How to Automate a Complete Register Verification Environment

Author: Anupam Bakshi

Version Number: 1.1

Abstract: In-house tools and register code generators have proven to be quite valuable in simplifying register design and verification. As SoC complexity continues to arise, verification continues to consume most of the project life cycle. Verification teams have to manually create the UVM test environment and dynamically adopt it to the changes in the specification. This white paper discusses the need for a comprehensive automated register verification. Central to the paper is Agnisys' solution for automatically generating a complete UVM test environment that provides 100% functional coverage via simulation and formal.

Introduction

The sheer number of register bits, their access types, properties and the functionality which they control can be staggering in modern designs. Although in-house scripts and register generators have been commercially available for several years and these generators may be capable of generating register models for RTL and verification, most lack in several key areas and do not provide 100% functional coverage. This white paper discusses the need for a comprehensive and automated register verification. Pioneered by Agnisys, the solution we will describe meets the need and guarantees that the specifications for registers match their implementation throughout the process.

The Problem

If, for example, you have a 32 bit address bus, you can access 2^32 memory mapped registers. If each of the registers are themselves 32 bit wide, the total register bits will be (32 * 2^32) or 2^37 or 137,438,953,472. What if the address bus is 64 bits? Even though all of the addresses are not used, the number of used address locations is constantly rising because designers may need to add more and more functionality and re-configurability to their designs. Thus, the size of the IP and that of the SoC is constantly growing, and the task of managing the large body of data associated with registers is growing with it.

Existing register generators may be able to take a specification in the form of a formatted text file such as a Word or Excel document or a SystemRDL or IP-XACT XML files and produce RTL for the register map and UVM code for verification among possibly other output formats. Often Perl/Python Scripts are used to accomplish this. However, once the design verification (DV) engineer gets the model, their work hasn't ended, it is just beginning. The DV engineer then needs to create the complete verification environment and take the generated models and integrate them into their verification environment. Additionally, the engineer must then create the tests and the sequences based not only on the access to the registers and their properties but also depending on the functionality of the design.

On top of this, special types of registers and special functionality associated with many of the design registers exacerbate this complex task. Following is a list of some of those special types of registers:

- Interrupt registers
- Counter registers
- Shadow registers
- Aliased registers
- RO-WO pair registers at same address
- Muxed registers
- Wide registers
- Triggered Buffered registers
- Modal registers
- Multiple bus domain registers
- Constrained register

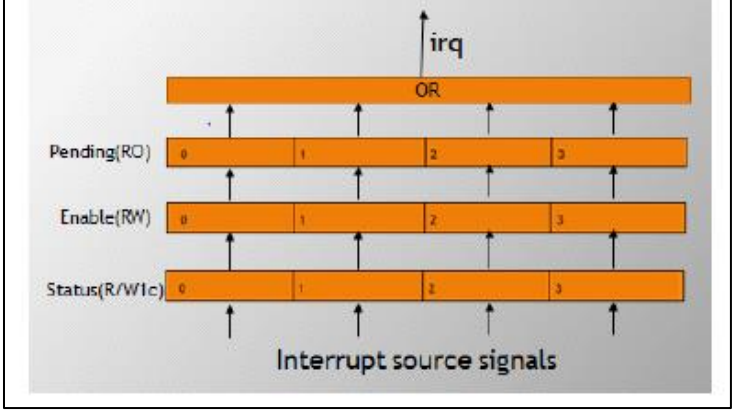

Figure 1: Register Example: Interrupt Register

The use of UVM for register verification is widespread because it offers countless built-in features which facilitate verification of registers in an address map. However, although UVM is an effective methodology, it is often a difficult and very time-consuming task for new teams to setup the complete, integrated environment. And, if changes are made to the specification, the implication is more time-consuming modification to this complex web of associated files, the environment and the tests.

Formal register checking offers an alternative to simulation based verification. This technique has the advantage that it provides exhaustive proof of register correctness, apart from the fact that register checking can take place much earlier, even before the simulation environment is ready and integrated. Failures can be conveniently debugged using the counter examples provided by the formal verification tool.

Formal verification has its own unique requirements. It requires capturing accurately the properties of all the register fields from the specification and SystemVerilog assertions to enforce the checks. In addition a top level file is needed to bind the assertions with DUT module –this ensures that the DUT RTL remains undisturbed while the assertions are hooked-up. In addition, a separate script to configure and run the formal tool will also be necessary.

In summary, register verification is a significant challenge. Registers are the interface by which software can talk to the underlying circuitry. Every configuration that is done on the chip is done by setting one or more bits in the registers that control them. Handling the sequence of configuring bits and creating test cases to exhaustively test them is challenging. Likewise, advanced knowledge of SVA is needed to accurately generate formal checks for every type of register field and their access policies. Any change to the specification entails updates to these properties and assertions. Even if a large team is involved in the verification project, the mere size and number of the test cases to be written can be so large that the time and money involved can become exorbitant. And if shortcuts to reduce time and tests are employed, greater risk is introduced into the project.

Why Automatic Register Verification?

One may argue that if the “Register Generator” (even if it is done through Perl/Python Script conversion) is being utilized to create a register model and the RTL, then what is the need to still verify the registers that were generated? As the reader of this document can probably attest to, the entire verification environment is still required. And this is typically done manually.

Not everything is auto-generated, there are interface points between the generated and the manually created RTL. These interface points need to be verified. For example, a hand crafted “external” register, or a memory instance or the rest of the application logic that interfaces with the auto generated registers.

Further, the verification environment should be complete with the software side agents and application hardware side agents. Note that the application hardware side agents are not fixed, they depend on the specification. For example, if the engineer changes the hardware access for a bit field from read-only to read-writeable, a change to the hardware side agent is required.

Next, the generated register models need to be integrated into the verification environment. The generated model must have the structural information about the registers, bit fields, their access and

addresses. Additionally, it should be comprehensive, and should also have cover groups and cover points with illegal bins, otherwise the quality and completeness of the tests cannot be meaningfully ascertained. Transactions from the Software and the Hardware application logic should be created. Tests must be written to test the variety of special functionality associated with the registers.

Running all these tests generates a lot of simulation data thus a dynamic test plan is needed. We say "dynamic" because a change in the specification needs to be reflected in the plan. For example, a new field added, or access changed may need additional tests. Such a test plan must be auto-updated once all the tests have been run, otherwise the project risks missing critical tests that could cause a bug to make its way into the chip.

Formal verification environment is comprised of the properties, assertions and sequences related to the register access policies described in the specification. A top-level module binding the properties and assertions with the DUT/RTL, and command script to compile and run the formal tool is also part of the environment. As in the case of simulation, any change in the specification leads to a change in one or more components of the formal verification environment.

Currently Available Solutions

From a recent survey of chip and IP designers and verification engineers, it has become apparent that the task of creating the verification environment using UVM requires a significant effort. According to some customers, this effort can amount to approximately 70% of the engineers' time. However, an automated way to create such an environment is not available commercially.

Creation of the UVM environment "from scratch" takes extensive time and meticulous attention even if engineers initially do it, maintaining it manually each time there is a change in the specification is also burdensome. Thus, automation of register verification has become a key requirement as semiconductor designs and IP become more advanced. SoC's are growing in complexity. Therefore, the corresponding address maps are becoming unmanageable by manual methods. Old techniques of manually performing verification or automatically generating only some parts/ components of the verification environment are no longer enough or adequate. The only successful strategy is AUTOMATIC REGISTER VERIFICATION

How to Automate a Complete Register Verification Environment

The good news is that all this can be automated to a great extent so that verification teams do not spend time developing the required verification environment, the tests and the test plan. What are the requirements for an automatic register verification solution? The automatic register verification solution should embody these capabilities:

A. It can take in Address Map specification in the form of MSWord/ MSExcel/ FrameMaker or textual standards like SystemRDL or IP-XACT or RALF
B. From this specification, create the following for a simulation-based environment:

1. Generate the RTL model (SystemVerilog / SystemC)
2. Generate the UVM Reference model (SystemVerilog / SystemC)
3. Take in additional information such as the register buses used for the address map, the least addressable unit, hdl_paths of each register, register group, memory or block.

4. Generate agents for the register buses. These agents must include the corresponding drivers and monitors.
5. Generate hardware side agents including the hardware side driver and monitor. The hardware-side driver helps simulate the transactions that are sent to the address map by the application logic. The hardware monitor is used to capture the same transactions used to update the UVM Regmodel with correct values.
6. Generate a hardware side predictor which gets these transactions from hardware monitor and predicts the correct value in the UVM model according to the access policies of the Register fields.
7. Generate a UVM environment class which contains all the above components
8. Generate sequences for each and every register according to their software access and additional special behaviors.
9. Generate UVM Test class to run all the above sequences.
10. Generate top level module connecting the generated RTL module and bus interfaces.
11. Generate a "Makefile" to run the simulations and collect results from simulation database appropriate for the simulator being used.
12. Generate a verification plan with ability to back-annotate these simulation results so that engineers can analyze the results.

C. For a formal verification environment, create the following:

1. Generate SystemVerilog properties and assertions to check the register access policies
2. Generate top-level file to bind the DUT RTL as well as third-party design IP with the assertions.
3. Create a make-file or TCL command script to run the formal verification engine.

The following is the complete, integrated environment that is to be automatically generated.

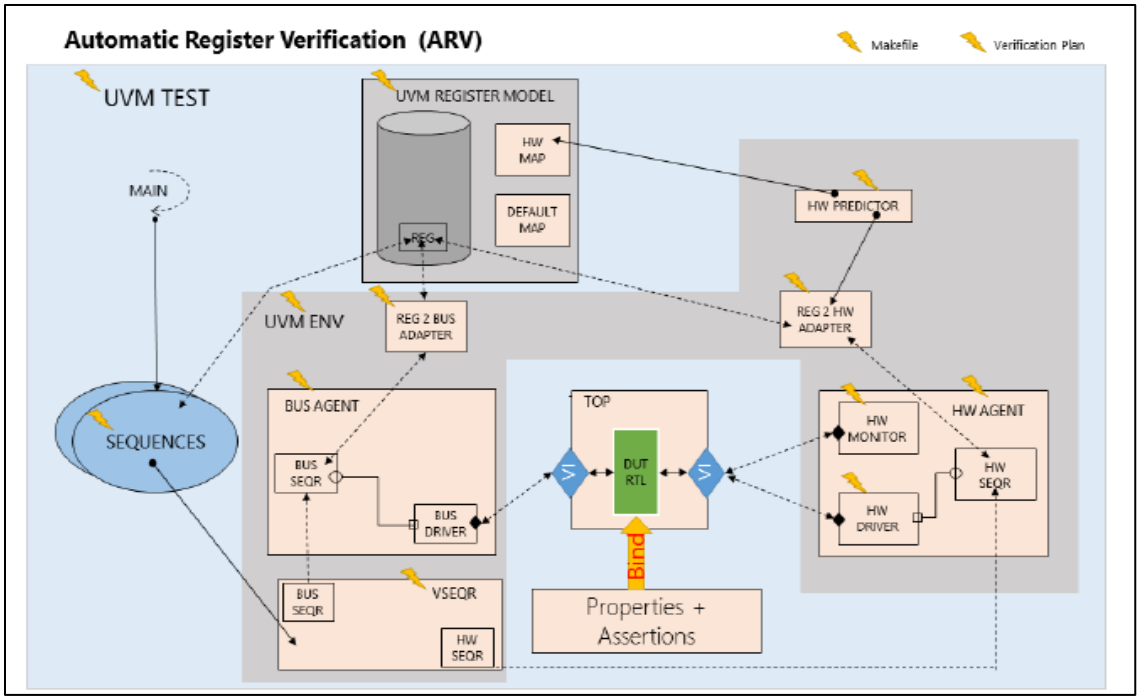

Figure 2: Components of Automatic Register Verification

Agnisys Solution

Automatic Register Verification (ARV™) helps with auto-generation of UVM test-bench: bus agents, monitors, drivers, adaptors, predictors, sequencers and sequences, enabling engineers complete the register verification right in the first run.

ARV automatically generates UVM sequences for their advanced verification covering all possible input scenarios and generates an effective verification report in HTML format. This report consists of a register verification plan, which shows the complete summary of the register verification along with their functional coverage and test status that correspond to the register component. The report also shows all the register elements along with the corresponding metric groups, showing the overall coverage for that element and relative test details which has been passed or fail.

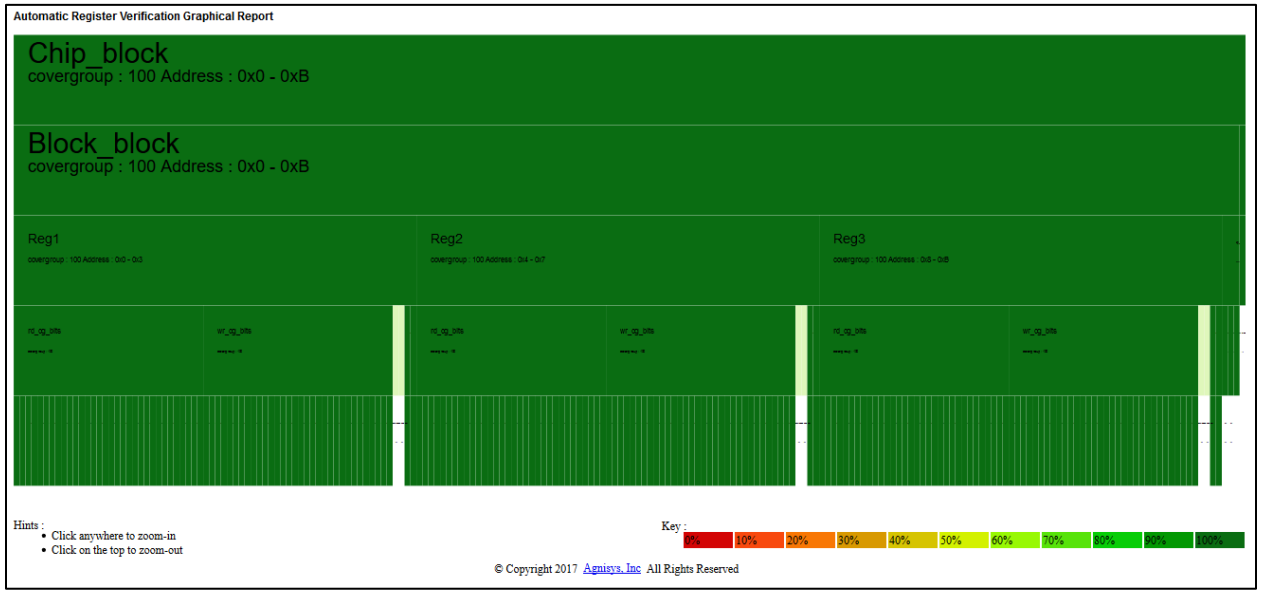

Figure 3: Components of Automatic Register Verification

ARV helps to auto-verify the special registers like Lock Register, Indirect Register, Constraint Registers and all Register Accesses etc. ARV coverage report which shows all the contents summary in a tabular form, specifies the register model summary, their overall coverage and number of test used along with the status.

With the ARV-Sim™ capability, the engineer only needs to do the following:

1. Create a Register Specification in any format such as SystemRDL, Word, Excel or IP-XACT.
2. Use the automatic ARV module to generate all of the above components, files, and integrated environment using the specification as an input.
3. If RTL from another source is also to be verified, perform custom stitching at the top level module.
4. Run simulations using the generated Makefile.
5. Analyze the results with the automatically generated verification plan.

With the ARV-Formal™ module, the DV engineer only needs to do the following:

1. Create a Register Specification in any format such as SystemRDL, Word, Excel or IP-XACT.
2. Use the ARV-Formal module to automatically generate all the property and assertion files, top-level bindings with DUT and scripts to run the formal tool.
3. Optionally include assertions to check protocol compliance of standard buses used by third-party design IP blocks
4. Run the formal verification tool using the auto-generated scripts.
5. Debug any failures using the counter example provided by the formal tool. Exhaustively verify all register access policies, saving significant effort and time.

Conclusion

This white paper has shown the challenges of manual register verification and the benefits of doing it automatically. At Agnisys, our mission is to empower the engineer to focus on product verification and design, and not spend time on what can be fully automated. From our perspective, SoC teams can choose to develop a limited and difficult to maintain in-house tool or utilize a fully supported and comprehensive commercial product that is well-proven and utilized by several international SoC companies.

Company Overview

Agnisys Inc. is the leading supplier of Electronic Design Automation (EDA) software for solving complex design and verification problems for system development. Its products provide a common specification-driven development flow to describe registers and sequences for system-on-chip (SoC) and intellectual property (IP) enabling faster design, verification, firmware, and validation. Based on patented technology and intuitive user interfaces, its products increase productivity and efficiency while eliminating system design and verification errors. Founded in 2007, Agnisys is based in Boston, Massachusetts with R&D centers in the United States and India. www.agnisys.com

For any query, please mail us at info@agnisys.com

Boston
75 Arlington St. Suite 500
Boston, MA – 02116
1 (855) VERIFYY (1-855-837-4399)
1 (866) 927-3653 (fax)

Noida
D-343, Sector 10
Noida, India - 201301