DesignCon 2006

A Comparative Study on the Effectiveness of Automated Assertions Versus Traditional Methods in a Project Design Flow

Andreas S. Meyer, Assertive Design, Inc.
andreas.meyer@assertivedesign.com, (603) 674 3631

Abstract
Assertions are generally considered to be valuable. However, they have not been widely adopted for a variety of reasons, including the expense of writing and maintaining a separate assertion model, and the difficulty of incorporating assertions into existing design flows. This paper examines the use of an automated assertion tool to make assertions practical in RTL design flows. We examine and compare results of three design projects that used automated assertions in parallel with traditional design. We then study the effects on the design, verification, and flow and compare overall project productivity and quality using each method.

Author(s) Biography

Mr. Meyer has been involved in digital design and verification for more than 20 years in engineering, project management, and senior management. Andy is the author of *Principles of Functional Verification,* a reference manual on developing and managing the verification process.

He holds a BSEE from the University of Illinois at Champaign and an MSECE from the University of Massachusetts at Amherst.

Introduction

Front-end digital design suffers from a massive functional verification bottleneck. As is commonly reported, approximately 75% of the design effort is spent in verification. Even with this large and increasing amount of verification effort, the average number of spins for the average ASIC is reported to be greater than 2 and growing. While similar metrics aren't readily available for FPGA-based designs, there is a clear increase of time and effort spent in lab debug for FPGAs, even after a significant verification effort.

Design groups are attempting to escape this bottleneck through many means, such as more complex verification languages, more abstract design languages, emulation, and heavy IP re-use. Many of these methods offer incremental improvements, but none solve the problem and all come with at least some additional cost if not high costs. IP re-use, for example, comes with the problem of debugging bugs in third-party IP. It has been reported that approximately 80% of bugs in SoCs are inside the reused or purchased IP.

Some of the most promising work in solving this problem has been with assertions, both static and dynamic. There is a range of assertion tools available, however assertions are seldom integrated into a design flow other than for special cases. It is not atypical to see a design of greater than 1 million gates with considerably fewer than 1000 assertions – which is unlikely to provide any significant coverage. This skewed ratio appears to be due to the cost of writing and maintaining assertions during a project. It generally falls to the designer to add both check and cover assertions into the RTL, and then test and maintain the assertions. While this adds significantly to the design effort, the maintenance effort is larger still, since for every change to RTL, the appropriate assertions must be found, modified, and re-tested. The result is that the time and discipline required to learn the assertion language, write, test, debug and maintain assertions can triple the design effort.

This paper examines three design projects that used a new design tool -- DesignPSL -- that generates automated check and cover assertions. The check assertions fire any time a functional discrepancy between the design intent and in the RTL code. A cover assertion fires any time a simulation exercises a specific functional component of the design. The check assertions are both static and dynamic.

All three projects had at least some components created using traditional tools and languages as well as DesignPSL. We compare the results of the two methods, looking at the design and verification processes, and then examining the overall project productivity and quality from each method, based on the experiences of these three designs.

Project Overview

Of the projects we studied, two were direct copies of an existing RTL design. A direct copy means that all of the logic, registers, and state machines were copied without significant modification. One project was a rewrite of an RTL design. In this project the RTL was created directly from the specifications without referencing the existing RTL

All three had data path, control logic, standard bus interfaces, and several clock domains. Two designs were meant to fit into a large FPGA, and one design was meant to be a component in an FPGA.

Overview of Traditional Design

A common design flow is shown in Figure 1. There are two separate teams, one doing the design and the other doing the verification of a component. The design team develops the synthesizable RTL, while the verification team writes functional verification tests using some type of High-level Verification Language (HVL). The verification code was written in Verilog, C, and E, with two projects using a mix of Verilog and C.

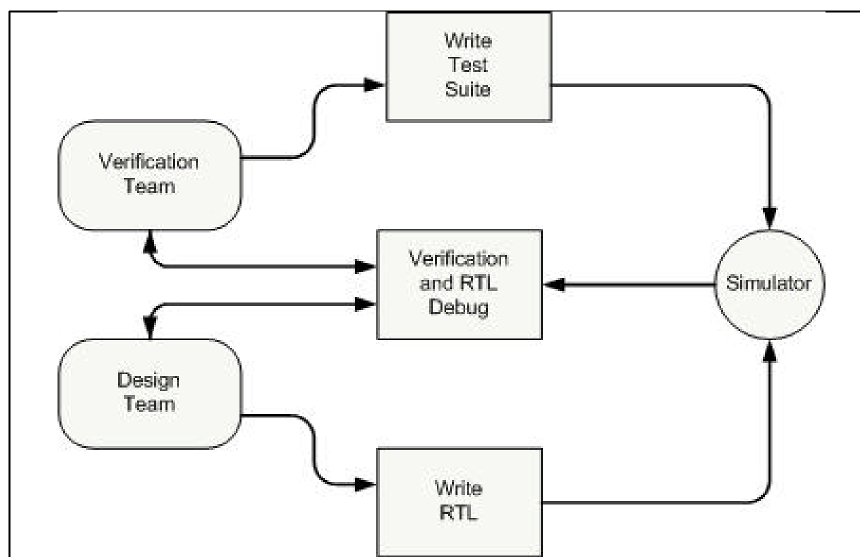

Figure 1: Traditional RTL Flow

For the traditional flows, there were several consistent issues:

? There was a long lag time from when synthesizable RTL was first written to when the verification environment was able to find anything beyond superficial bugs.

? The inability to measure verification effectiveness until late (if ever) into the project cycle made it difficult to modify the verification approach early.

? Designers were somewhat removed from the verification flow. They did not know exactly what was being verified or how complete the verification effort was, and when bugs occurred. They would have to learn about specific tests, and how to work with the verification tools in order to debug their code.

Overview of DesignPSL

DesignPSL is based on the idea that assertions are a powerful tool for functional verification, but far too time-consuming and complex to be written and maintained in a normal design flow.

In order to make DesignPSL acceptable to engineers to use, it had to be very similar in style to Verilog or VHDL. It is about a one-hour process for an experienced Verilog design engineer to learn DesignPSL.

The flow of a design project using DesignPSL is shown in Figure 2. Both the design and verification teams follow exactly the same project flow as in the traditional design. However, by using DesignPSL, assertions are generated automatically. When an assertion finds the conditions it was looking for, it will fire, letting the user know that the condition was found, and generating a report with all of the details about when and where the condition was found.

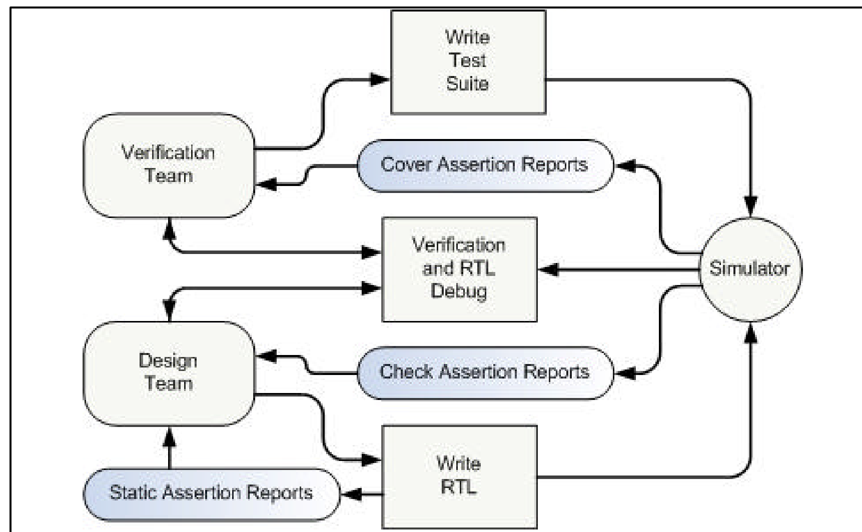

Figure 2: DesignPSL Flow

Two types of assertions are generated: check assertions, which will fire when a functional conflict is found anywhere in the RTL code, and cover assertions, which will fire when specific functional areas of the RTL are covered.

The assertions are all functional, based only on the RTL. Many of the check assertions are static, and will fire before a simulation is run. Many are temporal, covering multiple cycles of logic. Across the three projects, there were an average of 1.8 dynamic check assertions (dynamic) and 0.9 cover assertions generated for every line of RTL code. It is hard to count the static assertions exactly. A rough counting method yields approximately 5 static checks per line of RTL code.

Solving Bug Tracking Issues

Bug tracking systems tend to be used as a communication mechanism between the design and verification groups. The verification group files a bug generated by a test, while the design group works through the list of bugs to resolve, closing bug reports as they are fixed, or determined to be invalid.

In the DesignPSL flow, we found a significant departure from this pattern, because a designer was often the first to see bug information, often with no simulation, or in the designer's minimal testbench. Without the need for communication between the groups, designers would see the assertion report and fix the bug without ever entering data into the bug track system.

The benefits of this are immediate feedback and the ability to find errors while the design is still fresh in the engineer's mind. Unfortunately, this made direct comparisons between traditional and DesignPSL-based flows harder since many bugs were found and fixed without ever being entered into the bug track system. Designers were able to find and fix bugs from assertion reports before the design was integrated into the verification environment. As a result, we had to talk with designers about the errors they found and fixed, and separate syntax errors from functional bugs. A frequent first response was: There were no errors." Only by asking about syntax errors and then reviewing each one were we able to find errors that would have been considered functional bugs in the tradition RTL flow. This was particularly true with static assertion reports that are reported back to the designer at compile time and seem like syntax errors. This represents a positive fundamental shift from the design flow today. Designers are accustomed to detecting and repairing syntax errors in their code prior to handing that code off to the verification team. In the DesignPSL projects, we saw the same for some of the functional bugs

In Figure 3, an example of this is shown from an SDRAM initialization state machine with approximately 15 states. The last state shown was intended to increment the bank count and loop back to the start of the state machine. However, the loop logic didn't have the right state entered.

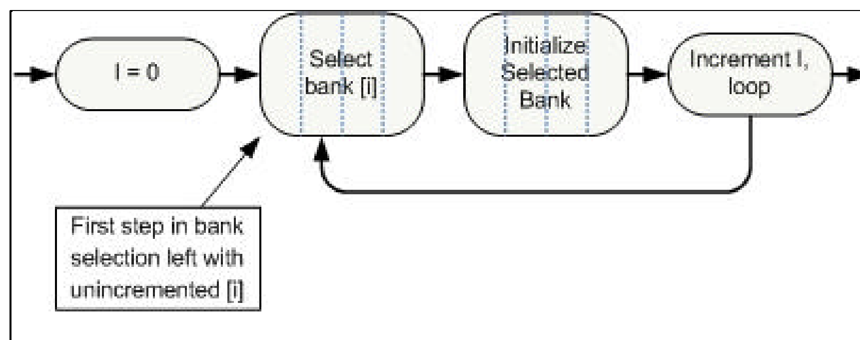

Figure 3: Example of Static Assertion

In the traditional RTL design, this error was still undetected. In the exact translation DesignPSL, this was a statically checked assertion that was reported at compile time and appeared as a syntax error. This type of error detection and report make the bug seem, essentially, of no importance because it is so obvious and is fixed so easily.

Debug Method

How one goes about finding the source of a bug varies based on the method of bug detection, as does the time required to debug.

In the traditional RTL flow, the verification team is usually responsible for examining test failures and determining if there is a problem in the RTL or in the test. For RTL bugs, the designer goes through a fairly constant sequence of steps: Determine what the test is doing; find the incorrect behavior between the input and the output; trace the vectors from the test back in time and into the RTL; determine when and where the RTL behaved incorrectly; fix the bug, then rerun the regressions to see that the bug is fixed and no new bugs are introduced. Repeat until the bug is fixed and no new bugs are introduced.

While this method is time consuming, it is well understood and designers are generally comfortable with it as long as they are able to understand what the verification test is doing.

Further, a large percentage of "bugs" are simply misuse or misunderstanding of the intended functionality by the verification team. The designer must still review and categorize these issues and educate the verification team as to the correct use of the device.

The process for the DesignPSL flow is somewhat different, and it did require some adjustment for designers on the first few bugs. Since DesignPSL displays an assertion report that points directly to the RTL source and the time, there is no need to understand the verification test. (In fact the test may actually pass despite the assertion detecting a functional error.) Note that this is different from other assertion-based tools where the assertions are separate from the RTL code, and the designer must determine the relationship between the assertion and the RTL, since the error in the case of manually generated assertions points to the assertion itself rather than to the RTL.

The steps that the designer must take to debug based on DesignPSL assertions are: Read the assertion report; understand how the assertion was caused – this often requires looking at interactions between parts that were active in the RTL to determine what was happening; fix the bug; rerun the regressions to make sure that no new bugs were introduced.

While there was some time for designers to get used to debugging from an assertion report, once that had happened, the process removed the largest problem in the debug process. There was no longer the need for the designer to trace back through time from a

test failure back to the source of a bug. When an assertion fires, it points directly to the time of failure, and the location in the RTL where the bug occurred.

The hardware design team examined the assertion reports first. In most cases, the assertions pointed to RTL failures. However, there were some that pointed to test failures. Examples of this would be improperly configured control registers, or undriven pins causing X/Z migration into the RTL logic.

Another change that occurred in the flow was the early introduction of coverage data into the debug flow. Traditional flows use coverage data as a means of analyzing the completeness of their test suite and gauging where additional testing is required. With DesignPSL, the coverage data often came into the debug process as a means of determining what the device was doing and had done prior to the failure. The coverage data served as the inverse of an assertion to affirmatively state what functioned correctly versus what did not (the assertion).

Most Common Assertion Reports

We tracked the frequency of assertion reports to see which assertion types triggered the most, and the types of errors that were found with each type. Some of these are fairly mundane, but others we found quite interesting. Figure 4 shows the frequency of the most common error types. Examples of each of the most common assertion types will follow.

Percentage of Total	Assertion Type
15%	Transition Failure
14%	Default Max Repeat
11%	Optional Boolean
10%	Size Mismatch
9%	Sere-OR Multiple Path
9%	Sere-OR No Path
7%	Unknown Condition
5%	Clock Domain Crossing
3%	Register Collision

Figure 4: Frequency of Assertion Types

The order in which these assertions would fire during the test process was unrelated to frequency. Figure 5 shows the order in which the most common assertions usually fired. The static assertions are reported at compile time, so they were always reported first. After that, the typical ordering appears to depend on how general the fault is. Some conditions happened almost any time the clock was running, while others required a specific sequence of events to occur.

Order of Assertion Firing	Assertion Type
1 (static)	Size Mismatch
2 (static)	Clock Domain Crossing
3	Unknown Condition
4	Transition Failure
5	Default Max Repeat
6	Optional Boolean
7	Sere-OR Multiple Path
8	Sere-OR No Path
9	Register Collision

Figure 5: Order that Most Common Assertions Fired

Many of these assertions are fairly mundane, just a normal part of early RTL debug. The primary difference is that the assertions point almost directly to the problem in the RTL, alleviating the necessity of tracking back through any verification code. Following are examples of each type of assertion report, how each triggered, and how each was debugged.

Transition

Transition failures occur when a signal was expected to change value but did not. Any time a signal needs to be set to a new value, a designer may specify either a normal equal, or a transition. Both will set the signal to the new value. The transition operator specifies that the new value must always be different from the old value. The equal operator specifies that the signal value may be the same. If the equal operator is used, then coverage assertions will check that, at some point in simulations, the signal was set to the value it was already at. The use of a transition operator becomes, essentially, automatic to the designer. In most cases the designer understands how they expect the signal to operate, so inserting a transition operator is a way to confirm that understanding..

In all the projects we examined, transition assertions were the most common type of assertion firing. They happened for a variety of reasons. The most common reason occurred early in code bring-up when some RTL was still incomplete. In one example, a request signal transitioned high at the start of an operation, but the line of code was missing that should transition the request low at the end of the operation. The next time the operation started, the transition assertion fired, reporting that the request was still high.

In addition to the more mundane bring-up issues, the transition assertion also found most architectural errors. In one case, two devices that were supposed to be mutually exclusive both accessed a RAM at the same time. A transition failure on an internal RAM enable line fired as soon as the second device started to access the memory controller. In another case, the architecture specified that a fixed sequence of events was

were expected to occur in a pipeline. On any occurrence of event A, event B was supposed to be the next event to occur. A shared signal was expected to transition to 1 on event A, and transition to 0 on event B. When the transition assertion fired, it was discovered that event C was able to occur between A and B. In this case, the assertion was the first indication of an architectural issue, and the assertion provided an early discovery of the architectural bug.

Default Max Repeat

Any time that a loop is constructed in a state machine, such as waiting for a grant after asserting a request, a default maximum repeat value can be specified which will fire an assertion if any loop ever goes beyond a specified number of cycles.

This assertion fired frequently during early bring up, when incomplete equations or missing signals caused a state machine to hang in a particular state. In one project, the common practice was to set the default maximum repeat to 10,000. This number was large enough to prevent any false positive error reports, but still small enough that an assertion would fire quickly when a state machine stopped behaving correctly. The default max repeat assertion was a convenient assertion because it pointed to the exact state of the hang, and with the help of the debugger, the designers could see where other state machines were relative to the hung state machine. As a result, finding the source of the problem was straightforward.

Once past initial debug, the default max repeat did find a few architectural bugs because it was often the first assertion to detect a state-machine deadlock condition.

Optional Boolean

The optional Boolean assertion occurs when the designer puts additional constraints into a design. This is essentially the designer writing additional assertions that will be checked, except that the assertion is dropped right into the RTL, with the same style and syntax as the RTL. We had a higher than expected number of these probably because several designs were translated from a traditional RTL design, and the optional Boolean was seen as an easy way to test that the translator understood how the design was expected to function.

One example of an optional Boolean that failed occurred in a section of control logic that triggered on an address match. Once the address match was detected, the control logic performed a fetch. An optional Boolean was put in to validate that the address continued to match during the fetch operation. It didn't, and the assertion fired. Another piece of logic was attempting to increment the address earlier than expected.

Examples of optional Booleans that didn't fire include checking that address and data didn't change during a ram RAM write, or that a fifo was not full during a write.

Size Mismatch

The size mismatch assertion fires statically, and is reported as a syntax error. This type of error should have been caught by a lint program, but our experience shows that the lint

program wasn't run. The size mismatch assertion fires when the result of a size mismatch is going to result in lost or meaningless data. The most common example of this was a 1-bit wire being set to the value of a 5-bit field. In most cases, the actual source of the size mismatch was a functional error, such as in the error above, the width of the wire was not specified correctly, and critical data was being lost.

SERE OR Multiple Path

The SERE-OR construct is the same as a case statement that has a parallel case specified. This assertion checks that the cases really are parallel. This assertion fires when two or more paths are both legal. Since there is no priority encoding in this case, two legal paths would result in undefined device operation

This assertion fired as a normal part of block-level debug. In our test cases, it pointed to functional errors in the RTL, but not to any architectural problems. This was typically because the designer did not realize that a priority was required. This could have been a problem for a traditional RTL design if the parallel case pragma had been applied to the case statement.

SERE OR No Path

The SERE-OR No Path assertion is similar to having a default at the end of a case statement that reports an error. As with the multiple path assertion, most times when this assertion fired, it was simply a functional error in the block-level RTL. The one exception was an unexpected stall in a pipeline that was first uncovered by this assertion. This brought up an inconsistency in the functional specification of the device.

Unknown Condition

The unknown condition assertion fires any time that an equation that is used for a control-logic branching condition resolves as an unknown. In those cases, a simulator will choose a path that may be different from the physical device, which may mask errors in the design. The normal example is a branch that is done with an if/else statement: if (f(x)). If f(x) resolves false, then the else path is taken. In most simulators, the else path will also be taken if f(x) does not resolve to a known state. The unknown condition assertion fires under those circumstances.

In our cases, this assertion fired for mostly two reasons. The first was for registers that needed to be reset for correct behavior, but weren't. The second was for verification tests that should have initialized registers before starting operations. This was a convenient assertion since it reported the errors almost immediately, and finding the source of the problem was a trivial exercise.

Clock Domain Crossing

Unlike any other hardware modeling language that we are aware of, DesignPSL requires that all clocks be explicitly declared. (With Verilog the language has no actual notion of a clock. The clock is inferred by synthesis tools based upon particular usage.) As a result, the tool is able to drive an assertion any time a signal crosses a clock domain

boundary without the designer explicitly stating that the signal is expected to cross that clock domain.

This static assertion caught a few simple RTL bugs where the designer forgot to allow the clock crossing, as well as a few instances where a synchronizer was missing. It also caught an architectural specification error, where a clock boundary was overlooked for a critical communication path. It also caught a place where the designer did not know he was getting an input signal from a different domain.

Register Collision

A collision assertion occurs any time that two state machines attempt to drive a signal on the same cycle. Normally, the simulation language will simply accept whichever state machine drives the signal last. This behavior may be different from what the designer intended.

In all of our cases, this assertion fired as a result of incorrect state-machine design. These were block-level RTL bugs.

Comparison to Traditional RTL Debug

The best comparison metric we have is with the rewritten DesignPSL project, where an existing testbench was used with a new DesignPSL design. In that project, almost 80% of the bugs were found with assertion reports. The remaining bugs were found by error reports in the existing testbench. The high ratio of testbench reported errors may be due to the fact that a complete, debugged testbench already existed at the start of the RTL coding.

Far and away the largest difference was for block-level debugging. It appears that the assertions were able to catch greater than 50% of all known bugs by running a few tests that only had to start most of the state machines. While the debug process was significantly shorter for assertion-based error reports, we were not able to gather meaningful debug time statistics both for reasons we mentioned earlier, and because our test cases were not actually identical. Often the traditional RTL designs were already partially debugged, so the simple bugs were already fixed when we started our examination.

What we did see is that many block-level errors were fixed early in the debug process. Static assertions fired the first time that the code was compiled, which meant that those bugs were removed before the verification tests were involved. Almost 50% of the block-level assertion errors fired after running the first few tests. Again, that provided an early report of errors. As a result, we saw a high bug open and close rate early in the design/debug process.

One of our early surprises was with the number of assertions that fired in code that was supposed to be working already. In a number of instances, an assertion indicated that there was a functional error in logic, yet that logic was already running in a simulation, or in one case, in a production part.

One example of this is in a bus master device. After issuing a request, this device asserted an acknowledge, and then never de-asserted the acknowledge. When we first saw this, we were sure we had made a mistake, since this device was actually in a production part. Upon further inspection, we found a corresponding error in the bus arbiter, where a high-priority request would override a low-priority acknowledge. The faulty master never did long transfers, and so the error was undetected in the operation. Had this bus master or arbiter been re-used in another project, then an error would have shown up during a simulation or in a production part.

Coverage

DesignPSL generates automatic functional coverage assertions for a design. This includes not only coverage of all states and state machines, but also all transitions, sequences and paths in control logic, plus cross-correlations between state machines, loops, counters, and boundary conditions.

Only one project had coverage metrics from the traditional RTL design. In this project, both line coverage and functional coverage metrics were available from an existing test suite. A few of the coverage results are shown in Figure 6. These show the results of a test suite that had a significant amount of work put in based on the traditional coverage results. The DesignPSL in this project was a direct copy of the traditional RTL, so the accuracy of the comparison should be quite high. The functional coverage in the traditional RTL design was done by specifying specific functional coverage points, and then specifying the definition of each operation, and finally specifying the legal types of interactions. All of the DesignPSL coverage results were done automatically.

Traditional RTL		DesignPSL	
Line Coverage:	95%	State Coverage:	100%
Functional Coverage:	98%	Transition Coverage:	82%
		Repeat Coverage:	89%
		Cross Correlation:	46%

Figure 6: Comparison of Traditional and DesignPSL Coverage Metrics

The most noticeable difference for both the design and verification teams was that meaningful coverage information was available immediately and automatically. In our projects, it meant that designers switched from having no access to coverage information to being able to see what had been (or more to the point – what had not been) covered in early tests. In all cases, the designers would look at coverage results as a normal part of debug, and provide feedback where they thought coverage could be improved. That did not happen in any of the traditional RTL projects.

State and transition coverage were most commonly used in examining the design behavior. State coverage is similar to the coverage information available in traditional RTL coverage tools. All functional states are automatically identified, and coverage

information is provided at all times. For transition coverage, all possible legal transitions between states are automatically identified, and full transition coverage is available at all times during a simulation.

Early in the block-level debug, the state coverage information was used to reveal problems in both the RTL design and the tests when a state machine did not operate as expected. These were situations where either the test was wrong, or the RTL was not being exercised as expected. Here, the state coverage information was useful because it was easily available. More traditional debug methods would also have worked, but were avoided because the exact state where the problem occurred was visible through state coverage. The most common example of this was a sequence that did not go down an expected path at some branch point, either because of missing inputs, or an incorrect equation. In this type of case, no check assertion would fire, since there in no conflict in the RTL code itself.

Transition coverage was also used for test debug. The major differences are that transition coverage is more thorough a coverage metric, and that it is able to find some errors that would have been difficult to detect with traditional coverage tools.

Example of coverage issues found by transitions occurred in pipelined logic where stall logic existed but was not tested as expected, and a memory controller that never did back-to-back operations.

The pipeline stall issue was discovered early in the debug process because the transition coverage was expected to be 100% for that particular state machine, but wasn't. Because the information was available, the verification engineer found and fixed the problem early. While all states were covered, the stall had separate logic to keep data in their current pipeline location, and to keep state machines in their current states.

The memory controller issue is more interesting because it is an error that was not caught in the traditional RTL flow. The transition coverage showed that the memory controller always had at least one idle state between the last and first state. This turned out to be a problem with the memory transactor that was used in both the traditional and DesignPSL projects. In the traditional project, this error made it past code coverage because most of the logic was exercised in the transition from idle to the first state. The DesignPSL tool recognized the direct path from the last state to the first state as a separate legal transition, and could identify that the transition had not occurred.

While we expected that functional coverage would improve the design and verification, we were surprised that the debug flow changed as well. What we found when the engineers were debugging tests is that they would go to the debugger, and look not only at the current state, but also at the coverage information. The debugger shows both the current state of the design at the time of the assertion, as well as the coverage information showing all places that had previously been covered. As a part of the debug process, knowing if something has already happened, or what never happened, turned out to be

useful information. As engineers became more experienced with the debugger, this information became more important.

One of the results of having more in-depth coverage information was that, in all cases, there was a perceived need for much more high-level verification. In all cases, the DesignPSL functional coverage data showed significant uncovered portions of RTL code in the designs.

Comparison of RTL Results

In some cases, the RTL was directly transcribed from the traditional RTL with no modifications in the logic. These cases provided a 1:1 comparison of code sizes and coverage since the RTL was tightly matched.

For one project, the DPSL code was written from an architectural specification independently from the existing RTL. Where this happened, the RTL implementation diverged significantly. DesignPSL is well suited to writing and maintaining complex state machines. As a result, when comparing the DPSL to the traditional RTL, we noticed a significant difference in the implementations.

Of the blocks we examined, there was an approximate 80% reduction in RTL code, where the original RTL was written in VHDL, along with a 40% reduction in gate count and a 10% improvement in speed.

The reason that we found for the decrease in gate size appears to be partially due to how larger state machines are designed in the traditional flow. Designing a state machine in DesignPSL is quite straightforward. However, in the traditional RTL it was broken into several smaller state machines with fewer states, with register intensive communication logic between the state machines to synchronize them. While the implementation of the traditional RTL was not unreasonable, the synchronization logic was never entered into the DesignPSL design

Where the DPSL design was a direct translation of the traditional RTL, the differences in size and speed were less pronounced. The structure of state machines and interconnect logic were identical. Despite this, we saw an average of a 10% reduction in gates count through synthesis and a 5% improvement in speed. These are average numbers. One reason for this difference may be in the synthesis structure. For the original RTL, the FPGA synthesizers were able to find 80% of the state machines in the RTL. For the DesignPSL output, the synthesizers were able to find 100% of the state machines. Presumably, this helped the synthesizers during code optimization.

Project Results

None of our comparison projects were ideal matches, which would have involved two independent, parallel design teams of equal capability working from a single specification. We generally had an existing traditional RTL design already in place when the DesignPSL project was started.

Nonetheless, we believe that we were able to see significant patterns in how the design and verification processes were altered in the DesignPSL flow.

In the new DesignPSL designs, which are where bug-rate comparisons are more realistic, we saw significantly steeper bug open and close rates in the early phases of the project. This is where most block-level bugs are caught. Interestingly, approximately 40% of bugs reported by the assertions were not detected by the traditional testbenches. It was fairly common for the traditional testbench to report a passing test while the assertions were firing on functional bugs in the RTL code.

From all of our examined assertions reports, about 15% fired due to test issues, and the rest fired due to hardware functional errors. There were no false positive assertion reports in any tests. As has always been the case, we have no way to measure the number of false negatives.

The impact on the complete coverage of the verification suite was significant. Since we did not have a DesignPSL design in place during the traditional verification effort, we do not know how it would have affected the verification flow of early testing. What we did see was a re-assessment of system verification based on functional coverage. Average coverage results in the test suites that were thought to be completed were lower than expected in all cases. What was different was the ability to measure with some precision where the system environments needed improvement, along with specific cases that had not been verified.

Conclusions

In the projects that we have studied to date, we have seen what we believe is the first practical implementation of intensive assertion use within a design flow. What we believe was critical was that the engineers did not need to take on additional work to instrument the code or the test environment.

For the design team, the time cost of using DesignPSL was less than that of Verilog or VHDL. Designers were comfortable with the design process. Within a few days, the normal concerns about designer control over speed and power constraints abated. The major impact for designers was the new level of feedback from assertions. Block-level bug turnaround time was significantly reduced, and designers were able to engage in examining coverage results. Several designers spent time reviewing coverage and making suggestions about critical areas or scenarios that needed work.

For the verification team, there were several differences in the DesignPSL flow. During early debug of the DesignPSL, it appears that a single test caught more errors due to the check assertions, resulting in more efficient test productivity.

We saw a new ability for the verification team to measure and correct verification tests early enough to have an impact on the overall approach. This resulted in additional system-level tests, and modification of several existing tests when it was discovered that they didn't actually cover all of the conditions that they were expected to.

Future Work

An area we were hoping to explore was IP, due to the increasing use of SoC and reuse in design flows, and the reported 80% of SoC bugs being found in the IP. We expect that assertions may provide a significant advantage in IP, since they are not dependent on the higher-level environment. Where traditional verification methods often don't migrate well into a new design environment, we believe that assertions can be moved in parallel with a block of IP and be incorporated into a new design with little or no cost. That should provide a significant boost to the IP verification capability, and result in improved SoC design productivity.

Acknowledgments

This work has been a team effort with Robert Fredieu and Michael Concannon. I would like to thank both of them for all of their time and expertise.