Debugging continues to be one of the biggest bottlenecks in today’s design flow. Yet, when discussing the topic of debugging among project teams, the first thought that comes to mind for most engineers is related to the process of finding bugs in architectural and RTL models, as well as verification code and test. However, debugging touches all processes within a design flow—including the painful task of coverage closure. In fact, one of the most frustrating aspects of debugging is tracking down a particular coverage item that has not been hit only to learn that the coverage item is unreachable.

In this article we explore the debugging aspect of coverage closure; with a focus on the unique ability of formal technology to automatically generate simulation exclusion files to improve coverage results while reducing the amount of time wasted trying to hit unreachable states.

INDUSTRY COVERAGE TRENDS

Coverage is one form of metrics used by today’s project teams with the goal of providing a clear, quantitative, objective measure for assessing their overall verification process—as well as assessing the project’s progress towards their completeness goal. Within the last three years we have seen a rapid maturing of the industry in terms of adoption of various coverage techniques. For example, the 2010 Wilson Research Group Functional Verification Study, as shown in Figure 1, found that the industry adoption of code coverage had increased by 50 percent since 2007. In addition, functional coverage adoption had increased by 80 percent within the same three-year period.

Clearly, coverage metrics have come of age for most of today’s SoC design projects.

THE COVERAGE CHALLENGE

As engineers start the process of capturing coverage during simulation, quite often they are initially excited by how rapidly their coverage metrics increase after only a few simulations, as shown in Figure 2. However, this optimism very quickly gives way to reality as the coverage metrics begin to level off without reaching their coverage goals. Finishing the job is a classic case of the 80-20 rule. That is, closing the last 20 percent of coverage will likely take 80 percent of the process time.

At this point, the engineer begins the painful process of debugging the root cause behind not hitting various coverage items. In general, there are two issues contributing to the coverage closure problem: (1) insufficient or incorrect input stimulus during simulation, and (2) an unreachable coverage item. To solve the first issue requires the engineer to identify the missing or incorrect input stimulus and either write a directed test for the correct case or adjust the constraints within their constrained-random testbench. For the second issue, the engineer needs to identify that a coverage item is unreachable. In general,
there are two root causes behind unreachable coverage. That is, the design might have a bug in it that is preventing the coverage item from being reached, or it might not be possible to reach a particular coverage item under certain IP configurations or error conditions.

Debugging these unreachable issues is both frustrating and time consuming. In fact, on a typical SoC project, it is not uncommon for a verification team to spend hundreds of man hours debugging coverage closure issues. Automating the process of identifying unreachable coverage items is obviously a big productivity win for these teams.

GENERATING COVERAGE EXCLUSIONS

As part of the signoff review process, many project managers require that the verification team present their final coverage results and assess the risk for not hitting a particular coverage item. Obviously, the verification team wants to minimize the effort of reporting and justifying unreach ed coverage items during the review process, particularly for items that really do not matter. Therefore, as engineers identify a coverage item that is unreachable and is not a bug, they often manually create exclusion files to effectively remove the coverage item from the design's coverage space. However, this manual process of creating exclusion files is often error prone, and the maintenance cost associated with these coverage exclusion files can be high as the design evolves.

Recently, a better approach has emerged that uses formal technology to automatically identify unreachable coverage items within a design and then automatically generate coverage exclusion files to be used in simulation for coverage items that are truly unreachable. There are obviously many benefits to this automatic approach, such as increased productivity, identifying bugs in the design, and ensuring that the coverage exclusion files are always up-to-date and accurate prior to regression runs.

THE SOLUTION

In this section, we describe our methodology for using formal technology to automatically improve coverage results.

The first step in the process is to run automatic formal to explore the reachability of the design. This can be done early in the design cycle, that is, as soon as the RTL code is ready, as the code itself is the primary input to this step. In addition, the designer may also define the operating conditions in the form of constraints, for example, setting constants to disable test mode. The tool will report code coverage goals that are proven to be unreachable through formal exploration of all possible combinations and sequences of legal inputs. At this point, it's important to review these unreachables to ensure they are valid, since, as mentioned above, they could be an indication of a functional bug. Once that is completed, the next step is to run simulation to capture code coverage results.
During the simulation phase, the exclusion directives generated in Step 1 for the unreachable coverage goals are included to prune the unreachables out of the coverage model. The code coverage metrics for the entire regression suite are captured and merged into an amalgamated database of results in this second step.

The third step in the flow is to run automatic formal on the design as in Step 1, but with the merged simulation coverage database as the second main input, which causes the formal tool to focus on only the uncovered goals in the database and target them with high effort to attempt to obtain a conclusive result. This step also differs from Step 1 in that the formal tool will be directed to create a witness for the coverable goals, that is, a waveform showing the sequence of input values that cause the design to reach a state that covers the goal. For any goals proven unreachable, again, it’s critical to review them and approve removing them from the coverage model. For coverage items hit through formal analysis, the witness trace can be used as a guide to help reach this un-hit goal in simulation, either through creating a directed test or by modifying constraints used to control random stimulus. In addition, it’s important to note that if a coverage item is not hit over the course of running the entire regression suite and formal analysis cannot determine conclusively that it can or cannot be reached, then it’s a strong indication that the logic in that area of the design is either very complex and/or not easily controllable. This information can help to guide modification of the design to make it more easily verifiable.

The fourth and final step in the flow is to prune out the coverage goals proven unreachable in Step 3 from the simulation database and generate accurate code coverage metrics that reflect what is truly reachable in the coverage model of the design.
One pitfall to be cautious of when implementing this flow is that of overconstraining the design. If formal analysis is run with constraints that over-restrict the input space that is explored, then unreachable coverage goals could be reported that are in fact reachable through normal operation of the design. This could result in pruning and ignoring coverage items that should be explored and hence missing bugs in the untested parts of the design. Therefore, it is recommended that any constraints used for formal are run as asserts in simulation to uncover any discrepancies.

SUMMARY

In this article we explored how formal technology can be used to improve coverage results while reducing the amount of time wasted trying to hit unreachable states. Questa™ Formal AutoCheck supports this methodology by automatically identifying unreachable coverage items and then automatically generating simulation exclusion files.

To learn more about benefits using formal technology to improve coverage results from a user’s perspective, we highly recommend the 2011 User2User paper by John Gryba from Alcatel-Lucent titled, “Improving FPGA Quality Through Code and Functional Coverage.”