The art of debugging

By Samek Mokryn

Introduction
The debugging of complex systems is the most challenging, frustrating and time-consuming part of the system design process. Systems like FPGAs (Field Programmable Gate Arrays) with internal embedded processors present a special challenge due to vastly increased complexity, very limited visibility to the internal operations, and the heterogeneous hardware/software nature of the designs.

The word "Art" is used here in its very basic meaning: a branch of activity, using a special medium and technique that do not rely exclusively on the scientific method. Not scientific in engineering? This sounds contrary to common sense. But, it turns out that the debugging process requires not only a lot of knowledge and dedicated tools, but also instinct and imagination on the part of the tester. This happens due to the lack of complete information about the source of the error, as is almost always the case.

The cost of system malfunctions versus the cost of testability (or lack thereof)
As often as the cost of system malfunctions is recognized and acknowledged, just as often the cost of lack of system testability is underestimated. In a fast-paced world, the emphasis is on reaching the project's target, rather than on mundane testing, and certainly not on dedicated hardware for error detection and correction. On the other hand, errors detected late can vastly increase the cost of support, decrease product acceptance in the marketplace, or even result in project cancellation.

In a constant drive to reduce costs, testability is often the first victim, shifting the total cost of the system from an initial NRE (non-recurring expenses) to the post-delivery support cost. The amount of initial cost savings is measurable; the future cost of support or product failure is not.

A closely related (motivated by the same reasons) problem is created by the selection of universal hardware/software platforms as the basis for a dedicated system. The reason is that such a platform, by being universal, cannot support the variety of dedicated system requirements (like various block checksums, redundant paths or dedicated error detection mechanisms). Testability is compromised yet again, while due to the lack of knowledge about the operations of these universal (mostly external) components, error determination is made even more difficult.

System malfunctions
There are various reasons that can cause a system malfunction:

1. Designer's errors;
2. Omitted states or state transitions;
3. Interface problems;
4. The physical behavior of underlying systems and technologies;
5. Defective components.

Designer's errors

Most of such errors happen early in the design process and some are relatively easy to spot. They can be as simple as a wrong expression, or they can be as subtle as a misunderstanding of the problem at hand. The first problem can be detected in simulation, while in the latter case no simulation will help if the simulation model is built upon the same assumptions as the design. There are many reasons why there can be misunderstandings, some of them not necessarily the designer's fault, (it is possible to write many articles about this problem alone).

One, rather common subclass of the designer's errors is wrong system behavior in boundary conditions, like counter overflows, system or subsystem resets in various operational states of the system, etc. Some of these problems cannot possibly be addressed in the schedule and budget driven design process and start manifesting themselves long after the product is delivered to the customer.

Omitted states or state transitions

In a non-digital world the other name for this error source can be "a problem complexity not fully recognized." In a simpler, digital world, the number of states of any complex design comprised of thousands of flip flops and millions of memory cells is larger than 2^10^9 (2 to power of n binary elements). And even this model is too simplistic, given non-zero transition time (i.e. in-between states), race conditions, metastability and other effects. It is impossible to cover all the state space during the design process, though the vast majority of states are "don't care" or will never occur. The problem arises when they do happen, often very late in the product lifecycle (on the customer site).

Interface problems

Interface problems are a common source of errors. If the remote system does what the designer didn't expect, the result may be unpredictable. Often, various systems respond differently to the same events. There are also possible secondary errors, i.e. if one system malfunctions then it sends improper responses causing the next system to malfunction. If there is no proper monitoring system in place, it is often impossible to determine who is at fault.

Interface problems can be regarded as a state space that was vastly expanded, while part of such space is not under the control of the designer.

There is always the desire to define interfaces as precisely as one can do. It can be achieved when the interfaces are simple and there is a strong separation between communication systems (a very small number of well defined common states). However, in the case of more complex or less precisely defined interfaces, where the word "shall" is replaced by the word "may", problems can be expected.

The physical behavior of underlying systems and technologies

There is a common assumption that modern chips are inherently reliable. But in reality these are complex analog circuits that are subject to many physical effects that can cause system malfunction. These effects can be both internal and external.

Internally generated errors

Internally generated errors can be induced by effects like thermal noise, variability in path delays (race conditions), metastability effects, line reflections, and most commonly, internally generated noise. A common CMOS technology consumes almost all energy in state transitions, and due to the very large density of modern chips and synchronous designs, thousands of flip flops can switch almost simultaneously. The resulting current spikes can easily exceed 100 Amperes or more, causing an internal "ground bounce" that in turn can cause a wrong state transition or an inadvertent state change. Also, due to an inherent capacitive coupling, these transitions can induce an undesired state change in neighboring flip flops as well. Interestingly, FPGAs are more prone to these effects than ASICs, since the noise distribution is a function of the design, unknown to the FPGA manufacturer; hence the preventive action that can be taken to avoid such errors is inherently more limited.

Externally generated errors

Increased density levels and lowered power consumption per bit result in a lower energy required to change the state of a memory cell or flip flop. Hence, any external energy delivered to the circuit can change the behavior of the system. This energy can come in a form of electromagnetic radiation, electrostatic discharge, alpha particles radiation, power noise fluctuations or induced noise. Even though most chips are designed to provide some defenses against these effects, these defenses are not perfect.

The sensitivity to disturbances, both internal and external, varies from chip to chip, since no one chip is identical to the other. Furthermore, the effect of these disturbances depends on the particular state of the design.

Since seldom (today almost never) do the designs incorporate a detection mechanism for errors generated by an unpredictable source, errors like these can go undetected and cause a system malfunction that cannot be explained.

Defective components

The component vendors prescreen devices for errors. Later in the process, the system vendors are testing their products (some more than others) for defects. The common methodology for determining if the component is defective is a comparison test; if the test is successful in the system containing one component, but is not successful in the same system containing a second, equivalent component, the second component is deemed as defective. Such simplistic testing seldom attempts to determine the real reason for the failure. As a result, errors generated by other effects, mentioned above, are often classified as component defects and subsystems containing the suspected components are discarded. This is a pretty expensive way to solve the problem and sometimes it is a futile one.

Since physical properties of materials change with time or can be affected by external conditions, components can become defective on the customer side, where the testing

capabilities are by far more limited than in the manufacturer's lab. The suspected subsystems or even whole systems are than replaced indiscriminately, sometimes to find later that the problem repeats itself.

For one reason or another, the system errors are here to stay. The question is if the level of errors is acceptable to the user, or if systems are built in such a way, that they can continue to operate in the presence of errors. But this is a completely separate subject.

The basics of debugging

The common methodology of finding the source of error consists of the following steps:

1. Collecting the information about the error state, and if possible, the history of how the error state was entered;
2. Determining the minimum subset of the system and input conditions that are involved;
3. Evaluate the selected subset for various error sources;
4. If step 3 finds a definitive reason for the error, take the appropriate action that solves or avoids the problem, otherwise go back to step 1;
5. Retest the modified system. If the problem is solved you're done, otherwise go back to step 1.

There are four basic factors that determine the success and the effort of the debugging process:

1. The amount of information the system provides about the error condition;
2. The tester's knowledge of the system under test;
3. System complexity;
4. Frequency of a problem's reoccurrence.

<u>The information about the error condition</u>

The amount of information available to the tester helps him define the subset of the system and inputs that are involved in the error, and as such allow him to define smaller, more managaeable sets to look for the mechanism of the problem at hand. However, collection of test information online requires additional system and design resources, and is often neglected in the system design, with an obvious adverse effect on the debugging process.

In the 370 Series of IBM mainframes almost one third of the hardware was dedicated to the "unit check" information collection process, which was clearly a factor in establishing this product as the most reliable computing platform. Today technology and fabrication processes are much better and more reliable than in 70s. However, modern systems are also more complex and integrated, while the amount of resources dedicated to testability is far lower, making the debugging process so much harder.

There are various methods that one can try to substitute for the lack of information, like incorporating higher level checks, differential testing - forcing the system to some specific states or slightly modifying the system and retrying the tests (if the problem disappears there is some additional information here), or comparison testing – compare

the system behavior with a second, nearly identical system under the same conditions and checking if both system behave the same way.
Higher level checks (various checksums, error detection and correction mechanisms and protocols) are systemic methods defined during the system definition process.
The other methods tried during the debugging process are often very time and effort consuming. *Ironically, the most time is taken by problems, which, because of lack of information, cannot be solved.*

The tester's knowledge of the system under test

Given incomplete information about the error environment forces the tester to guess the next step. In this situation, the tester's knowledge and experience come into play. The more knowledge the tester has about the system and the more experienced the tester is (i.e., had previous experience with the same sort of system behavior), the higher the probability that he/she will guess correctly. Once we are in the stochastic space of behavior, the debugging process becomes more an art than a science. The tester is guided now more by intuition than by anything else. And, interestingly, some testers will consistently be better than others.

Since no human mind is capable of comprehending a system with all of its details as a whole, it is mandatory to minimize the problem analysis to a part of the system and input conditions that are directly involved with the error detected. This can be done only by analysis of the data available and knowledge of the system operations. Once determined, this part can be tried, simulated or forced to various states (some systems give you such capability, system simulators always do). The simpler the definition of the subsystem is, the easier it is to find the source of the problem and provide an appropriate solution.

In the case that the source of an error cannot be determined, the only alternative is to try to imagine what the mechanism of the problem is, and modify the system to be immune from this error, if at all possible. Please note, that if an error disappears, as a result of any modification, if any, it doesn't mean it will not return, in one form or another. *Only thorough understanding of the problem can provide a proper solution.*

System complexity

This theme is mentioned in this article time and time again. Hence, the reduction of system complexity is of paramount importance. Ideally, the system is comprised of dedicated modules, each as simple as possible and dedicated to a particular function, which can be thoroughly tested, and with simple, very precisely defined interfaces between adjacent modules.

Universal, integrated systems, on the other hand, are very difficult to test. A variety of errors can lead to the same system malfunction, making error determination extremely vague. Generally, they also provide very limited testing information, often making error determination impossible. As a result, familiar system restarts are required, while users' expectations are that the problem will not repeat itself. Another alternative is to blindly replace part of the system, or even the system as a whole, hoping that the problem will disappear – a practice rarely admitted but often performed.

Frequency of a problem's occurrence

Each occurrence of an error can provide some additional information. Generally, persistent problems are the easiest to detect. If the error is transient, it is very difficult to catch; and oftentimes is never determined. Sometimes, the frequency of an error's occurrence can be artificially amplified by forcing the system to some repetitive conditions that are associated with the suspected error source. Such tests are often used and are useful for some classes of errors, but they cannot assure correct operation of the system.

Another technique for seldom occurring errors possible in a system is setting traps for a suspected error, i.e., forcing the system to perform in a different way once the system is affected by the error, and as such provide some extra information about the error. This technique requires a very thorough understanding of system operations.

Debugging tools

Various debugging tools are well described in their respective vendor literature and will not be addressed here. However, let me share a few comments.

First, the tool must be extremely reliable, since the tester depends upon it.
Second, they should be active online (i.e., while the tested system performs it's intended function in normal conditions), while minimally interfering with the system and its functionality. In offline testing, errors encountered online may not be visible. This goal is only achievable if the testing function is an integral part of the design, a rare case today.
Third, the device under test (DUT) should not be the only one error reporting mechanism. The reason is that if as a result of the error the DUT is disabled, there is no other way to collect the error data and define what the error is. In other words, it is advantageous to have an external error monitoring mechanism and resources. An additional advantage of such an external mechanism is its ability to offload the DUT from some online testing and data collection tasks. This advantage is most visible in the embedded applications where the DUT is fully loaded with its main application, and has no resources available for collecting testing information.

Debugging FPGAs with embedded processors

Debugging FPGAs with embedded processors poses all of the challenges noted above. These systems are complex, highly integrated and have a very limited visibility of internal operations. An additional challenge also arises from the heterogeneous hardware/software nature of the designs, each one using its own control and debugging environment. Often, in order to bring the design to some desired state, both programming and hardware changes are required simultaneously, while tools used by both environments are separate. There are attempts (like Xilinx's EDK) to put the tools under one common umbrella, however such solutions are seldom sufficient.

The approach that we are using in our FPGA development platform (see www.halstor.com) employs an external processor which is the operational base for the debugging tools. This processor is tightly coupled with the DUT; it can control its

operations and has access to common, memory-based debugging information. In this way, the embedded processor is largely free from the tools overhead and their special requirements (like a specific operating system), while the external processor can be adopted for any set of tools. There is one related and important point: Using somebody else's operating software or hardware cores can make the debugging process harder, due to problems related to interaction between various parts and sometimes bugs inside purchased components. This brings us to one more rule:
Thorough debugging requires complete control over all parts of the design, including purchased components.

The embedded processor communicates with the external processor using operations like xput(), xget() and xprint(). These operations can be easily inserted into the operating program, providing the external processor with information about the system operations. In addition, placing the program stack and data regions into the common memory provides the external processor with debugging information even when the embedded processor is hanging due to some severe error. No user interface is required to be implemented by the embedded processor, since this role has been shifted to the external processor. As a result the execution program was greatly simplified, making the testing easier.

Halstor tested its approach and platform in our SAS and Fibre Channel designs. The result was that we were capable of implementing these complex designs within record time using very limited resources, while most of the debugging was done online.

Final Thoughts
The debugging process is iterative in nature, with the number of iterations unknown from the start. Finding some errors can take days, weeks, even months. As a result, this is the most unpredictable and possibly most expensive part of the design process. Even worse, this is a never-ending process, since problems can be found once the product is manufactured in quantities and distributed to thousands of users. More than this, some users will invariably put the product to work under conditions not expected or not tested by the manufacturer.

The testing considerations should be part of system specifications. Often these considerations are partial at best, covering only a subset of possible error sources and operational only under specific conditions. Such approaches may be sufficient in some applications, but not enough in applications that rely on system reliability.

There is always a difficulty in striking a balance between the cost of the system and its testability. Too often the testability is underestimated and sacrificed for the sake of expediency. However, the more complex the systems are, the more important the testability is, and any compromise can backfire badly.

About the author:

Samek Mokryn is the founder and president of HalStor, Inc. (www.halstor.com), a design and consulting company specializing in storage and communication interfaces and architectures. He founded HalStor, Inc., after more than 30 years of experience in product development, design and project management.

In 1996, Samek founded C-Star Corp., a company that focused on the development of new data technologies. C-Star later became SANgate Systems - a storage appliance company - where he invented SANgate Systems' breakthrough technologies for managing stored data. SANgate Systems later became Sepaton, Inc., (http://www.sepaton.com) currently located in Marlborough, MA.

Samek previously held senior-level technical positions with EMC Corp., including leading the development team for ESCON connectivity within EMC's Symmetrix Information Storage Systems. He also influenced EMC's product and marketing policies. Before joining EMC, he served in various development roles related to I/O controllers and computer systems.

Samek holds an MSEE degree from Columbia University and earned a BSEE degree from Technion of Haifa, Israel.
He holds two patents in the area of a data replication.