus, and there is need to run interleaved transactions to access all of them in any order. To attain both the independence and the ability to interleave and synchronize activities, the information on the various independent entities, and the inter-relations, need all to be captured in the verification environment. This will allow us to later write tests in either approach. High-level verification tools can support that.

Modeling transactions and BFMs: An important part of the reusable code is keeping the description of components and activities at an abstract level. There are two central concepts in the abstraction of SOC verification environments, both of them are essential in creating reusable verification code:

- *Transactions*, which can capture the building blocks of protocol activity or of SOC scenarios. For example, a transaction can be a command to a DMA block. Execution of the transaction involves a sequence of six register reads and writes. The register reads and writes are the lower-level primitives, while the test scenarios focus on the DMA higher-level transactions.
- *Transactors*, which capture all rules of activity on the SOC interfaces, and translate the high level transactions into the lower-level activity on the bus or pin interface. Bus Functional Models *(BFMs)* are one common example of such transactors. They relieve the tes writer from the bus details, and supply a set of primitive actions such as read_IO, read_MEMORY, write_IO, write_MEMORY.

Modeling the verification environment around transactions and BFMs is very helpful in creating reusable verification code.

Reuse between groups: In SOC design more than other design projects, there are many separate groups working on the verification of the units, and possibly separate people work on the full-system verification. It is important that all verification code can be shared between those groups, and especially the group involved with full-system verification. Writing high level, short, and clear code capturing the verification needs is essential for sharing these components. HDL code, especially when written for verification, tends to be hard to maintain and share. A high-level verification language can promote passing code between groups.

4.4 Verifying IP Integration (Integration Monitors)

As mentioned earlier, the primary focus of SOC verification is on integration. The integrated hardware blocks are usually assumed to be correct, and the area where most bugs appear is in the interactions between those blocks The task of integrating blocks is even harder when incorporating external IP blocks that come from other companies, because there is no local expert that understands exactly what goes on in the IP.

An integration monitor that comes bundled with the IP can be of great help. Especially if it can be plugged into the simulation environment with no knowledge of the IP and just "do its job". Such a monitor should be able to supply two main services

- Automatically identify any violation of the IP interface

rules and report it immediately. This can save time dramatically, as the problem is reported when it can easily be debugged. Without this, it could take days to trace down the cause of a late side effect.

- Measure the overall quality of the IP integration verification. This sort of report should point the system integrator to areas and block interfaces that were not exercised thoroughly enough. Then, the integrator can continue testing the unexercised areas and avoid unnecessary silicon re-spins

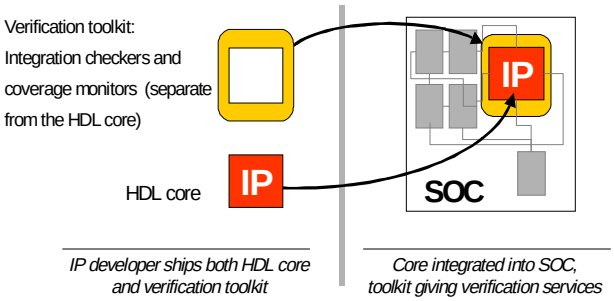

Figure 1. Pure IP toolkits

Ideally, the integration monitors are written by the IP developer and distributed to all users of the IP core. Such IP monitors can bring immense benefits, both in saving time and in quality of the SOC. The same type of monitors can be applied to any interface between blocks in the SOC, and especially to buses, which interface between multiple blocks.

Interface checkers are usually elaborate descriptions of complete handshake or bus protocols. It can be very hard to capture all rules and follow lengthy sequences of events in an inadequate language. A high-level verification tool can effectively describe all temporal properties and legal scenarios of the interface

To measure test quality, there is need to measure which scenarios were encountered, and make sure all interesting combinations of data and sequences were exercised extensively. To achieve that, the verification tool should be able to easily capture all functional behaviors, create combined reports showing the matrix of all relevant combinations, and give an easy way to measure progress. Given such tools, the decision to stop testing and tape out can be much less arbitrary.

Another issue with IP verification is that the IP can be integrated into SOC designs written in VHDL,Verilog or in both. These designs may be run on many different simulators. It is therefore required that the IP verification component will be able to run with any HDL language and any simulator

IP verification toolkits such as those described above can be shipped today, as described in [4].

4.5 Full Visibility into HW/SW Co-Verification

HW/SW co-verification is an essential part of SOC verification. Most SOC have one (or more) processors on chip. As mentioned earlier, there are several commercial tools that enable HW/SW co-verification. But if we want to consider the hardware and software together as the full device under test (DUT), we need full visibility into both hardware and software. We need a way to capture hardware and software dependencies in the tests we write and in the coverage measurements we collect.

Let us take an example to illustrate this. Assume we have an ATM router that does most routing in hardware. For some special types of cells received, the hardware issues an interrupt so that the software may make some routing decisions. In standard co-verification environments, we could send a stream of ATM cells to the router, and execute the interrupt handling routines on the co-verification system. But that would not be enough. We would not be able to measure how well we exercised the HW/SW interactions, nor would we be able to direct tests to untested areas.

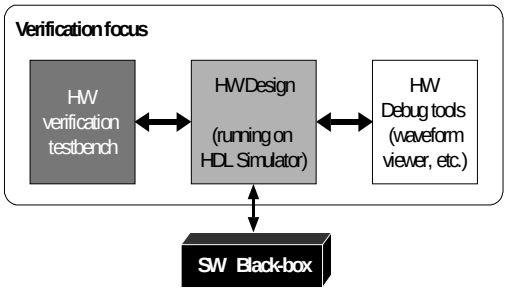

Figure 2. Typical hardware debug environment

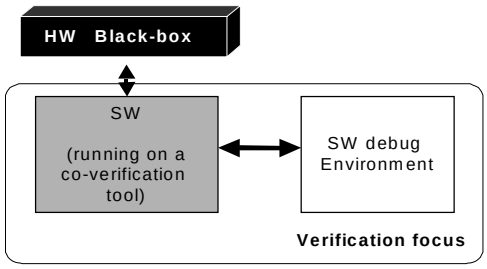

Figure 3. Typical software debug environment

A fully capable co-verification system will be able to integrate all hardware and software state information, and give us access to it for all levels of verification: test stimuli,

checking, and measuring test coverage. Continuing with the example above, we would be able to define test scenarios where any required type of ATM cell could be sent while the software was in any given state, or while software is in a specific interrupt handler routine. We would be able to stress the HW/SW interactions to their limits, and then look at a coverage report summarizing all concurrent hardware and software states exercised.

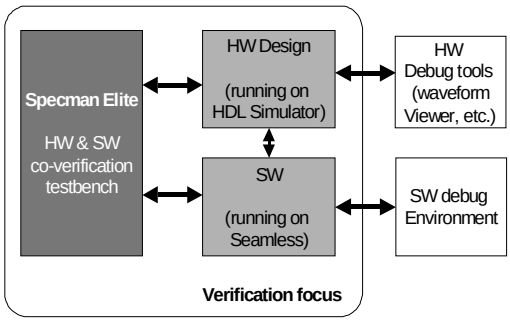

Figure 4. Full HW/SW co-verification using Specman Elite and Seamless

There are many corner case bugs that are extremely hard to find without such techniques. A typical bug in an ATM switch could be that the hardware might overwrite a management cell that the software handler is processing, if the software is not quick enough in removing the data it needs. Software seems OK, hardware seems OK, it's just the combination between the two that is buggy. In this example, the bug could manifest itself only if a management cell is received while the software is processing a previous management cell.

Now lets assume such a scenario was never exercised by your tests. How would you identify such a hole in coverage, if you don't have a combined report that shows which packet types were received by hardware while the software was in each possible interrupt handler? And then how would you direct a test to exercise that scenario, if you couldn't send specific cell types dependent on the SW being in a specific routine?

4.6 The "Coverage First" Paradigm

Coverage is an important tool for identifying areas that were never exercised. But there are two other important benefits it can give:

- Identify areas that were sufficiently exercised, and therefore need not be exercised any further
- Replace the need to write a lot of deterministic, delicately crafted tests, by showing that these scenarios were already encountered

The reason these two are much less recognized is that most people think of code or toggle coverage when they speak of coverage, while the biggest value can be found, in fact, in functional coverage.

Both code coverage and toggle coverage are good "first-level indications" for areas that were never accessed. However, they can never tell you if you have exercised "enough". Exercising enough means that all your test plan was executed, and all interesting scenarios were exhausted. Code and toggle coverage do not give you any indication of "functionality" covered. For example, code coverage can not tell you if all types of cells were received on all ports of an ATM switch, nor can it tell you which sequences of opcodes were executed by a CPU. You can achieve 100% code coverage, and still miss key areas where bugs can be hiding.

Functional coverage, on the other hand, allows you to define exactly what functionality of the device should be monitored and reported. This means you can make your functional test plan executable. You can get reports that will measure exactly what you care about, and describe that information in your own terms.

Looking at functional coverage reports, you may conclude that certain features or scenarios of interest in a certain area were already exercised to the full extent. Seeing that, you could stop running tests in that area, and focus your efforts on the areas that were neglected.

A functional coverage tool can also be used as a query mechanism, to investigate further into what was or was not exercised. You should be able to interactively explore more combinations of events. For example, if you have a coverage metric on a state machine, and a metric on the opcode a CPU has processed, you may be able to combine the two and see what opcodes were processed while the state machine was in each possible state.

But most significant impact of functional coverage in context of SOC verification is in eliminating the need to write many of the most time consuming and hard to write tests. Using functional coverage, you can describe complicated scenarios of interest as a coverage metric. Instead of spending days on crafting a few directed tests to hit these corner cases, you can write one generic test aimed at the whereabouts of those corner cases. Such a test can run with some variance factors multiple times, for a full day, and flood the area of interest. The test might hit the corner case only occasionally, but at the end of the day, you may open the coverage report and see that the corner case was encountered several dozen of times, in a variety of scenarios. You may have achieved, in less than an hour, more than you would have achieved in days of writing directed tests. Using the coverage report you can also choose the best test instances and add only those to the regression suite you run

periodically.

Why is this so important for SOC designs? Because in SOCs the interdependencies and test complexity are such that fulfilling the verification task by writing an exhaustive set of directed tests is virtually impossible. Using the approach of "coverage first," we can shift our main effort from expensive test writing to the much cheaper analysis of results. Instead of spending much time on writing delicately crafted tests, we can run some generic, wide angled tests and investigate the coverage reports of the results. Typically 80% of the interesting scenarios will be covered without any special effort on our side, rather early in the verification cycle.

5. Conclusion

SOC verification might seem very similar to ASIC verification at first glance, but it is actually special in many aspects. The main focus of SOC verification needs to be on the integration of the many blocks it is composed of. As the use of IP is prevalent in SOC designs, there is need for well-defined ways for the IP developer to communicate the integration rules in an executable way, and to help the integrator verify that the IP was incorporated correctly. The complexity introduced by the many hardware blocks and by the software running on the processor points out the need to change some of the traditional verification schemes, and trade them in for more automated verification approaches. Similar conclusions can be seen in [1, 2, 3].

References

[1] D. Geist, G. Biran, T. Arons, M. Slavkin, Y. Nustov, M. Farkas, K. Holtz, A. Long, D. King, S. Barret, *"A Methodology For the Verification of a 'System on Chip',"* DAC, 1999.

[2] C. Hanoch, "High Level Verification Automation: A New Methodology For Functional Verification of Systems/ ASICs", DesignCon, 1998

[3] "Spec-Based Verification", http://www.verisity.com/html/technical_papers.html

[4] "Functional Verification Automation for IP, Bridging the Gap Between IP Developers and IP Integrators", http://www.verisity.com/html/technical_papers.html