

First Principles: Why Bother With This Methodology Stuff, Anyway?

by Joshua Rensch, Verification Lead, Lockheed Martin and Tom Fitzpatrick, Verification Methodologist, Mentor Graphics Corporation

Many of us are so used to the idea of “verification methodology,” including constrained random and functional coverage, that we sometimes lose sight of the fact that there is still a large section of the industry to whom these are new concepts. Every once in a while, it’s a good idea to go back to “first principles” and understand how we got where we are and why things like the OVM and UVM are so popular. Both authors have found ourselves in this situation of trying to explain these ideas to colleagues and we thought it might be helpful to document some of the discussions we’ve had. If you’re new to the idea of object-oriented testbenches in SystemVerilog and maybe are wondering what all the fuss about UVM at shows like DAC and DVCon is all about, or if you’re getting ready to take that plunge, we think these ideas might help you “begin with the end in mind.” If you’re an “expert” at this stuff, we hope that this dialog will help you take a step back and appreciate how far we’ve come as an industry and remember not to get too hung up on the whiz-bang features of a methodology but to keep in mind the ultimate goal, which is to make sure that our chips are going to work properly.

For discussion purposes, we will refer to “Design Verification” (DV) as the process by which an ASIC or FPGA is checked to make sure the design is accurate and correct. It involves several techniques that depend on the complexity of the design and its intended application. For example, if you were interested in verifying a graphics processor for video game systems its DV would be different than a design for a safety system of an aircraft. These techniques include but are not limited to functional verification, formal verification, formal equivalence and emulation. This paper will focus on functional verification but there is a lot of crossover of skills for each of these techniques.

WHAT IS FUNCTIONAL VERIFICATION?

Functional verification is testing a design in a virtual environment crafted by a DV engineer utilizing a verification methodology such as Universal Verification Methodology (UVM). This is done by driving the various inputs to a design through simulation and checking to make sure the outputs are correct. These inputs can be various standard

or proprietary buses and/or discrete signals to the design. The virtual verification environment is made up of various components, which will be defined later in the paper. [note: The DV engineer is trying to create a model of the “real world” in which the chip will operate.]

These designs can have 1000s of inputs, which would be impossible to test adequately without the use of various levels of abstraction. This allows a DV engineer to write tests at a high level of abstraction but still be able to drive the design. To give a real world example, say you need to move a house. At the top level of abstraction, you need to move the contents of the house; this is similar to moving a large piece of data from one memory to another. As you move down the levels of abstraction, you need to move a pantry; which for a design is like a transaction on the memory. To the final layer of abstraction, the jar of peanuts you need to move, which are the individual signals on whichever memory bus you are using. All you really would like to worry about is that the house was moved. But the reality is that part of move is the jar of peanuts.

HOW DO WE TEST EVERYTHING?

These designs can be massive, encompassing many different functions and paths. There is no practical way to test every permutation of all states of a large design. That is technically true, but we can test relevancy and sufficiently representative large parts of it. A DV engineer uses a technique called Constrained Random Verification (CRV) which stresses the design in the most comprehensive and efficient manner. This consists of constrained random stimulus, self-checking testbenches, and functional coverage.

Constrained random stimulus ensures that the stimulus is meaningful. The constraints are the key, otherwise we just have random stimulus that may or may not be representative of real-world stimulus. The design can be stressed more quickly and the randomness can create corner cases that a human verifier might miss. Since it is random, the test can be run multiple times and create different stimulus each time. This is controlled by a seed fed into the tool which, along with the constraints, provides

a way to produce near infinite sets of valid stimulus. Say you want to check and make sure that every part of the house's roof is waterproof. You could methodically use an eyedropper on every shingle, which would take a prohibitively long time. Wouldn't it be better to use a sprinkler on it and randomly hit most of the locations and then use the eyedropper on parts that were missed?

Self-checking is exactly what it sounds like; it automatically determines if the design performed correctly. In a traditional testbench methodology, the outputs are typically manually checked after the inputs are created. With random stimulus, that method can be tedious and prone to errors, self-checking at the proper abstraction level to the rescue. Utilizing a higher level of abstraction to check the correctness of the design allows for quicker environment creation and there are additional checkers for the lower levels of abstraction. Using the house analogy, check that each room of the house was properly moved and have lower level checkers verify that each room was moved properly.

HOW DO WE KNOW WHEN WE ARE DONE?

This is the typical question from various managers in the field. Luckily there are three metrics that help determine when the verification is completed. They are functional coverage, code coverage, and bug curves.

Functional coverage is defined by a concept of assertions and cover statements. An assertion is defining the proper behavior for something. An example assertion would be that a house should keep a person dry. Therefore, if a person inside a house ever gets wet when it rains, the assertion fails. The problem is that if it never rains, that assertion will always be true, but is never tested (we call that "vacuously true"). That is where a cover comes into play. A cover states, a person should be inside the house when it is raining outside and testing isn't complete unless that happens. One of the main responsibilities of a DV engineer is to make sure that all the coverage points are exercised.

Code coverage is built into most simulators. It gives the percentage of the code that has been exercised. This is

a raw metric, which by itself has limited value. But when tied with the other metrics, it gives a good measure of confidence. The reason code coverage is not the only metric that should be used is that without an understanding of previous executed code nor the interrelationship with other statements of code and the results are not verifying the manner in which the code was executed. This is a problem because a number of latent defects are found when code interacts with other code. To take it back to the house, code coverage would give us an understanding that we have a house, that we were inside it and that it rained; it would not give us the idea that we were inside the house during the rain. It is possible to have 100% code coverage on a buggy design.

Bug metrics are the last part of this tripod. Tracking bugs found is important because the verification job isn't complete if the bug curve doesn't flatten out, meaning that if the DV engineer is pulling out four to five bugs a week, the design is not completed. But just because the bug curve is flat doesn't mean that testing is completed, the functional and code coverage need to be checked simply because the DV engineer might be testing the same small piece of logic and not getting to other logic that needs to be tested.

THIS SOUNDS COMPLICATED, IS THERE ANYTHING THAT CAN SPEED DEVELOPMENT UP?

Glad you asked, now for a brief history lesson. For a long time, the industry was fragmented, forcing vendors to have their own tools and techniques to solve this problem. That was until SystemVerilog became an IEEE standard in 2004. Vendors started to support the verification tools within System Verilog and developing their own methodology using System Verilog. It wasn't until this year that they standardized on a methodology UVM. This had all the tools to create an environment. To use the house analogy, System Verilog provided the raw materials to build a house, wood, stone, metal and glass. UVM provided frames, doors, windows and sinks to build your environment from. No longer did you need to develop your own way of doing things.

A UVM environment is built using various objects; the basic building blocks of which are called UVM sequences and UVM components.

UVM creates stimulus by using sequences and sequence items. These define the test that will create the operational parameters for the test. Think of these as a list of instructions on how to build the house. There can be several sequences, which don't have to have any knowledge of each other. In the house analogy, they describe how to build the various rooms that will make up the house. The order in which these rooms are built does not typically matter.

One pre-defined component is called an agent. These are the objects that control the various buses into the DUT. It contains both a driver that presents transactions to the bus and a monitor, which captures transactions on the bus and reports them back to the environment. It arbitrates for usage of the resource it is connected to. Arbitration is essential to make sure that the environment works in an organized manner since many sequences could be trying to access the bus at the same time.

Another pre-defined component is a scoreboard used to verify that transactions coming out of the DUT are correct. This is done by using an agent to capture the transaction and checking against a transaction that was created by some object in the environment that is predicting what the DUT will do. These predicted transactions could be created by some component in the environment getting the same as the DUT and determining what the DUT should create.

ARE THERE ADDITIONAL BENEFITS TO USING THE VERIFICATION ENVIRONMENT?

There are a couple of advantages to using a sophisticated verification environment. One is that once you have it built, adding another path or feature isn't typically difficult. If you are using a traditional testbench methodology, you may have a long task to modify the environment to test it. With this approach it could be as simple as changing the path the data takes. Also, if a bug is found later on in the development cycle, it can be replicated and triaged in the DV environment. This will lead to faster turnaround times since with simulation there is better visibility into the design.

WHAT KIND OF PERSON DO WE NEED TO DO THIS?

The typical DV engineer has to be a hybrid of a hardware and software engineer. They need to be able to understand the hardware world, comprehend the specification of a hardware product, and translate that into the environment that needs to be created. They need the skills and understanding of the world of software but with the grasp of design to create more complete and correct testing. Good DV engineers have an understanding of concepts like Object Oriented Programming and Transaction-Level Modeling (TLM) to better utilize industry standard verification techniques.

The DV engineer or team needs to be independent of the designer or design team. This provides for two sets of eyes on a design, as each will interpret a specification or requirements with their own personal bias. This independent checking will lead to a better design and a more complete set of documentation, for if the engineers don't agree; the customer will likely misinterpret it.

CONCLUSION

How do you calculate the cost of missing a bug? The typical profile is that there is some initial investment in putting the infrastructure together followed by a substantial gain as results start to come in. As a rough example, suppose a design needs to be verified and it is determined that it would take 5 days for a DV engineer to create an environment. In the same span, a non-DV engineer creates, debugs and executes a test a day so after a week, you have five working tests. After four days, the directed approach has 4 tests and the DV approach has none. On the 5th day, the directed approach has 5 tests, but the DV approach has literally 1000's of tests. Why?

A properly architected UVM environment allows you to create many variations on the theme automatically. Remember, that in addition to randomizing stimulus, you're also randomizing the structure of your testbench.

DV is standard for all ASIC companies around the world and until recently its use on FPGAs has been limited. That was until the size and complexity of FPGAs made it no longer a trivial task to debug it in the lab. The earlier in the development cycle the bug is found, the quicker and cleaner it can be fixed. By doing DV in parallel with the design, it reduces time in the lab and slips in program schedule.