Verification of Application-Specific Integrated Circuits at the Block Level in Solid-State Drive Controllers

Non-Volatile Memory Solutions Group
Intel of Canada, Ltd.
Vancouver, British Columbia

James Mackenzie
V00737373
Work Term 3
Computer Engineering
jimack@uvic.ca

September 4th, 2013

In partial fulfillment of the requirements of the B.Eng. Degree

Supervisor's Approval: To be completed by Co-op Employer

I approve the release of this report to the University of Victoria for evaluation purposes only.
The report is to be considered (select one): □ NOT CONFIDENTIAL □ CONFIDENTIAL

Signature: __________________________  Position: ____________  Date: ________________

Name (print): ______________________  E-Mail: ______________________  Fax #: ____________

If a report is deemed CONFIDENTIAL, a non-disclosure form signed by an evaluator will be faxed to the employer. The report will be destroyed following evaluation. If the report is NOT CONFIDENTIAL, it will be returned to the student following evaluation.
# Table of Contents

List of Figures................................................................................................................i

Summary .........................................................................................................................ii

Glossary of Terms and Symbols ....................................................................................iii

1.0 - Introduction.............................................................................................................1
  1.1 - Intel and Non-Volatile Memory Solutions Group .............................................2
  1.2 - Intel Solid-State Drives ......................................................................................2
  1.3 - Constrained Random Verification and the Universal Verification Methodology............................................................................................................2

2.0 - Constrained Random Verification and Coverage-Driven Verification ..........4
  2.1 - The Framework of Coverage-Driven Verification..........................................4
    2.1.1 - Driver ........................................................................................................5
    2.1.2 - Monitor .....................................................................................................5
    2.1.3 - Model ........................................................................................................6
    2.1.4 - Scoreboard ...............................................................................................6
    2.1.5 - Design Under Test ...................................................................................6
  2.2 - The Process of Coverage-Driven Verification................................................6

3.0 - The Universal Verification Methodology.........................................................10
  3.1 - Why UVM? ......................................................................................................10
  3.2 - Using UVM to Verify a Design ....................................................................11
    3.2.1 - Sequences ..............................................................................................11
    3.2.2 - Phases ....................................................................................................11
    3.2.3 - Factory ..................................................................................................12
    3.2.4 - Register Modelling .................................................................................12
List of Figures

2.1a The Basic Coverage-Driven Verification Framework

2.2a The Coverage-Driven Verification Flow Chart

2.2b An Example Coverage Point in SystemVerilog

3.2.2a UVM Phases
Summary

Completely verifying the operation of an integrated circuit is inherently difficult. In modern integrated circuit designs, there can be millions upon millions of gates and billions of transistors. Verifying such a design by creating specialized directed tests is one method of testing, but then one is presented with the inevitable question: How many tests would adequately verify the design? Moreover, what defines “adequate” verification? In the place of directed testing, functional verification along with coverage-driven verification are used. Coverage-driven verification aims to verify a device by stimulating said device with the broadest amount of inputs as possible. This stimulation is intended to coerce the device under test into as many different states as possible. These different states should ideally activate the device’s full functionality, which leads to the functional verification of the device. Coverage-driven verification is achieved through a software test bench, which is created around the HDL used to design the DUT. As with a hardware test bench, the DUT is surrounded by devices used to provoke, measure, and model the operation of the device. Test benches that make use of coverage-driven verification are often made up of four types of components as well as the DUT: drivers, monitors, models, and scoreboards. Drivers create and send stimulus to the DUT. This stimulus should be as unconstrained as possible, allowing for as many possible states and outputs from the DUT. Monitors are used to retrieve values both input and output from the DUT, so as to allow later use for verification of the information output from the DUT. These values are stored by the scoreboard for later use. Monitors often only ensure the data was produced by the DUT in the order expected, whereas verification of the values themselves is done by the model. As the name suggests, the model models the operation of the DUT, and when differences arise between the output data found in the scoreboard and modelled data, an exception is thrown and the test is stopped. Any functional coverage data for test runs that end because of errors in the test bench is discarded, as such data does not aid in confirming the validity of the design. The final component of the test bench is the scoreboard: the scoreboard’s role is to examine the DUT’s inputs for transactions to occur, capture pertinent information, and store the information for later use, either in functional coverage or as a data source for the model. The Universal Verification Methodology enhances the coverage-driven verification methodology using a number of software constructs: sequences, phases, register modelling, and the factory. Sequences allow logical and complete sequences of data to be created, randomized, and dispatched to the driver. Phases allow the verification engineer to organize the building, running, and cleanup of test cases and test benches in distinct phases, allowing fine organization and control over what is executing in a test at any given time. Register modelling allows the integration and modelling of all the registers and memories in a DUT into the test bench. The factory is a general-purpose interface used to provide information about objects on their instantiation, and allows verification engineers to easily override objects created in the test bench. This report recommends two specific areas of the UVM verification environment be improved: first, the SystemVerilog language is recommended to be improved so as to allow overriding and extension of coverage groups; second, it is recommended that EDA vendors investigate methods that could do away with complex algorithmic solving and find effective methods to more easily generate statistically-agreeable data, perhaps by allowing open-source solvers or saving pertinent data.
Glossary of Terms and Symbols

**Accellera Systems Initiative (Accellera)**
A standards organization focused mainly on developing electronic design automation and integrated circuit design standards.

**Application Specific Integrated Circuit (ASIC)**
An integrated circuit designed specifically to perform a certain task or set of tasks.

**binary**
The base-2 counting system, consisting of combinations of 1s and 0s.

**bus**
A communications system that transfers data between computer components.

**coverage**
A measure used to describe the degree of which a particular metric has been tested.

**Device Under Test (DUT)**
A device being tested.

**First In First Out (FIFO)**
A software data structure concept. The first item to be “pushed” onto the queue will be the first item “pulled” off the top when accessed.

**functional verification**
The task of ensuring a logic design used for an integrated circuit performs as expected.

**Hardware Design Language (HDL)**
A specialized programming language used to design integrated and electronic circuits. Common HDLs are Verilog and VHDL. See “Verilog”.

**Hardware Verification Language (HVL)**
A programming language used to verify circuits designed in hardware design languages (HDLs). Common HVLS are SystemVerilog. See “SystemVerilog”.

**hexadecimal**
Also known as “hex”, this is a base-16 numbering system.

**Mentor Graphics**
An electronic design automation company, producing among other tools such as Questa Advanced Simulator.
Negated AND or NOT AND (NAND)
A logic gate used as the basic building block of digital systems due to its ability to implement any boolean function using a combination of them. Often arranged in series and used to store data (NAND flash memory).

Open Verification Methodology (OVM)
The verification methodology succeeded by UVM.

Printed Circuit Board (PCB)
A non-conductive sheet (usually made of plastic) that physically holds electronic components in place. These components are linked together using conductive pathways printed directly onto the board.

Questa Advanced Simulator
A HDL simulator. Also a competitor to Synopsys’ Verilog Compiler Simulator (VCS).

register
In computing, a register stores bits of information. These bits can usually be both written and read out of the register, but this is register-dependant (i.e., some registers may only allow external devices read access to certain register, only allowing those registers to be modified by internal influence).

regression testing
A method of software testing that aims to discover new bugs after making any change to the environment or the software itself.

stepping
A term primarily used by Intel as a sort of revision number for integrated circuits. For example, a new design would be designated A0, with minor and major revisions of the circuit increasing either the number or the letter, respectively.

Synopsys
One of the largest companies in the electronic design automation (EDA) industry, Synopsys designs tools for engineers to design and verify electronic designs. See “VCS”.

SystemVerilog
An object-oriented programming language, combining both a hardware design language (HDL) and a hardware verification language (HVL). Based on the Verilog HDL.

test bench
A virtual software environment to verify a certain product. In the context of this report, the test bench is the verification platform that contains the DUT along with the framework of tests and related software used to verify the design of the DUT.
Universal Verification Methodology (UVM)
A standardized methodology from Accellera for verifying integrated circuit designs. Also includes a base class library that allows for the straightforward adoption of its methodologies.

Verilog Compiler Simulator (VCS)
A HDL design simulator and functional verification tool designed by Synopsys.

word
In the context of this report, a word is a unit comprised of 8 bits.
1.0 - Introduction

Predicting every single use case of any device is an almost impossible task. What is often even more challenging is predicting what would happen to a device if it was used in an unexpected way. For example, it is certainly very easy to think of a number of intended uses for and actions taken by a travel website. Perhaps the site would be designed so as to not recommend a hotel in Los Angeles to someone flying from Vancouver to Toronto. Once you had selected and confirmed your flight to Toronto, the website would then automatically contact the airline, process your reservation, and charge your credit card. But what would happen if you tried to book a flight from London to Rome while also booking a hotel room in Iceland and a day pass at Disney World in Florida? Would the site allow such a combination to happen, even though there are very few scenarios which might allow this scenario to work? If it did, what actions would the website take, and how could they be verified to be correct?

The goal with constrained random verification is to ensure randomly-generated stimulation of a device’s inputs falls within certain ranges, where these ranges are often, but not always, established to ensure the device stays within well-defined behaviour. By stimulating the device with a large amount of random constrained stimulus, a verification engineer can eventually reach the vast majority of values a device could possibly see in its operational lifetime. Such randomness in inputs can often cause bugs to be revealed that were not anticipated by the verification engineer. If the generation of random input has approached approximately all the possible inputs and outputs of a device with the device functioning as expected throughout, one can say with a high degree of certainty that the device is functionally verified. If there are still areas of the design that have not been tested after a large amount of constrained random verification, writing test cases that focus directly on that area of the design (“directed” test cases) would allow a verification engineer to ensure the whole design worked as expected. In the case of the travel website, one could constrain the site’s inputs early in testing to only allow hotels within a certain radius of the departure or arrival airports to be chosen, then expand those radii as bugs are discovered and fixed. Eventually, one could write a directed test case for the scenario posited in the first paragraph to ensure the site handled the scenario as expected.

This pseudo-random stimulation can be combined with device verification methodologies to automate the procedure of generating constrained stimuli, writing and reading to and from registers and buses, and modelling the device under test (DUT). This report focuses on the constrained random verification methodology used at Intel’s Vancouver office to verify solid-state drive controller application-specific integrated circuits (ASICs) used in Intel solid-state drives. Specifically, this report will cover the Universal Verification Methodology and the use of SystemVerilog to verify block-level ASIC designs in the pre-silicon (i.e., HDL) phase of design.
1.1 - Intel and Non-Volatile Memory Solutions Group

Intel is known worldwide as a leader in excellence and innovation in the technology and semiconductor sectors. With divisions that focus on computer processors, non-volatile memory, supercomputers, and client software, Intel’s worldwide workforce of over 82,000 employees foster an unique atmosphere of innovation and collaboration [1]. This dedication to innovation is easily verified by the more than 1,600 US patents issued to Intel in 2010, and has seen the US Patent Office issue more than 1,000 patents to Intel a year for the last 20 years [2]. Its Canadian division, Intel of Canada, Ltd. (hereafter “Intel Canada”), maintains a strong presence in Canada with its Vancouver location serving as an important office for ASIC, firmware, and hardware engineering teams in the Non-Volatile Memory Solutions Group (hereafter “NVM Solutions Group”).

The NVM Solutions Group designs and manufactures solid-state drives for desktops, laptops, supercomputers, and data centres. The NVM Solutions Group often designs every major part of any given Intel solid-state drive, from the controller ASIC through the firmware architecture and the PCB layout to the final verification of the drives themselves.

1.2 - Intel Solid-State Drives

Intel’s solid-state drives (hereafter “SSDs”) are renowned in the semiconductor industry for their high reliability and extremely fast and consistent performance when compared to other vendors [3]. One 2010 report found that the mean failure rate for Intel SSDs was “more than three times [lower] than [other SSD-producing companies]” [4]. The root of this reliability can be theorized to arise from a combination of different factors: NAND sourcing, PCB design and testing, ASIC design and testing, and overall quality control.

1.3 - Constrained Random Verification and the Universal Verification Methodology

In any given ASIC design, it is extremely common for there to be tens of input and output registers, hundreds of configuration registers, and millions of gates. In order to adequately verify the correct operation of a design, these inputs and configuration registers must be stimulated with as many different combinations of values as possible. Writing tests that are directed to address certain features (“directed” tests) may allow a general verification of a HDL design, but this inevitably begs the question: how many hundreds or thousands of directed tests would allow an adequate verification of a design? More importantly, what defines “adequate”?

One answer to these questions is random verification. Random verification allows a verification engineer to stimulate a design with random values that ideally allow each register to incur every possible value. The most apparent problem with this approach is that such testing could allow a forbidden value to be input into the ASIC, leading to an error condition or other unwanted state.
Constraining these random values to those that only lead to valid configurations would solve this issue. This constrained-random verification is a common solution to block-level ASIC testing, as testing methodologies using this notion can often produce the highest functional verification of all possible values in and out of the ASIC.

Using a testing methodology to design an overarching framework to test a device is considered essential in constrained-random verification. These methodologies are commonly developed by either vendors or standards bodies and allow verification engineers to create “test benches” - virtual frameworks of code that stimulate, read, and verify a design. The Universal Verification Methodology (UVM) - developed by electronics design standards organization Accellera as a solution to a number of competing testing methodologies - is a direct descendent of the Open Verification Methodology (OVM), itself a descendant of the Advanced Verification Methodology (AVM) and the Verification Methodology Manual for SystemVerilog (VMM).

These other methodologies are well-known and popular within the electronic design industry, but not largely supported by simulators anymore like Synopsys VCS and Questa Advanced Simulator - this is especially true in the case of AVM and VMM, the specifications and methodologies of which are years old at the time of this report and have been mostly replaced by OVM at this point. OVM is still very widely used in verification, but a number of verification vendors and training providers now recommend the use of UVM as all the benefits of OVM, AVM, and VMM have been combined into it.
2.0 - Constrained Random Verification and Coverage-Driven Verification

When it comes time to verify a block-level ASIC design, there is often little discourse on whether or not to primarily pursue constrained random verification as opposed to directed verification. While writing directed test cases to cover certain important features of your design can still lead to bugs being found, the edge cases of the DUT can be difficult or impossible to discover and verify. Using constrained random values and coverage-driven verification, a verification engineer can build a test bench that ensures the stimulation and configuration of the device is as random as possible while ensuring the stimulation stays within such values that keep the DUT in a predictable state.

In coverage-driven verification, there are two types of coverage: code coverage and functional coverage. Code coverage measures whether or not the code in the design of the DUT has all been run: if it has not, then the code that did not run - the “hole” in the coverage - must be examined. There may be a bug in the DUT, or the constrained random verification has not so far been able to coax that section of the design into operation. Code coverage is very useful in the sense that if a section of code has not been run in a test, it has clearly not been verified. Further to that, in many simulators, code coverage can be enabled by simply enabling a code coverage option before a given test is run, making it very simple to collect. However, high code coverage does not necessarily mean the design of the DUT is verified, as high code coverage does not imply the code in the DUT is performing its task correctly.

In order to ensure the DUT is producing the correct values, there is functional coverage. Functional coverage is used to record the values and states of the DUT for later analysis by a verification engineer or commercial analysis program. The process of defining the functional coverage of a device is not an automatic one unlike code coverage. Defining functional coverage requires an engineer to manually define what values must be input and output of the DUT. For example, functional coverage could measure whether or not every type of instruction was created and written to an external interface by the DUT, or whether every possible value of a pointer was seen when reading from a FIFO stack.

As more and more constrained test runs of constrained random verification take place, both functional coverage and code coverage of the DUT will increase. Bugs are found and fixed as edge cases occur, and the test bench is revised through the removal, addition, and adjustment of constraints and the addition of both constrained random tests and directed tests so as to improve coverage.

2.1 - The Framework of Coverage-Driven Verification

In order to facilitate the verification of a design using coverage-driven verification, a test bench must be developed to stimulate, monitor, verify, and store the results of the DUT. Much like a
physical test bench where an engineer would test an electronic device with power sources, wave generators, and oscilloscopes, the software test bench seen in Figure 2.1a provides a full environment with which to carry out constrained-random verification. A test bench for coverage-driven verification must contain the following software components: a driver, monitor, model, scoreboard, and the DUT.

2.1.1 - Driver

The stimulus (or “driver”) component is designed to model the other design block or blocks surrounding the DUT and provide the stimulus that would normally be provided to the DUT. The entire behaviour of the neighbouring design component should not be modelled; instead, the stimulus component should only mimic the interface inputs to the DUT. This not only makes the test bench code easier to maintain, but also allows the stimulus engine to drive the interfaces free of the burden of the realities of the neighbouring design component it is mimicking.

A key concept in coverage-driven verification is that the stimulus component must drive what the DUT is capable of accepting and not restrict itself to what the real neighbouring design component or components might send. This allows a verification engineer to exert the maximum amount of stress possible on the DUT. If at all possible, this stress level should exceed that which will ever occur when the ASIC block is placed in the overall ASIC design. By exceeding the limits of the design, it is more likely the DUT will encounter edge cases. These corner cases otherwise might never be seen until production or release, and could if not fixed in the verification phase cause a costly and time-consuming new stepping to be created. This concept of over-stressing the DUT allows a software test bench with a finite amount of cycles run to compete with the relatively infinite number of cycles run in fabricated hardware, and thus allows for a much more complete verification.

2.1.2 - Monitor

A monitor observes the DUT’s input and output values. A monitor can also observe other values in the test bench that may be values of interest. At a minimum, the monitor must observe the outputs of the DUT. If the DUT does not adhere to the protocol in use in the interface, the monitor has the capability to cause an error to be raised in the test bench. Some verification of the received data is done at this stage, but it is only often parity checks and other similar data
integrity verification. While the monitor does not drive any signals and simply observes data in the test bench, this is so the monitor can be reused in other levels of the design. It is in the monitor that functional coverage data is often collected and analyzed by the use of cover points and bins (see section 2.2).

### 2.1.3 - Model

While at first glance similar to a monitor, the model of a design is far more complex. Also called a “checker”, the monitor validates that the design is working as intended from a functional standpoint as well as a protocol standpoint. Often the most difficult component of the test bench to get correct, the model must simulate the design of the DUT in the sense that given the same input as the DUT, the model must return the same value. The model must retrieve the DUT’s input values and then produce a response just as the DUT would. Once the DUT and the model have both computed output values for a given set of input data, the two outputs are compared. If they are the same, the test is allowed to continue. If not, the test case is considered invalid and all coverage (both functional coverage and code coverage) is useless, as this coverage did not result in the verification of the correct operation of the DUT.

### 2.1.4 - Scoreboard

The scoreboard’s role is to examine the DUT’s inputs for transactions to occur, capture pertinent information, and store the information for later use. When the model observes data on the output interfaces of the DUT, the model queries the scoreboard to get said data so it can be compared with the reference model and verified.

### 2.1.5 - Design Under Test

The design under test (DUT) is the device to be verified using the test bench. This component of the scoreboard is often the Verilog source files or other HDL source files used to design the DUT. The DUT is connected to the test bench using the monitor and the driver to read and write to the DUT. The level at which the DUT is tested (block, chip, etc.) is up to the verification engineer, but it is straightforward to use the constructs outlined above to do so as they do not depend on a design being a block level one or any other.

### 2.2 - The Process of Coverage-Driven Verification

Coverage-driven verification is an intensely iterative process. As seen in Figure 2.2a, the iterative process involves three steps: running regression tests, reviewing the results, and refocusing efforts to patch holes in coverage. Regression tests aim to uncover any new bugs in the DUT, and are run when changes have been made to the source code of the DUT or the testing environment. This is to ensure no new bugs have emerged or old ones have resurfaced. However, before these iterations of testing, reviewing, and refocusing can begin, one must first perform two steps: define the functional coverage points, and then write said coverage points inside the test bench.
A coverage point is a language construct that monitors a specific variable in a test bench. These variables are often output values, but can also be input values or other values inside the test bench. These coverage points contain objects called “bins”; these bins are used to define ranges for the values of the variable to be collected. It is good practice to define these coverage bins so that all possible values that could be seen in that particular variable are recorded, including values that may indicate illegal operation.

For example, a verification engineer might want to collect the value of a variable that contains an address on an external interface of the DUT spanning from 0 to 0xffff on an external interface of the DUT. One possible way to define this coverage is to use four arbitrary bins named “low”, “medium”, and “high”, as shown in Figure 2.2b. One can see there are two bins declared for “medium” values by the use of the index next to the bin name. To ensure complete coverage of these values, the engineer should write this coverage point in a part of the test bench that can see every value the address variable has taken. As mentioned in section 2.1.2, the monitor of a test bench is the most common place in a test bench to store functional coverage points and bins.

```systemverilog
coverpoint address {
    bins low = {{16'h0:16'h3fff}};
    bins med[2] = {{16'h4000:16'hbfff}};
    bins high = {{16'hc000:16'hffff}};
}
```

**Figure 2.2b: An Example Coverage Point in SystemVerilog**
Once appropriate coverage points have been defined and written into the test bench, regression testing can begin. These regression tests make use of constrained random values for the vast majority of the values generated by the test bench. While care must be taken to correctly constrain the values within ranges that do not create unwanted conditions inside the DUT, the end goal of constrained random verification is to produce values that are as unconstrained as possible. This is because as the random variables become more constrained, the chance of accidentally hiding a bug becomes larger as fewer distinct values will be stimulating the DUT. However, when first beginning regression testing, these constraints may have to be extremely restrictive, perhaps only allowing one or two distinct values per test run. This is done to ensure basic DUT functionality and to aid debugging.

As verification continues and more results are generated, the constraints on the values driven to the DUT can be gradually loosened. By monitoring the results of the regression testing carefully, holes in the coverage - cases where outputs are not being properly stimulated and therefore not generating values to be recorded in coverage bins, or simply where coverage points were perhaps too liberally defined - are discovered and fixed. These coverage holes are often so-called edge cases: for example, if an output of the DUT is 16 bits (0xffff in hexadecimal) wide and the defined coverage bins indicate that every bit must be stimulated, some edge cases could be at the values 0x0 and 0xffff - the smallest and largest value the output could take. These holes at edge cases are often due to bugs in either the test bench or the DUT, and will not be solved by running more regression tests and generating more random data in the hopes of eventually stimulating them. In practice, it is often best to have coverage bins defined specifically for edge cases, with much larger bins storing values across the rest of the values that must be covered.

After running more regression tests, one method to solve coverage holes that are not edge cases - the cases where entire registers or interfaces may be largely unstimulated - is to write directed test cases. Directed test cases are used in conjunction with constrained random verification to ensure the DUT can be completely tested. Even though constrained random verification can and does test large parts of an ASIC design, there are often cases where specific features must be tested independently to the main regression test. These tests can easily make use of the test bench that was created for constrained random verification: in fact, directed tests are often simply highly constrained random tests that make use of specific fields, registers, or interfaces that the main regression test does not stimulate adequately or at all. Directed test cases often stimulate the DUT to produce error cases, and possess special routines that handle exceptions specific to the directed test case. It is also not uncommon to place coverage for such errors inside a directed test case, as the existing implemented coverage may not cover the error or feature that must be tested at all.

Both directed tests and loosening constraints are good ways to achieve functional coverage of an ASIC. Another method is enhancing the test bench itself. As verification of a DUT begins, a verification engineer may create a test bench only capable of verifying the major interfaces of the DUT. As verification continues the test bench may be enhanced to cover other smaller interfaces, or cover the existing interfaces of the DUT more completely or efficiently. For designs of any
size, enhancing the test bench as verification progresses is a far better option than waiting until a complete test bench is completed before beginning regressions. This is because issues in the DUT design can be solved earlier in the verification process and perhaps bring even more issues in the design to light. This also aids the overall project in moving to completion faster.
3.0 - The Universal Verification Methodology

Verification engineers look to practice efficiency and code reuse at every possible juncture. Using a testing methodology that is designed specifically for as much reuse of code as possible while still ensuring complete functional verification of a design can save verification engineers countless hours or months of work in creating a testing environment. The Universal Verification Methodology enables a verification engineer to do exactly that.

3.1 - Why UVM?

UVM is the successor to many different competing verification methodologies, integrating a number of features its numerous predecessors overlooked. UVM is directly descended from OVM, the Open Verification Methodology, which is in turn descended from URM, the Universal Reuse Methodology. All of these previous methodologies have aimed to enhance the ease of verification through various built-in types, coding patterns, and data automation features, such as automated object comparison, which is of particular use in the model to verify the integrity of data (see section 2.1.3). As with OVM, UVM is open-source, which enables the verification engineer to modify and enhance code in the UVM base libraries to their particular needs. However, because of the fact that UVM was designed to be used across many different variations of test benches and different verification tools, this is rarely necessary.

A new feature to UVM is that it provides register layer classes to create a high-level, object-oriented model for memory-mapped registers and memories. Previously, verification of registers and memories had to be handled by specialized libraries separate from the testing methodology. UVM also adds a command line processor to enable tests to view from within the test the command line options that were used to start the simulation. This processor also provides the capability to obtain the simulator’s name and version from inside the test, which can aid in compatibility and reuse of a test bench.

UVM facilitates the construction of test benches and tests, both by providing reusable functions and macros in the form of a library of SystemVerilog classes, and also by providing a set of guidelines for best practice when using SystemVerilog for verification. The architecture of UVM is designed to encourage modular and layered verification environments, where verification components at all layers can be reused in different environments. Low-level driver and monitor

Figure 3.2.2a: UVM Phases [6]
components can be reused across multiple DUTs, and the whole verification environment can be reused by multiple tests and configured top-down by those tests. This degree of reuse is enabled by having UVM verification components able to be configured in a very flexible way without modification of their source code.

The degree of reuse that UVM enables, coupled with the steady, incremental upgrades to its built-in library of SystemVerilog classes and additions such as register modelling, show that UVM is a significant upgrade over OVM and its predecessors. Those benefits have seen UVM be recently adopted by companies such as Intel, IBM, Texas Instruments, and ARM [7].

3.2 - Using UVM to Verify a Design

This section will provide a brief overview of UVM-specific features to functionally verify a design. As the overall framework of a UVM-based test bench is very similar to the coverage-driven verification framework defined in section 2.1, the structure of such a framework will not be reiterated. Rather, some of the differences between the coverage-driven verification framework and UVM will be elaborated upon, as well as the features UVM adds to aid verification engineers in implementing a full test bench for coverage-driven verification.

3.2.1 - Sequences

Complete stimulus generation in UVM consists of the following: a sequence item, a sequence, a sequencer, and a driver. In UVM, sequences of transaction items are used to create stimulus. This stimulus is then sent to the appropriate driver. While sequence items, sequences, and sequencers are new concepts, the driver mentioned here is equivalent to the driver presented in section 2.1.1.

A sequence item is an object that models data that the DUT would expect to receive. A sequencer would create the sequence item and send it to a sequencer. A sequencer is responsible for coordinating the driving of items between the sequence and the driver. When the driver receives an item from the sequencer, it is then driven to the appropriate interface of the DUT. Sequencers can also receive information from drivers and send it back to its parent sequence if necessary.

For example, in the case of verifying an ethernet switch one type of sequence item sent to stimulate the DUT would be an object that simulates an ethernet packet. A sequence would create the packet object, randomize its fields, and send the packet to a sequencer dedicated to the specific type of ethernet packet generated. The sequencer would then send the packet to the driver, where it would be driven into the DUT.

3.2.2 - Phases

Phasing allows tests written using the UVM base class library to build, run, and cleanup themselves in distinct phases, allowing fine organization and control over what is executing in a
test at any given time. As seen in Figure 3.2.2a, there are 20 different phases in which to perform different actions. The main phase, for example, is used to generate and drive stimulus to the DUT. Commonly, all 20 of these phases are not used in every test. Activities that could be performed in the pre_shutdown or post_shutdown phases of a test could often just as easily be performed in the shutdown phase, as there is often no need for such granularity for common shutdown tasks (such as reading registers, for example). Nevertheless, the granularity is there if a verification engineer can use it.

As the UVM standard advances and matures, more features are added to the phasing system. The ability to jump backwards and forwards between phases, for example, is a recent addition. However, more advanced features such as making the test bench aware of which phase the test is in and managing sequences in the context of phasing are still in the beta phase.

3.2.3 - Factory

In software development, factories are a common design pattern. These factories allow object-oriented languages such as the SystemVerilog language used with UVM to “define an interface for creating an object, but let the classes that implement the interface decide which class to instantiate” [8]. The UVM factory implementation allows exactly that: all objects to be created are registered with the factory, and as new objects are created the factory can decide exactly what derivative of the object to create.

For example, in the case of verifying an ethernet switch, a new test could be required that drives only ethernet packets with errors. Instead of creating new sequences, sequencers, and drivers or modifying existing ones, a new sequence item derived from the base ethernet packet class. This new derived packet class would have its data randomized in a fashion to only create errors in the DUT. The factory could then be used to override the base packet class in a new test with a single call to the factory, removing the need to modify any sequence, sequencer, or driver.

3.2.4 - Register Modelling

Similar to the model presented in section 2.1.3, in UVM the verification engineer can also create an abstract model of the registers and memories in the DUT. Since the registers and memories have straightforward API access for read and write operations through the UVM class library, they can be completely integrated into the functional verification coverage results. One must first create an abstract model of all the registers and memories in the DUT. Once this model has been created, a UVM-based hierarchy of the DUT’s register hierarchy can be generated. There are a number of vendors that provide automated tools to do this, such as ralgen [9]. Once this step is complete, register access can be enabled by the use of the UVM class library, which then easily facilitates the creation of a model of an address map. These registers can then be accessed from inside the test bench, allowing verification engineers to, for example, read error registers directly or toggle individual bits to enable the start the processing of commands by a DUT.
Conclusions

Because of their size and scale, modern ASIC designs do not yield themselves to trivial verification. To ease verification, functional verification along with coverage-driven verification are used. Coverage-driven verification aims to verify an ASIC by stimulating said device with the broadest amount of inputs as possible. This stimulation is intended to coerce the device under test into as many different states as possible. These different states should ideally activate the device’s full functionality, which leads to the functional verification of the device.

Coverage-driven verification is achieved through a software test bench created around the HDL used to design the DUT. As with a hardware test bench, the DUT is surrounded by devices used to provoke, measure, and model the operation of the device. While the software test bench, complete with all the required drivers, monitors, models, and scoreboards is time-consuming to implement initially, the method of coverage-driven verification is standard practice when it comes to verifying integrated circuits in the computer industry. This is due to the fact that for large integrated circuit designs, it is highly unlikely that directed testing would locate the vast majority of the bugs that could be present in the design.

The Universal Verification Methodology builds upon the idea of coverage-driven verification, and improves upon the basic framework of such verification in a number of key areas. One improvement over the conventional model of coverage-driven verification is the notion of sequences. Drivers make use of sequences of items created inside the sequences themselves; these sequence items are then passed to a sequencer which then dispatches the sequence item to the driver to stimulate the DUT. This granularity of the sequence/driver process allows for fine control over the process of stimulating the DUT, and can allow the straightforward reuse of sequences throughout different tests and test benches. In many cases, these sequences can be transported from test bench to test bench without any modification. If modifications are required, sequences and the sequence items used for transactions themselves can be easily overridden using the factory integrated into UVM.

When verifying large integrated circuit designs, relying on the method of directed testing alone will lead to a device that could perform incorrectly when stimulated with data even slightly different from the data used in the test bench. Coverage-driven verification is far more likely to generate edge-cases and other errata in the DUT as compared to directed testing simply because more random data will be generated. With UVM, the creation of well-designed test benches is made easier with features such as sequences, phasing, and register models. While initially time-consuming to create, verification test benches using UVM are more portable across IC designs and are more likely to locate errors in DUTs, saving both time and resources in the verification stage of ASIC design.
Recommendations

While the Universal Verification Methodology is considered the new standard in verifying large-scale ASIC designs, certain improvements to the UVM framework and the SystemVerilog language used to create the UVM base class library would benefit the task of architecting the test bench and the constrained random test cases.

SystemVerilog is an object-oriented language, and so “objects” can be instantiated and take advantage of object-oriented features such as polymorphism, inheritance, and encapsulation. However, not all objects in SystemVerilog strictly obey the tenets of object-oriented programming. Coverage groups, instrumental to the capturing of functional coverage, are one area of SystemVerilog that do not perform as would be conventionally expected in an object-oriented language, as these coverage groups are not “first-class” entities. A first-class entity allows an object to have certain properties, such as overriding types or having a distinct intrinsic identity [10]. Therefore, when a verification engineer is required to edit coverage defined in a previously-defined coverage group, the engineer must find another solution than simply overriding the the existing coverage group with the required modifications. One solution involves the creation of a subclass of the class containing the coverage group; one can then define a coverage group with the required modifications in the new subclass. The UVM factory can then be used to override the class containing the original coverage group with the new subclass. This process is far more complex than simply overriding the coverage group directly, and hampers the reusability of the code. It is recommended that in a future iteration of the SystemVerilog standard, coverage groups are given first-class status and therefore are allowed to be overridden and extended.

As the name implies, the very nature of complete constrained-random verification relies on the generation of random values to stimulate the DUT. However, these values are never truly random, as they are constrained by both the intrinsic predictability of a computer and those imposed by the verification engineer in the test bench. The software that solves the constraints imposed on specific values in a test bench is called a solver, and these solvers rely on values called seeds passed to the test simulation at runtime to help generate random values. While software constraints are often written with statistical distributions imposed on them, the solver has no knowledge of what values have previously been produced by itself for previous test runs and so must rely on complex algorithms to properly generate values that agree with the statistical model. It is recommended that EDA vendors investigate methods that could do away with complex algorithmic solving and generate statistically-agreeable data. One method could be simply saving each value the solver has produced for a particular test bench: though this method would lead to a huge amount of memory used, this would allow the solver to have knowledge of previous values and thus would allow it to adjust its values to more uniformly agree with the statistical models declared in the test bench. Another solution could be to allow the use of open-source solvers to “plug in” to existing EDA simulation solutions. This would allow verification engineers to modify solvers for their particular test benches, potentially saving time and aiding in the uniformity and statistical adherence of generated values.
References

Cited References


General References


