Focused Random Validation with BURST

Kent Dickey, President, ProValid, LLC

January 2005

This white paper describes ProValid's leading-edge chip validation tool BURST™. BURST performs automated validation of FPGA and ASIC-based systems. It can be used for pre-silicon or post-silicon testing to increase the overall system quality.

The first section gives a summary of current verification technology and lists some shortcomings with current popular verification approaches. The second section describes BURST and how it addresses these issues.

1. Current Industry Verification Technology

Computer chip designs strive for a defect-free product. Defects in a shipping product can be extremely costly due to product returns, liability, and lost customers. For most complex designs, more resources are spent ensuring the design is correct than are spent creating the design.

Testing for system correctness is critical for product success. The standard functional checks performed are called *verification*. Verification checks that the chip operates according to its specification. ProValid defines a stronger standard for correctness called *validation*. Validation ensure that the desired chip or system works as intended. For example, the chip specification may be incorrect relative to a larger system specification. Or the chip specification might not be complete and not describe necessary behavior to avoid deadlocks or correctly order transactions. Validation is a superset of verification. A major issue is that many companies only perform verification tasks. This leads to reduced coverage and increases the chances of missing critical bugs.

Pre-silicon testing attempts to discover most chip bugs using simulators before hardware is created. Using RTL simulators is an important step for ASIC and FPGA designs. Debug of simulations is much easier since most internal chip signals can be viewed. Chip complexity creates some bugs too complex to find using the limited simulation time available. Experience shows that some defects such as timing races, hold time problems, or a large class of chip and board manufacturing issues are not easy to find using common simulators. Post-silicon testing then attempts to catch these remaining bugs by running tests on the completed system at operational speed. Once this testing completes, the product is shipped.

1.1. Standard Verification Techniques

A well-used approach to verification is to run hand-written directed tests. These tests are usually described by designers in a large test list and then crafted by a verification group to ensure the listed functionality works as specified. Writing these tests is tedious and slow and requires a large team to finish the tests on schedule. This approach has historically provided the lowest coverage and allows many complex bugs to slip through testing.

As an improvement, random test tools are now commonly used for chip verification. A random test tool uses random numbers to guide generating a test case, allowing for good coverage of

targeted functionality. By “random”, it is implied that in general these tools use pseudo-random numbers which are generated by a formula, but the term random is commonly used to mean pseudo-random. These existing test tools help find many chip bugs and are a standard part of most test plans. However, many random test tools can still miss serious and critical defects in chips due to tool limitations. Many bugs are missed due to complex interactions among various components that may not be well exercised by simple random tests. Also, the random tool may be too unfocused and not be an efficient exerciser of the chip. It wastes time testing uninteresting variations of what has previously been tested.

An example of a simplistic random generator is to test a CPU by generating random 32-bit values, placing them in memory, and then executing them. Chances are that almost all tests will end after a few instructions with an illegal instruction, or take a branch into a bad memory address. This is an inefficient testing strategy since billions of tests would need to be run to have a reasonable chance of exercising even basic functionality. Although no actual tool would be so simplistic, many tools do fall into the same problem. The tool's coverage hits a limit after only a relatively small number of tests and practical time limits prevent these tools from finding further bugs. A related issue with random test generation is a simple rule of probability: if a particular CPU instruction has a 10% chance of being generated, the chance of generating nine in a row is a one in a billion chance. This makes simplistic random generators unlikely to fill chip queues.

Another common approach is template-based random testing. When testing a CPU, the tool uses sequences of 2 to 8 instructions that were predetermined from a table. The random generator just splices multiple templates together. Coverage is now limited by the templates. If a bug requires an instruction sequence not in a template, it is impossible to hit. These generators can easily generate nine of the same instructions in a row, which addresses some of the simplistic generator issues, but the template limitations create new coverage holes. These generators tend to achieve their maximum coverage fairly quickly and then cannot find further bugs.

Another drawback for some random test tools is managing “knob” files where each knob controls the probability of some aspect of the test. Engineering resources are required to craft knob files, generate tests, and manage the knob files for regression testing. Knobs are a way to help guide the random testing tool. To test a particular area, changing the knobs can help create more targeted tests. Unfortunately, knob interactions can be complex and not well documented in the tool. Changing the knobs without this understanding may create holes in the test coverage rather than test new areas. Managing the knob files takes engineering resources, too. It would be better for the user if they did not have to fiddle with the knobs.

1.2. Pre-Silicon and Post-Silicon Validation

The earlier a bug is found, the easier and faster it is to fix. Finding a design flaw soon after the RTL (logic source code) is written leads to extremely fast fixes for very low cost. Pre-silicon simulation thus focuses on giving quick feedback on simple test cases. Random test generators are often used, but since debug time is so critical, they usually generate fairly simple tests so that failures can be debugged quickly.

Once chips and boards are manufactured, post-silicon testing begins. Many sources of bugs other than logic flaws are possible, so stressful testing of the final system configuration is critical to ensure product success. Post-silicon testing ensures the system meets reliability and manufacturing quality goals and is a necessary final check on all pre-silicon system assumptions.

At a minimum, post-silicon testing runs applications in various system configurations and checks that the system operates correctly. For computers or embedded systems, this often means booting an OS and running the expected applications with peripherals attached. This testing will often

find bugs, but the overall system coverage is limited. Some hardware failures can be masked by the layers of software, making bug detection very difficult. And since the application code is relatively static, once basic testing is complete, simply running longer tests is less likely to find more bugs. This testing also leaves the product vulnerable to exposing chip bugs when future application updates occur.

It is important for a post-silicon test plan to include testing beyond running the expected applications to ensure the hardware defects are discovered before a future application update exposes them. Debugging complex applications is a difficult and time consuming process. It is much more efficient to do chip validation with a dedicated testing tool.

1.3. System-Level Testing

Many verification tools address just one area at a time. There are floating-point tests, network packet tests, CPU integer tests, memory pattern tests, etc. Although each test can be effective at finding some bugs, there are bugs at the intersection of these areas. ProValid has discovered many bugs where a chip flaw is revealed only when multiple chips in the system are being fully exercised simultaneously.

To find this class of bug, a random test tool must get all components in the system doing work simultaneously. This means the CPU, memory, peripherals, and all buses need random but coordinated traffic all at the same time. Just running a memory test while running a network packet test is not sufficient. For example, the failure may only occur if the CPU and network device access the same memory locations, which will not happen when running independent tests. The tests need to interact to find the subtle ordering bugs, deadlocks, or queue full flaws.

To find failures caused by manufacturing process variations, temperature change, or excessive noise on buses or power supplies, it is important that testing include all of the system. To generate maximum noise on the power supplies and grounds, all chips need to be busy. Data patterns need to be carefully selected to increase the noise, since again simple random values are not the worst-case patterns for generating noise.

2. Validating Systems Using BURST

ProValid has developed an industry-leading chip validation tool named BURST. It addresses the limitations of current verification practices and provides greater overall system coverage. BURST (**B**ug-finding **U**sing **R**andom **S**tress **T**ests) applies advanced validation principles to generating effective random pre-silicon and post-silicon tests.

BURST excels at system validation. Teams implementing a verification test plan often focus at too low a level by closely following a chip specification when creating tests. Validation has a broader focus and creates tests based on how the system should behave.

2.1. BURST General Operation

BURST generates random test sequences which stress the whole system. For post-silicon testing, BURST runs completely on the system to be tested and is booted like an OS. For pre-silicon testing, BURST runs on Linux (or any Unix-like system) as a normal application. It prepares simulation stimulus files to pre-load system state and then starts the simulator. When the simulator ends, the output files are read back in and BURST checks the results. BURST is a self-checking tool which will check for its own correctness by calculating the expected final system state. The same code for test generation and checking is used for both pre-silicon and post-silicon testing.

BURST consists of driver modules which create the test for that component. One module generates CPU instructions and each supported peripheral device has a driver module. Modules can interact and use a shared memory space to allow modules to create contention with each other. In general, most aspects of the test sequence are randomized, including addresses used, amount of memory used, number of instructions or transactions, etc.

After each test is run, the final system state is checked against BURST's predicted values. If anything mismatches, BURST provides copious debug information to aid debugging. If the test passes, BURST begins generating the next test automatically. By being autonomous, BURST executes quickly by running thousands of tests per second in post-silicon testing.

Like many other random test tools, BURST has knobs to control how features are tested. However, BURST's knobs are intended for special circumstances and they are not changed for normal testing. By removing the need to adjust the knobs, testing uses hardware resources around the clock and engineering resources can be spent on debug or developing tests for new features. Automation also enables a small team of less experienced engineers to efficiently utilize a large quantity of hardware. This accelerates the schedule by allowing testing to conclude sooner.

2.2. Focused Random Validation

Many random test generators have common coverage holes. The two main issues are: being too random and so are slow to exercise complex cases; or using templates which restrict the potential coverage.

A better approach is called focused random testing. Special random algorithms ensure that the most interesting functions are tested the most. When applied to testing a CPU, a focused random tool such as BURST generates instructions one at a time without using a template. Instructions are chosen with register values selected to avoid execution exceptions. Care is taken so that there are few if any restrictions on the possible instruction sequences. The focused randomness will create a series of the same instruction much more often than the usual randomness would indicate. This provides faster coverage of complex cases.

BURST also uses multi-level randomness to create more useful tests. Each test will determine certain modes randomly once for this test, and then craft the rest of the test within these modes. This is effectively the same as having the tool randomly turn knobs which eliminates knob fiddling. All of the BURST knobs are pre-set to reasonable values, so everything has a chance of occurring. The multi-level knobs ensure that the interesting cases get exercised much faster than randomness would normally allow.

2.3. BURST Portability and Reusability

A sophisticated automated validation tool requires significant resources to develop, often requiring expertise not available in most organizations. When designed well, the tool will consist of modules that can be added or removed to allow porting to new architectures and configurations. This leverages most of the previous work forward to the next project.

BURST is modularly designed so that components can be enabled or disabled to fit the system to be tested. The CPU instruction generator is separate from the peripheral card drivers and other BURST infrastructure code. This allows BURST to be ported to new architectures with a minimum of work while being able to reuse most of BURST's existing functionality.

Configuration testing is also a key factor in stressful system tests. Systems need to be tested with various peripherals present in order to ensure high coverage. BURST automatically detects peripherals present allowing quick hardware changes to occur without needing to continually recompile the tool.

An automated random tool also solves the reusability problem inherent with directed tests. Hand-written tests tend to rely on specific cycle timings to hit the desired test cases. Although the tests may run on the next generation platform, they no longer are increasing coverage or hitting cases of interest. But with an automated random tool, coverage on new systems stays high, even if the system underwent significant changes. Work can be focused on supporting new features to increase coverage. This high leverage reduces the effort for validation throughout generations of products.

By focusing on validation, BURST is less closely tied to chip implementation details. BURST achieves high coverage by exercising how the system is intended to operate, rather than the details of how the implementation works. This abstraction enables most BURST code to be reused when validating on future similar systems.

2.4. BURST Debug Features

BURST is designed to find design flaws and expects failures to occur. Unlike applications which assume the hardware is working correctly, BURST defensively checks continuously for correct operation. BURST therefore detects failures soon after they occur and provides information about the system state at the time of a failure.

Since BURST's purpose is to detect failures, it provides extensive debug information after a failure is detected. Every module in BURST keeps a detailed record of what the test was expecting to accomplish, what the wrong answer was, and many other test details. Many bugs can be debugged to the failing hardware component without needing a logic trace just by analyzing BURST's debug information.

BURST makes each test repeatable and independent. If a given test fails, it just takes a few seconds to re-start BURST and run just that test again, even if the failure occurred after the machine was running for weeks. The test will behave the same way when re-run and so the failure can be reproduced to help get a trace of the failure or to test potential fixes.

BURST also has features to aid in capturing traces in post-silicon testing. It creates logic analyzer triggers to allow easy capture of a failure. BURST creates tests whose traces are designed to fit within the storage capacity of the analyzer.

2.5. Automation Reduces Resources

Test automation maximizes two critical resources during validation: engineering hours and hardware utilization. Using an automated validation tool, engineering time is spent doing productive work such as developing code to stress new product features or debugging failures. It is not spent tediously preparing tests on machines. Automated validation allows testing around the clock without requiring engineers to work multiple shifts. This increases system utilization.

2.6. Successes Using BURST

ProValid has shown that BURST is highly successful in finding bugs. BURST has found bugs on systems which had already been released into production, but which had not run BURST before. Companies not using automated random validation techniques for their complex designs are potentially releasing products with bugs that could have been discovered with better validation. BURST has been the only tool to find some bugs in CPUs, peripherals, and other IP after full testing had already been completed. For example, BURST generated the worst-case pattern test for a marginal DDR SDRAM memory problem and found a hold-time bug in a DDR controller. In the first case, months of Linux testing with applications had not discovered the problem.

BURST can be debugged faster than running applications. An embedded computer running Linux had disk failures that were being actively debugged for weeks with little progress. BURST was applied to the problem and in less than a day, PCI traces showed a defect in the PCI host bridge.

3. Conclusions

Complex chip designs need to use the best validation tools available to find the tough bugs. Not only does BURST find bugs that other tools miss, it provides investment protection by migrating forward to new systems and more efficiently using your testing resources.