

Chapter 2

Functional Verification: Challenges and Solutions

Chapter Introduction

This chapter will discuss the overall design verification (DV) challenges and solutions. Why is DV still such a long pole in the design cycle? We will discuss a comprehensive verification plan and see the type of expertise required at each step of verification and how to improve the develop => simulate => debug => cover loop.

2.1 Verification Challenges and Solutions

So, what are the specific challenges that SoC design verification faces? Why is it such a long pole? Let us look at the higher-level challenges and their solutions. After that, we'll go through a detailed verification plan that the author has successfully deployed in many successful projects.

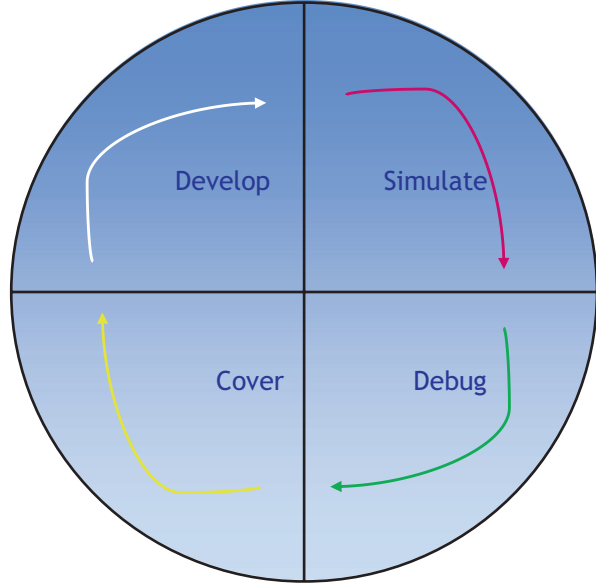
Here's a simplified, but to the point, functional verification cycle (Fig. 2.1). The cycle consists of four phases:

1. Development: verification plan, DV architecture, testbench, and tests development.
2. Simulation: software simulation, acceleration, emulation, etc.
3. Debug: transaction level, signal level, etc. This will be a big component if you did not deploy assertions, for example.
4. Cover: functional, code, and SVA coverage that feeds back to the development stage.

Each of these four phases poses significant challenges. Obviously, we need to reduce the time to complete and improve efficiency and robustness of the tasks at each stage:

1. Reduce time to develop and improve robustness.
2. Reduce time to simulate and improve simulation accuracy and throughput.

Fig. 2.1 Typical design verification methodology loop



3. Reduce time to debug and improve efficiency.
4. Reduce time to “comprehensive” cover.

Let us solve the challenge of reducing time and improving robustness at each stage of the functional verification cycle.

2.1.1 *Reduce Time to Develop*

Development time includes functional verification plan development, verification environment creation, DV architecture development, testbench development, and tests development. Of these, the tests and testbench development are the most time consuming. Here is a strategy to reduce time to develop.

1. Raise abstraction level of tests. Use TLM (Transaction Level Modeling) methodologies such as UVM, SystemVerilog/C++/DPI, etc. The higher the abstraction level, the easier it is to model and maintain verification logic. Modification and debug of transaction level logic is much easier, further reducing time to develop testbench, reference models (scoreboard), peripheral models, and other such verification logic.
2. Use constrained random verification (CRV) methodologies to reach exhaustive coverage with fewer tests. CRV is discussed in detail in Chap. 5. Fewer tests mean less time to develop and debug.

3. Develop verification components (e.g., UVM agents) that are reusable. Make them parameterized for adoptability in future projects. UVM is discussed in detail in Chap. 4.
4. Use SystemVerilog Assertions (SVA) to reduce time to develop complex sequential and combinatorial checks. As we will see, assertions are intuitive and much simpler to model, especially for complex temporal domain checks. Verilog code for a given assertion will be much lengthier, hard to model, and hard to debug. SVA indeed reduces time to develop and debug. SVA is discussed in detail in Chap. 6.

Verification Plan Development

Instead of using the age old .docx- or .xls-based plans, use the new technology/tools available from EDA vendors that allow an automated and coverage-driven verification plan development in an organized way. The plan needs to correlate to the design specification in an intuitive and comprehensive manner. Changes in design specs should be automated to reflect in the verification plan.

Nothing is fully automated in life (unfortunately). Even with automation, you need to carefully layout what is it that you need to include in a verification plan. Verification plan is only as good as your knowledge of the design architecture and to some extent microarchitecture. Detailed attention needs to be given to issues such as asynchronous FIFO, state transitions, live locks, dead locks, etc. Relying only on end-to-end verification can miss corner cases. All this needs to be specified in a verification plan. A comprehensive verification plan will go a long way in reducing time and effort consumed by the later stages of verification.

A comprehensive verification plan is presented in Sect. 2.2.

2.1.2 Reduce Time to Simulate

- Higher-level abstractions simulate much faster than pure RTL testbench which is modeled at signal level. Use transaction level testbench (e.g., UVM, TLM 2.0 transaction level models). TLM reduces time to develop, debug, and simulate. Electronic system level (i.e., TLM 2.0 level modeling) has come a long way for practical deployment in verification environments. Refer to Chap. 11 for complete detail on ESL and TLM 2.0 methodology.
- Deploy well-thought-out hardware acceleration, emulation, or FPGA prototype methodologies. Develop transaction level testbenches that interact directly with the accelerated or emulated design. Chapter 12 is devoted to deploying hardware acceleration, emulation, and virtual prototyping to speed up simulation as well as develop software.
- Use coverage-driven verification (CDV) methodologies to reduce the number of tests to simulate to reach the defined coverage goals. Refer to Chap. 7 on functional coverage to understand CDV.

2.1.3 Reduce Time to Debug

- Use SystemVerilog Assertion-based verification (ABV) methodology to quickly reach to the source of the bug. As we will see (Chap. 6), assertions are placed at various places in design to catch bugs where they occur. Traditional way of debug is at IO level. You detect the effect of a bug at primary output. You then trace back from primary output until you find the cause of the bug resulting in lengthy debug time. In contrast, an SVA points directly at the source of the failure (e.g., a FIFO overflow assertion will point directly to the FIFO overflow logic in RTL that failed) drastically reducing the debug effort.
- Use transaction level methodologies to reduce debugging effort (and not get bogged down into signal level granularity).
- Constrained random verification allows for fewer tests. They also narrow down the cone of logic to debug. CRV indeed reduces time to debug.

2.1.4 Reduce Time to Cover: Check How Good Is Your Testbench

- Use SystemVerilog *functional coverage* language to measure the *intent* of the design. How well have your testbench verified the “intent” of the design. For example, have you verified all transition of write/read/snoop on the bus? Have you verified that a CPU1-snoop occurs to the same line while a CP2-write invalid occurs to the same line? Code coverage will not help with this. We will cover functional coverage in plenty detail in Chap. 7.
- Use *cover* feature of SystemVerilog Assertions to cover complex *sequential* domain specification of your design. As we will see in Chap. 7, “cover” helps with making sure that you have exercised low-level sequential domain conditions with your testbench. *If an assertion does not fire, that does not necessarily mean that there is no bug.* One of the reasons for an assertion to *not* fire is that you probably never stimulated the required condition (antecedent) in the first place. If you do not stimulate a condition, how would you know if there is indeed a bug in the design? “Cover” helps you determine if you have indeed exercised the required temporal domain condition.
- Use code coverage to cover *structural* coverage (yes, code coverage is still important as the first line of defense even though it simply provides structural coverage). As we will see in detail in the section on SV functional coverage, structural coverage does not verify the *intent* of the design, it simply sees that the code that you have written has been exercised (e.g., have you verified all “case” items of a “case” statement, or toggled all possible assigns, conditional expressions, states, etc.). Nonetheless, code coverage is still important as a starting point to measure coverage of the design.

2.2 A Comprehensive Verification Plan

Following verification plan is typical of large SoC projects. Let us also establish during each step the type of expertise required. That will tell us the need for a well-diversified DV team. Each of the following verification plan point is discussed in detail throughout the rest of the book. This is mainly a higher-level snapshot. Chapters 15 (Voice over IP SoC) and 16 (Cache Memory Subsystem) will showcase real-life verification plans based on the outline described below.

Here is the outline of a comprehensive verification plan.

1. *Identify Subsystems Within Your SoC*

- For example, audio subsystem, memory subsystem, graphics subsystem, etc.
- You start verification of subsystems and then move onto concurrent subsystems leading to full system verification.
- Subsystem also allows you to determine a methodology for subsystem-block level stimulus/response methodology. Block level verification can/will be imported to subsystem level verification.
 - Expertise: Hardware design architecture and microarchitecture.

2. *Determine Subsystem Stimulus and Response Methodology*

- For example, graphics subsystem will require a “way” to feed the external bus with a single/multi-frame input. What should be the format for this input data? How would you measure response? How many different UVM agents would you need? What type of scoreboard? Reference Model?
 - Expertise: Hardware design architecture and microarchitecture. UVM, SVA, C/C++, SystemC/TLM2.0.

3. *Stimulus Traffic Generation Requirements*

- For example, what type of traffic would you generate for a video subsystem? Will that be single frame? Multi-frame? Do you require a live video stream? How will that simulate on RTL? How would you verify without a live stream?
- How will you generate Ethernet traffic? An external software stack (won’t find all the bugs) or a constrained random packet generator (much better choice)?
 - Expertise: SoC design verification, UVM, SVA, C/C++, SystemC/TLM2.

4. *Subsystem Response Checking Methodology*

- How will you check the output of a video engine/subsystem?
- How will you detect a corrupt Ethernet packet transmission?
- How will you check for SoC interrupt generation logic (i.e., did the logic generate an interrupt when a corrupt Ethernet packet was received)?
- For a CPU, how will you check for the CPU architectural state integrity at the end of an instruction?

- What about reference model generation? Do you need it? What will be the sync point between transaction level reference model output and signal level RTL output?
- What is the methodology for deploying assertions both at the microarchitectural level and block/SoC IO level interface?
 - Expertise: Reference model generation, UVM, SVA, TLM2.0, etc.

5. *SoC Interconnect: Determine Verification of Either an NoC (Network on Chip), Cache Coherent NoC- or Bus-Based Interconnect*

- Again, how would you generate real traffic to verify the interconnect?
- Would you create “stub” models (i.e., BFM/UVM agents) to act as initiators and targets of the interconnect? What kind of stimulus would you provide to these UVM agents to stress the interconnect?
- What would be your methodology for performance measurement of the interconnect?
 - Expertise: Knowledge of NoC and bus interconnect. UVM, SVA, SFC, NoC generation tools.

6. *Low-Power Verification*

- Identify power subdomains. Power switches. Isolation cells. Retention cells.
- Create UPF generation methodology. Can it be automated?
- Determine stimulus strategy to verify each individual power domain turned ON/OFF. Verify the same with multiple power domains.
 - Expertise: UPF, low-power technology, power domains, power switches, isolation cells, retention cells, single- and multi-power domain concurrency.

7. *Static Formal or Static + Simulation Hybrid Methodology*

- Static formal (as of the writing of this book) does not work at full SoC level for a large SoC. So, start with identifying control logic paths where assertions can be written to limit the logic cone(s). Identify critical clock domain crossing blocks, critical state machines, asynchronous logic interfaces, etc. Static formal has the problem of state/space explosion for larger logic blocks. Static formal exercises all possible permutations of combinational and sequential domain tests to see that the assertion holds. So, what’s the strategy to partition the blocks?
- If static formal does not work, identify tools that allow static + simulation hybrid simulation. This methodology will simulate inputs to the logic cone, determine valid input values, and then apply static formal to see if assertions hold.
 - Expertise: Static formal and static + hybrid formal tools.

8. *SystemVerilog Assertions (SVA) Methodology*

Deploying SystemVerilog Assertions is one of the most important strategies you can deploy to reduce time to cover, develop, and debug. Carefully plan on writing assertions for microarchitecture, subsystem, system, IO interface, inter-block interface, and critical state machines.

- Assertions need to be added to RTL by designers while block and SoC level interface assertions need to be added by the DV engineers.
- How will you know that you have added enough assertions? Rule of thumb is if a test fails and none of the assertions fire, you haven't placed assertions in the failing path.
 - Expertise: SystemVerilog Assertions language and methodology. Note that UVM does not cover SVA language.

9. *Functional Coverage*

- Determine logic that needs to be *functionally* covered.
- How will you leverage code coverage with SystemVerilog functional coverage?
- What's the strategy to constraint stimulus to achieve desired functional coverage?
- How will you determine that you have specified all required coverpoints and covergroups? This is the hardest (and sometimes subjective) question to answer. Continue to add functional coverpoints as the project progresses. Do not consider the very first coverage plan to be an end in all.
 - Expertise: UVM, SystemVerilog functional coverage language. UVM does not cover SFC language.

10. *Software/Hardware Co-verification?*

- Think about deploying advanced methodologies such as TLM2.0 (ESL) $\Leftrightarrow$ RTL. This allows you to speed up software running on a virtual platform of the SoC. TLM2.0 is transaction level and so is UVM. So, integration of UVM testbench with software virtual platform will not have significant challenges. You will be able to run software code with such methodology. If TLM2.0 virtual platform is not your cup of tea, have a plan to deploy hardware acceleration or emulation to do hardware-software co-verification.
 - Expertise: TLM2.0 SystemC. C++ expertise. SystemC TLM2.0 is an entirely different language orthogonal to SystemVerilog. Simulation acceleration, emulation, and FPGA prototyping.

11. *Simulation Regressions: Hardware Acceleration or Emulation or FPGA Prototyping*

- What are the pros/cons of acceleration vs. emulation vs. prototyping?
- Acceleration will have better debug capabilities than emulation.
 - But the speed maybe in a few MHz at best.

- Does acceleration provide enough speed for software development?
- If the testbench is still in SystemVerilog (i.e., outside the acceleration box), will the SystemVerilog $\Leftrightarrow$ acceleration maintain required speed?
- What about memories? What about multiple clocks? What is the debug strategy?
- How about assertions? Will they compile into acceleration hardware?
- How will functional coverage be measured?

Expertise: Knowledge of hardware acceleration. RTL to acceleration netlist mapping, multi-clock domain, etc.

- Emulation will be orders of magnitude faster than acceleration.
 - BUT emulation cannot start until RTL is ready. Since that is the case, will it be too late for software development?
 - How easy/hard will it be to debug since internal node visibility maybe poor.
 - What about assertions and functional coverpoints? Will they be emulated?

Expertise: Knowledge FPGA (or not)-based emulation technology. ASIC RTL mapping to FPGA logic or acceleration hardware, clocks, SCHEMI interface, compile times, etc. UVM does not cover this.

12. *Virtual Platform Methodology*

- This is the ESL/TLM2.0 methodology. Do you need it? How will you use a virtual platform as a reference model to check SoC response?
- There are significant advantages to this methodology.
 - You will be able to develop software before the RTL is ready.
 - You will be able to create and verify tests before the RTL is ready.
 - The virtual platform can act as a reference model to match the architectural state of the SoC at transaction boundaries.

Expertise: TLM2.0 SystemC standard language. C++ expertise.

13. *AMS (Analog/Mixed Simulation)*

- What will be the verification strategy to verify analog $\Leftrightarrow$ digital boundary crossing?
- How will you generate analog behavioral models? Simulation with Spice will be extremely slow and won't be practical.
- How will you guarantee analog behavioral model is 100% accurate to Spice model?
- Do you need to deploy real number modeling?
- How will you do low-power verification on analog $\Leftrightarrow$ digital boundary?
 - Expertise: Real number modeling (RNM), Verilog-A language, AMS technology, analog $\Leftrightarrow$ digital interface, analog behavioral modeling (non-RNM), etc.

ASIC/SoC Functional Design Verification
A Comprehensive Guide to Technologies and
Methodologies

Mehta, A.B.

2018, XXXI, 328 p. 175 illus., 160 illus. in color.,

Hardcover

ISBN: 978-3-319-59417-0