

AUDIT YOUR DESIGN TO DETERMINE AND EVEN REDUCE THE AMOUNT OF RANDOM TESTING NEEDED

Dan Joyce, Staff Engineer, Hewlett-Packard Co., Austin, TX

Raymond Harlan, Project Engineer, Hewlett-Packard Co., Austin, TX

Ramon Enriquez, Staff Engineer, Hewlett-Packard Co., Austin, TX

Abstract

This paper describes a three step formal audit process which will identify designs that have a high risk of corner case bugs and will require a large amount of random testing to verify. The audit will also highlight areas of the design which are at the most risk. Methods are demonstrated which reduce both the amount of random testing needed to verify a design and the risk associated with corner case bugs showing up after tape-out.

Introduction

The verification effort and the number of test vectors needed to verify a design can be determined by looking at the combination of the **Directed** and the **Random** testing needed. The directed testing is easy to determine and is proportional to the size and complexity of the design. This is just testing that the design does what the specification says it can do.

Determining the amount of Random Testing required is much more difficult. Random Testing is used to find the Corner Case bugs in the design. These account for many if not most post-silicon bugs. [1] Random Testing is also the area of testing responsible for the recent exponential increase in test cycles needed to verify today's most complex chips.

This paper proposes an audit which will help to measure the amount of effort to verify that a design is free of corner case bugs. Exhaustive testing is a nice ideal, but most designs are too complex for that goal to be reached. This audit takes this into account and considers other ways to find corner case bugs.

Randomly Timed Stimuli

Corner Case bugs are caused when *Randomly Timed Stimuli* interact in the design. Each Design Under Test (DUT) has several externally generated events that stimulate the design. When two of these events can occur with random timing relative to each other, those two events are considered *Randomly Timed Stimuli*. To fully verify the device against Corner Case bugs, all combinations of timing between those two stimuli must be simulated. An example would be a memory read of the DUT as one event, and an interrupt of the DUT as the second event. If those events could occur with the "read event" first and the "interrupt event" 5 clocks later, or 20 clocks later, or 3 clocks before or any other timing, then all timing relationships of those two events must be simulated. *Exhaustive* testing would cover all of the timing combinations. Exhaustive testing is possible with 2 or even 3 randomly timed events, but the number of clocks of simulation grows exponentially as the number of Randomly Timed Stimuli are added.

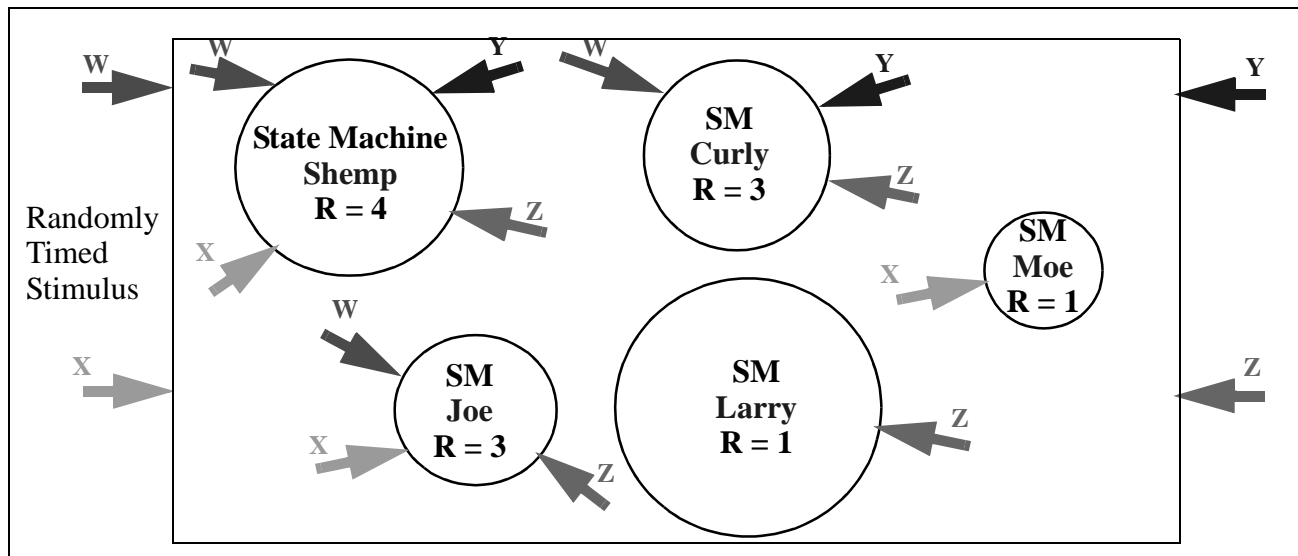
Figure 1 "State Machines affected by 4 Randomly Timed Stimuli" on page 2 shows an example design with five state machines affected by four Randomly Timed Stimuli. This audit is based on the

assumption that a large or complex state machine will have a large number of Corner *Cases* to verify. In addition, the number of Randomly Timed Stimuli which interact simultaneously in that state machine will indicate the effort needed to simulate all the Corner Cases in that state machine.

Note: We don't make any attempt to figure out how many Corner Case *Bugs* are in the design, we just assume all Corner *Cases* should be simulated to verify that there are no Corner Case *Bugs*.

From the figure, you can see that **Shemp** is a large state machine with 4 Randomly Timed Stimuli affecting it simultaneously. Therefore, Shemp has a large number of corner cases AND with 4 Randomly Timed Stimuli, it will take many, many clocks of simulation to exhaustively simulate all combinations of timing of those 4 stimuli. On the other hand, **Curly** is smaller and has only 3 Randomly Timed Stimuli affecting it simultaneously. So there are less Corner Cases to verify, and they are easier to verify. **Moe** is very small and has only one Randomly Timed Stimulus. **Larry** is very large but also has only one Randomly Timed Stimulus. Larry and Moe have no corner cases at all since there are no randomly timed interactions. Even though Larry is larger than Shemp, it has no risk to Corner Case bugs where Shemp has a high risk.

Figure 1: State Machines affected by 4 Randomly Timed Stimuli



Note: Randomly Timed Stimuli are not the same as Asynchronous Signals. Lots of Randomly Timed Stimuli exist in designs that are completely synchronous - even designs with a single clock zone.

Designing for Minimal Random Testing

One of the biggest challenges to ASIC design is the Asynchronous Clock Crossing. Bugs in asynchronous clock crossings are notoriously hard to find with simulation. Just as important, these bugs rarely have a work-around if found post-silicon. The typical approach to designing an Asynchronous Clock Crossing is to limit the logic dealing with signals from both clock domains to the smallest amount of logic possible. The next step is to use techniques proven to handle the possible problems (double register to avoid metastability, fifos for data transfer, etc.). Finally, the logic is thoroughly reviewed.

Randomly Timed Stimulus and the Corner Case bugs they cause are similar to Asynchronous Clock Crossings and the Asynchronous bugs they cause in that they are very hard to find in simulation and they are unlikely to have an acceptable work-around after tape-out. For these reasons they significantly add to the total risk in a design.

Designers should deal with Randomly Timed Stimulus and the Corner Case Bugs they cause in the same ways they deal with Asynchronous Clock Crossings.

To do this, the team must identify each of the randomly timed stimuli that affect the design. Then architect the design such that the amount of logic affected by those randomly timed operations is as small as possible. This will reduce the number of corner cases in the design, thus reducing risk. It will also reduce the amount of code that must be exhaustively tested, code inspected, or covered; thus reducing effort.

From a random testing perspective, a worst case design is one with many randomly timed stimuli, AND all logic in the design affected by each randomly timed event. This type of design would fail the Verification Audit described in this paper. Such a design would require a re-architecture to reduce the number of corner cases in the design before simulation and/or reduce the effort to simulate them.

Verification Audit

The following Verification Audit has been created to assess the risk of Corner Case Bugs in a design. The Verification Audit consists of three steps:

Design Overview.

Analyze the design to determine the random events that affect it. For each interface determine what random events stimulate the logic, e.g., read/write, requests. Include error conditions that introduce randomly timed events like time-outs on a master port, or aborts on a slave port. These should not include errors which do not introduce randomly timed events like parity errors.

Design Analysis.

Identify each state machine in the design. Determine how many random events affect each state machine simultaneously. If two Randomly Timed Events affect a single state machine, but never at the same time, corner cases will not be created. One event must affect the state machine while the other one is still being serviced. Also assign a relative size or complexity to each control logic structure. This could be a count of the number of states, or lines of code, but a better sizing would be the number of arcs in the state machine or the size of the binary decision diagram to show complexity.

This sizing effort is pretty simple, but determining the number of Randomly Timed Stimulus that interact in a state machine simultaneously requires the collaboration of the designers, since they possess the intimate knowledge of the design. An EDA (Electronic Design Automation) tool that does this automatically would be very nice to reduce the effort and chances of error.

Table/Equation Analysis.

Plug in the numbers for each control element that is affected by more than one random event. The value calculated in the equation will give an indication if it is a problem design.

$$\text{Effort-Risk} \approx S_1 \cdot (R_1 - 1) \cdot 2^{R_1} + \dots + S_N \cdot (R_N - 1) \cdot 2^{R_N}$$

- R_N is the number of random events that affect each control element.
- S_N is a rough size estimate of the control structure, relative to the size of all the control logic in the design (including control logic that is not affected by more than one random stimulus). Therefore it should be between 0.0 and 1.0 for each control structure and adding all control structures together should equal 1.0. Size can be determined by using a simple line count, state count, or arc count in each state machine.

An Effort-Risk value of less than 1.0 would indicate a relatively low risk design for corner case bugs, and should not require too much random testing. This indicates that design either has a very small amount of control logic affected by multiple random stimuli, and the logic affected by randomly timed stimuli is affected by only a small number. It would still be worth investigating state machines that contribute significantly to the total. These should be taken into account in the Test Plan. On the other hand, an Effort-Risk value well over 1.0 should raise red flags. All control structures which contribute significantly to the total should be examined for possible redesign. Again, the individual components in the equation will point to areas of concern, and the Test Plan should take these into account. At this point there is no hard rule on how low the risk number should be, or how much concern should be placed on a very high number. The equation should be used as a rough indicator which can point to problem designs and further point to problem areas in the design. Once those areas have been highlighted a more detailed analysis of the problem would take into account the specifics of the design.

See “Appendix: Effort-Risk Equation” on page 9 for a description of the derivation of the equation.

Methods to Redesign Logic

There are methods to reduce risk when the Verification Audit shows a design is at high risk to Corner Case bugs. These are some example solutions that have been used to reduce the amount of logic in a design stimulated by multiple randomly timed stimuli. Think of it as making the majority of the logic - especially complex logic - always perform the same boring operations. State machines that always go through the same few sequences will cause the least trouble and will get through code coverage the easiest.

Single Point Arbitration

If there are two or more randomly timed stimulus attempting to use a shared resource it is better to handle the arbitration in a single small state machine designed just for that purpose. The rest of the control logic that handles the request should be kept insulated from the multiple requests and this more complex logic should be designed to handle each requestor in the same way - if possible. This will limit the amount of logic affected by the random timing of the requests to just the arbiter state machine. It reduces the number of corner cases in the logic, reduces the amount of code to be inspected and reviewed, and reduces the number of lines of code to analyze with Code Coverage. See Figure 2 "Single-Point Arbitration" on page 5.

Isolate Aborts or Errors to Where They Occur

Isolate logic that deals with randomly timed error cases. One common source of randomly timed stimulus are aborts seen by “master ports”. This is where a port is being accessed by an off-chip master device. If the master aborts the operation in the middle or just drops off (releases the control lines) and goes away, a randomly timed event has occurred. In this case, if the request operation has already been sent out by the master port, just allow it to complete as normal. Do not propagate the abort with a randomly timed cancel operation that may interfere with the return operation. At the slave port, responses (normal or time-out) will be randomly timed events. But the response won’t interact with the request operation in progress as long as no cancel operation is allowed to propagate past the dotted line. See Figure 3 "Isolate Random Event Logic" on page 5.

Figure 2: Single-Point Arbitration

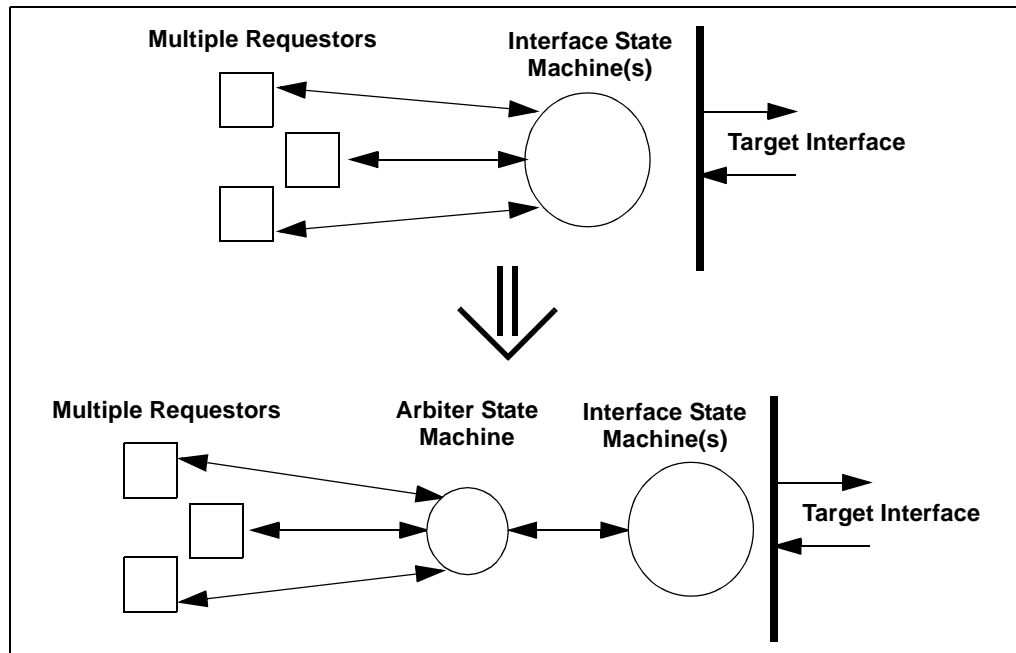
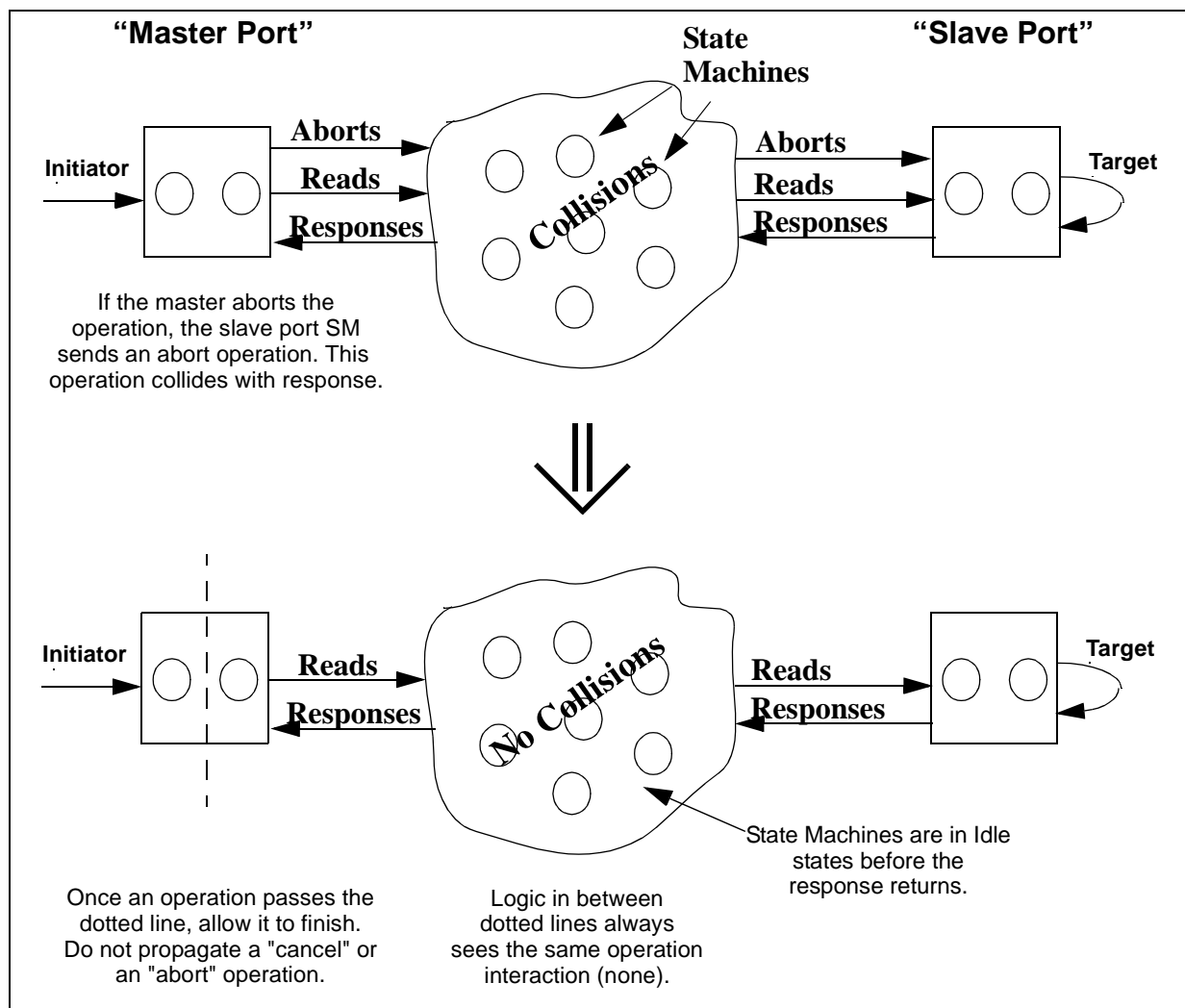


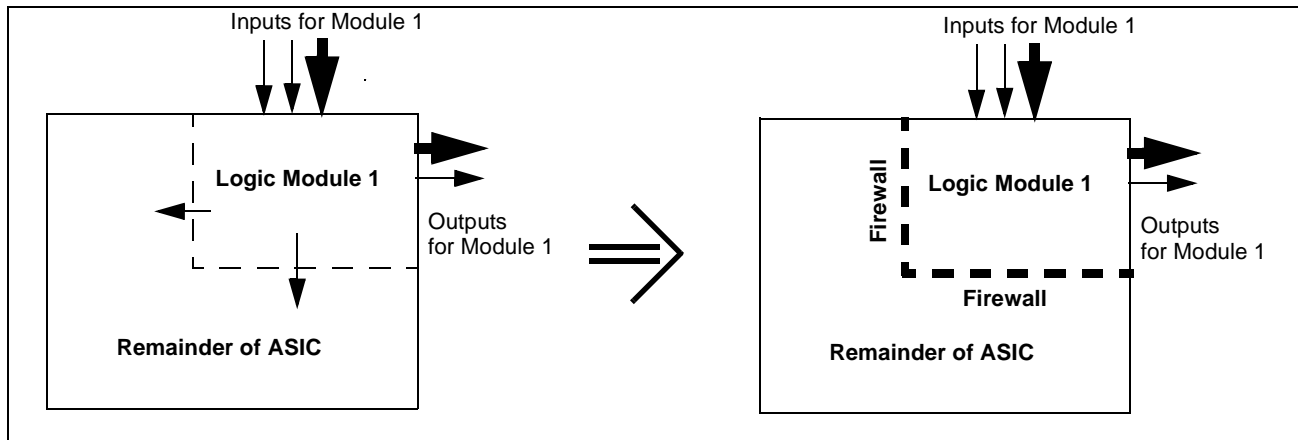
Figure 3: Isolate Random Event Logic



Logical Firewalls

Often multiple designs will be put on a single chip to reduce part count. Sometimes the need arises to communicate between the otherwise separate logic. Designers must keep in mind the cost in verification of doing this. Putting a firewall between the designs and keeping them completely separate will reduce corner cases because it will keep the randomly timed stimulus from one design from interacting with the randomly timed stimulus in the other design. If some communication is required, do it carefully as you would with an asynchronous clock crossing, i.e. very few signals handled carefully with very little logic. See Figure 4 "Logical Firewalls" on page 6.

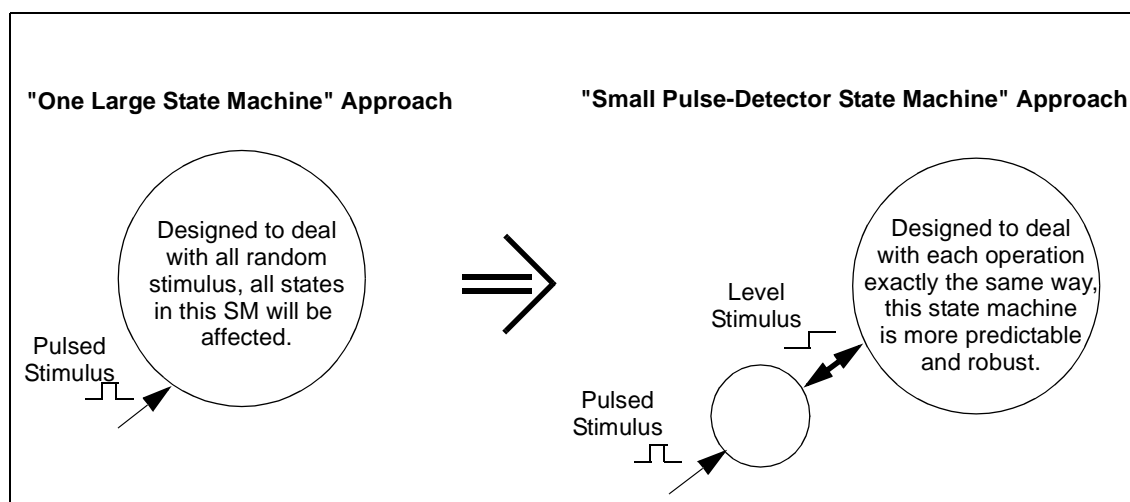
Figure 4: Logical Firewalls



State Machine Division

Often state machines have to “watch” for a pulsed signal (only asserted for a small number of clocks). One solution is to have each state check that signal and behave appropriately. This method can lead to a corner case for every state. Alternatively, use a small pulse-detection state machine (with a handshake to the more complex state machine for clearing the signal) to handle the “watching” of the signal. This allows the complex state machine to look for the signal at only a limited number of states. Less corner cases result in fewer test cycles, and less risk. See Figure 5 "State Machine Division" on page 6.

Figure 5: State Machine Division



Performance Considerations

Some of the redesign examples appear to have a performance impact to the design. Creating a smaller state machine to “shield” the more complex state machine from sources of Randomly Timed Stimuli can add a clock of latency to those paths. However, for cases where a clock of latency would impact the overall performance of the system, Mealy state machine design can be used. A Mealy state machine is one which has a purely combinatorial path for the paths that are critical to performance. This path would add an AND or OR gate to a combinatorial path, but not necessarily a clock of latency. Additionally, about half of the redesigns usually involve error cases. This is because error cases are common sources of randomly timed stimuli. However, error handling should never affect the performance of the system - unless the error happens frequently.

Methods to Find Corner Case Bugs

Corner Case testing is traditionally performed by having the Verification team create an intelligent test bench that can easily generate long sequences of legal stimulus. Verification languages like Vera and Specman, as well as old fashioned task based Verilog using forks and joins have been very effective in creating flexible, high level Bus Functional Models. These are created to crank out billions of clocks of test vectors with very little effort from the test writer. In the past it was easier to allow the simulation machine to work at getting into the corner cases of the design.

However, the ever increasing size of today’s ASIC designs [2]; coupled with the fact that HDL simulation speed bogs down at large design sizes [3]; the simulation speed (Clocks per Second) of large chips is slowing down. Add in the fact that some high level testbench languages require heavy PLI instrumentation of the design, which also slows down simulation speed, and it is becoming harder and harder to rely on the machine doing our work for us.

Exhaustive Testing

Ideally we would like to exhaustively test all the combinations of timing between all the possible types of Randomly Timed Stimuli. This is simply not possible with the designs of today. So we use methods to allow us to find as many bugs as possible within the time allotted. This audit shows which of the Randomly Timed Stimuli interact simultaneously somewhere in the logic. Therefore, a Verification team can start by trying to exhaustively test the timing between all Randomly Timed Stimulus that interact with each other in the logic.

Random Testing

Random testing involves using pseudo-randomly generated test stimulus. This method gets much of the corner case coverage very quickly. But some of the corners do not get exercised for way too many cycles. Methods to force the random testing into the corners often seem more like an art than a science. Good verification engineers are worth their weight in gold because they can see how the random testing is interacting in the design and know just how to tweak the tests to find the corners. Some code coverage tools are starting to provide ways to integrate code coverage information to the test environment. Unfortunately, this requires an extremely advanced test bench - and test bench architects that understand these complex tools and methods.

Multiple Simulation Environments

Using multiple simulation environments involves breaking up a design into smaller pieces. Instead of one large simulation environment testing the entire chip, the chip is broken into several very

large pieces and each piece is verified using its own environment. This lowers the size of the Design Under Test (DUT), and therefore raises the simulation speed allowing more effective exhaustive and random testing. But it significantly adds to the amount of work for the Verification team. Bus Functional Models must be written and maintained to emulate the functionality of the other parts of the ASIC. Many bugs found in the DUT are not real bugs at the ASIC level, but are only due to the way the BFM's are stimulating it. Most importantly, since a chip level test environment is still required in addition to all the sub-chip environments, verification tasks are often duplicated between the separate sub-chip simulation environments and the chip-level simulation environment - thus wasting Verification resources. [4]

Code Coverage and Assertions

Code Coverage Analysis is a very effective way to find corners that have not been tested. It is very cost effective if used correctly. [5] However, there are limitations. First, Code Coverage Analysis tools, when enabled, slow down the simulation speed for those runs. Second, Code Coverage Analysis will show parts of your design you did not cover in your regressions, but it will not show you everything. At the very least, teams should use Code Coverage Analysis to make sure all state machine states have been entered, and all state machine transitions have been traversed. But there are many Corner Cases which involve more than just that level of coverage. An example would be a scenario where one state machine is doing one thing while a second is doing another particular operation at the same clock. This event will not be tracked by standard Code Coverage tools unless someone has inserted a custom coverage point for just that scenario. Third, many teams get bogged down with Code Coverage. It can become a very difficult hurdle if the goals are not clearly stated and followed. There are always some lines or conditions which are impossible to test - or not worth the effort to test. If the team is able to intelligently decide which lines or conditions are not worth the effort and has a process to sign off certain coverage points and archive the decisions, then the cost of doing code coverage analysis is kept in check. Usually Code Inspection of those coverage points and the logic around them is used as an alternative.

Observable Coverage is another recent improvement that only enhances the power of Code Coverage Analysis. [6] This process further refines the coverage feedback to show only coverage that actually causes visible changes at the periphery of the Device Under Test. However the price is an additional slowdown of simulation when running with the coverage tracking tool enabled. This again reinforces the need to reduce the random testing needed to verify a design. Code Coverage Analysis has another very relevant attribute. It will often illustrate the purpose of this paper since designs that fail the audit will typically have very hard state machines to fully cover.

Assertions and Custom Code Coverage flags are used to give the design and verification team additional visibility where automatic Code Coverage instrumentation falls short. These can be very effective but also require effort to create.

Formal and Semi-Formal Tools

Semi-Formal tools are becoming available which claim to search out the corner cases in a design and find the bugs using a formal or semi-formal approach. Unfortunately these tools have significant limitations. The primary limitation is a limit to how much logic they can work on at a time. They also require rules at the interfaces to tell the tool what stimulus is required. Hopefully, some advances in the near future will make these tools less effort and therefore more cost-effective. It would be nice if they had the ability to: hierarchically partition a design as specified by the user to focus on a manageable design subsection, extract the interface rules automatically from the surrounding logic, and apply the formal routines on the smaller subsection to find the corner cases.

Conclusion

In this paper we have discussed the growing risk of corner case bugs and how they are caused by the interaction of multiple sources of *Randomly Timed Stimuli* interacting in the same logical structures. We described how to design logic “up front” in a way that reduces the effort to verify it for Corner Case bugs. We described a Verification Audit that can be performed if you have intimate knowledge of the design. We then described several methods to improve the score in the Verification Audit, and ultimately reduce the risk of a corner case bug and reduce the amount of random testing needed.

While the Effort-Risk calculation provides a number which can be used to highlight problem designs and even point to the areas of concern in a design; it is only based on general characteristics of the design. When a red flag is raised by this audit, more detailed analysis is required. This analysis should use the specifics of the design and its use in the system to determine the real risk and how best to deal with that risk.

Appendix: Effort-Risk Equation

Our equation is an attempt to measure two things at once: the effort to verify that the logic has no corner case bugs; and the risk that a corner case bug gets through to silicon. To make such an audit cost-effective, some simplifications were made.

Effort-Risk Calculation

To *exhaustively* test a control structure that is affected by 2 randomly timed operations, the test bench must simulate those two randomly timed events as they pass across each other in time. Starting with one occurring first in time, some dead time, then the second occurring. Then the two are simulated again and again with the second coming one clock earlier, relative to the first, until the second operation is actually starting and completing before the first one starts. If the length of the longest random stimulus is L_1 clocks and the length of the shortest is L_2 clocks, we can determine the number of clocks needed to exhaustively test all corner cases of the two randomly timed stimulus (the number of simulation runs times the average number of clocks per run). It will require X simulation runs where $X = L_1$. The number of clocks of stimulus per run will vary between L_1 and $L_1 + L_2$. So the average number of clocks per run will be $Y = L_1 + 0.5 * L_2$. The number of clocks to exhaustively test the interaction between these two randomly timed operations is $Z = X * Y$. Substituting for X and Y gives $Z = L_1 * (L_1 + 0.5 * L_2)$. If we simplify the calculation by assuming all the randomly timed operations are about the same length and normalizing them to a single value of L we get $Z = L * 1.5L = 1.5L^2$. Adding a third random event of the same length gives us $2.5L^3$. Four random events gives $3.5L^4$. This pattern fits the equation $(R - 0.5) * L^R$ but we approximate it with $(R - 1) * L^R$ where R is the number of Randomly timed stimuli affecting that control structure (see bullet #4 in “Caveats and Future Research” below for a discussion of the approximation).

However, this paper is advocating methods to reduce risk to corner cases by methods other than exhaustive testing, when exhaustive testing is impractical. To represent these improvements, and to reduce the effort of the audit, we replace the L with 2 (see bullet #1 in “Caveats and Future Research” for this discussion). So the effort to verify a single control structure with R randomly timed stimuli, using more options in addition to exhaustive simulation can be represented by the quantity $(R - 1) * 2^R$.

Size of the control structure - whether measured in gate count, state count, line count, arc count, or size of the binary decision diagram, will have a direct relationship to risk of corner case bugs because a larger design will typically have more corner cases. In addition, the larger the control structure, the

more effort will be required to do code inspection and the more lines of code to be Code Covered. Therefore, to represent risk and effort, each control structure has a size element multiplied to weight it.

To do this we determine relative sizes for each control structure. We normalize those size multipliers so they will add to 1.0 for all the control logic in the design. So if a control structure is 1/10th of the size of all the control structures (including those only affected by only one randomly timed stimulus), the size multiplier for that element (S_N) would be 0.1.

So the effort to exhaustively test AND/OR use alternative approaches to verify - and the risk of corner case bugs getting past the verification of the design is roughly proportional to $S_N * (R_N - 1) * 2^{R_N}$ for each control structure. The cumulative effort for the whole design can be estimated by adding together all these quantities using the equation:

$$\text{Effort-Risk} \approx S_1 * (R_1 - 1) * 2^{R_1} + \dots + S_N * (R_N - 1) * 2^{R_N}$$

Caveats and Future Research

In any attempt to quantify an assessment with a real number, there is a risk of placing too much importance on the number generated. It is more important to look at how the design attributes are affecting each component of risk and look for ways to improve the result with design modifications. At this point we have not performed a study to evaluate the relative size of the Effort-Risk number to bugs found in chips after the verification phase. We have not compared the Effort-Risk number to random testing on previously shipped chips. This would be a great opportunity for future research. The equation seems to roughly indicate the amount of risk we have witnessed in previous projects. But the simplifications we used in generating the equation should be discussed.

- The one simplification which has a *major* impact to the result was removing the requirement to determine the length of each randomly timed stimulus and replacing it with the number 2. For cases where some of the stimuli are very long, this may be an oversimplification that could hide some real issues. But more importantly, this simplification dramatically reduces the impact of control structures with a high number of randomly timed stimuli (say 4 or more). But remember we started with the equation to calculate the number of clocks to *exhaustively* test a control structure. However, this paper advocates use of many other techniques where exhaustive testing is impractical. If we wanted a number to reflect the effort of exhaustive testing, L would be more appropriate. To illustrate this point, a single state machine that is only 1/20th of the control logic, affected by 4 randomly timed stimuli - each with a length of 40 clocks, would contribute a value of $(1/20) * (4 - 1) * 40^4 = 384,000$ using an exhaustive calculation. However using 2 for the length (to account for additional tools used to verify against the corner cases that the randomly timed stimulus cause) gives that control structure a Effort-Risk value of $(1/20) * (4 - 1) * 2^4 = 2.4$. Depending on the effectiveness of your additional tools, you may want to use a number larger than 2. If you are not using any additional tools to help reduce the risk of corner case bugs, L may be a more appropriate base number.
- In the calculation of the size multiplier, we normalized to a total size of 1.0 for all the control logic. This means a very large chip design will tend to have an Effort-Risk value about the same as a very small design - except for the fact that larger designs tend to have more interfaces, and hence more sources of randomly timed stimulus. For this reason this audit will not calculate the number of man-months of effort do verify the corner cases in a design. Perhaps a total design size multiplier could be used for this purpose - suggesting another area for future research.
- The Effort and Risk components could be broken out and calculated separately. We decided not to do that for simplicity and cost-benefit reasons.
- The $(R_N - 1)$ multiplier in the calculation of each logical structure was an approximation of $(R_N - 0.5)$. The multiplier $(R - 1)$ made more sense when for the cases when only one randomly timed event is affecting a piece of logic. No random testing is required because the risk to corner case bugs should

be 0. Random testing is only needed when 2 or more randomly timed events interact in the logic. Using $(R_N - 1)$ zeroed out the component of any design element with only one randomly timed stimulus. Purists are free to use $(R - 0.5)$, and just not include any control structures with less than two randomly timed stimulus. The difference to the total will be *very* small.

Acknowledgements

The authors would like to acknowledge the ASIC and FPGA Design and Verification teams of the Hewlett Packard Non-Stop Enterprise Division. The integration of the design and verification processes in the architecture of designs has led to very successful ASIC and FPGA implementations. This integration, specifically the iteration between the architecture of the designs, and input from the verification teams, was the inspiration for this paper.

References

1. "Newbridge Networks Adopts a Top-Down Verification Strategy" by Dan Schumacher; <http://www.eedesign.com/editorial/1998/topdown9806.html>
2. "Design Methodologies for DSM ASIC designs" by Ravi Thummarukudy; <http://www.eedesign.com/editorial/1999/designmethod9903.html>
3. "High Level Verification Language tools throttle performance of simulations when linked with RTL simulators through PLI." by Tom West (twest@broadcom.com); <http://janick.bergeron.com/guild/3-15.html>
4. "Nuts and Bolts of Core and SoC Verification" by Ken Albin; http://www.sigda.org/Archives/ProceedingArchives/Dac/Dac2001/papers/2001/dac01/pdf/files/16_2.pdf
5. "Verification of Configurable Processor Cores"; by Marín'es PuigMedina, G'ulbin Ezer, and Pavlos Konas; http://jamaica.ee.pitt.edu/Archives/ProceedingArchives/Dac/Dac2000/papers/2000/dac00/pdf/files/24_2.pdf
6. "Code coverage techniques -- a hands-on view"; by Alain Raynaud; <http://www.eedesign.com/features/exclusive/OEG20020912S0059>