Leveraging Formal to Verify SoC Register Map

Abdul Elaydi
Marvell, Inc
5488 Marvell Ln,
Santa Clara, CA 95054
aelaydi@marvell.com

Jose Barandiaran
Cadence Design Systems
12515-7 Research Blvd
Austin, TX 78759
joseb@cadence.com

Abstract

Today's SoCs are built up of many IPs and subsystems. The inability to properly control or receive status from these components can cause significant issues and even result in a dead chip. Unfortunately, issues around the control and status registers (CSRs) are fairly common and cannot typically be "fixed in software" as this is the layer that interacts with the software. Historically people have used simulation-based approaches to validate CSR functionality but these methods are insufficient as they are not fully exhaustive.

Exhaustive verification of an SOC register map is one of the verification goals at Marvell. Traditionally, verification has been done using directed tests issuing read and write cycles targeting the register to verify the register access policies and reset values, based on functional description in the design documentation. While this approach can verify the register map to a certain degree, it is not exhaustive with regards to providing comprehensive data pattern and address aliasing testing capabilities. The effort required for a user to define all the possible permutations of data patterns and register accesses is simply prohibitive.

Formal analysis addresses this verification challenge. Using an IP-XACT description of the register map generated from the register documentation, assertions can be created automatically to verify the register access policies. Using formal analysis to prove the assertions provides exhaustive verification, without the need for testbench, with a typical turnaround time on the order of minutes or less per check. Additionally, since formal analysis can prove assertions independently of each other, compute farms can be leveraged to allow many assertions to be verified in parallel, greatly reducing total turnaround time. Debugging assertion failures from formal analysis is typically easier than with simulation approaches, since formal will automatically find the shortest path to the failure, and can provide a waveform counterexample showing the failure.

The paper reviews the results of using this approach on Marvell designs, and specifically highlights a problem that was missed by simulation but caught by this technique.

Keywords—register validation, formal verification, formal techniques

1. Introduction

The number and complexity of design components like registers used in an IP are growing with each subsequent version of a system-on-chip (SoC) design. The inability to properly control or receive status from these components can cause significant issues and even result in a dead chip. Unfortunately, issues around the control and status registers (CSRs) are fairly common in today's SoC designs and cannot typically be "fixed in software" as this is the layer that provides the interface between the hardware and the software. Historically people have used simulation-based approaches to validate CSR functionality but these methods are insufficient as they are not fully exhaustive. A flow which can automate the task to validate the use of specified registers is a key requirement for IP developers and verification engineers.

This paper describes a register validation flow which addresses the register map validation requirement. The flow leverages formal verification to exhaustively check SoC Register Maps, which are specified using IP-XACT descriptions of the register database. The flow converts register documentation into IP-XACT format using in-house scripts, then a new commercially-available formal verification application is used to generate assertions, to fully verify register access methods.

2. Background and Motivation

At Marvell, we had experienced problems with the time needed to setup and execute simulation-based register verification, which not only took too long but failed to provide conclusive evidence that verification was fully achieved for the entire register map. This led to requirements for both speed-up and completeness. We wanted to have a complete verification environment, capable of testing at the SoC level, that would verify all the SoC and IP-Level registers against latest specification updates without need for elaborate setup.

3. Pre-existing Solutions and their Limitations

The previous verification environment for register map utilized directed tests written in a C-language environment for all SoC Registers. We found it very difficult to create C-based directed tests to verify every situation we needed to test, and in particular, we were not able to resolve issues related to register duplication, over-lapping or out-of-range addresses. We did not consider using UVM_REG (the base class for the register modeling layer in UVM) due to the environment setup and ramp time involved, especially since UVM in general is not yet fully adopted by our team.

4. Formal Register Map Validation Flow

The register validation flow is based on the following three key components:

- IP-XACT: An essential repository of information, which provides details about

memory maps and registers that are part of IP.

- Assertions: These are used to validate the behavior of the design.
- Verification Using Commercial Formal Engine: Performs the RTL block verification. The formal, assertion-based approach and exhaustive analysis ensures verification and detection of errors.

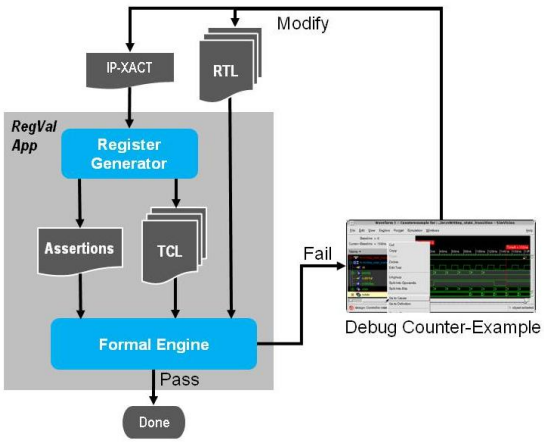

Figure 1: Register Map Validation Flow

Figure 1 represents the various elements of the register validation flow and their interaction with each other. These key components are now explained in detail.

IP-XACT, now a formal IEEE standard (IEEE 1685-2009), is widely used for register and memory map management. IP-XACT provides XML schema for vendor neutral IP descriptions, which are used to conveniently capture details about interfaces, signals, ports, parameters, memory maps, registers, files, and also manage design changes with each revision. This enables easy design assembly with different configurations, reduces development and verification time and cost, and leads to simpler data sharing. The IP-XACT specifications are used to build a register validation flow to verify the design.

In addition to the register IP-XACT we also needed to supply SystemVerilog Assertion (SVA) sequences for read, and write sequences in addition to the design reset sequence. The flow is extensible and not specific to an interface protocol. For common protocols, such as APB used in this design interface, the read and write sequences existed in a library, which we used to run read and write sequences to the design interface.
The supplied APB SVA Write sequence (WRITE_SEQ) is:

```
PSEL && !PENABLE && PWRITE &&
(PADDR == REG_chk_addr) &&
(PWDATA == REG_chk_data)
##1 PSEL && PENABLE && PWRITE &&
PADDR == REG_chk_addr && PWDATA
== REG_chk_data
```

The supplied APB SVA Read sequence (READ_SEQ) is:

```
PSEL && !PENABLE && !PWRITE &&
(PADDR == REG_chk_addr)
##1 PSEL && PENABLE && !PWRITE
&& (PADDR == REG_chk_addr)
```

REG_chk_addr is a predefined register that represents the target register address being tested.
REG_chk_data is a predefined register that represents all the data patterns driven on the write data bus on the interface.

The sequences were combined to create testing sequences to verify

access policies as shown in the discussion around read-write access policy checks below.

The reset sequence (RESET_SEQ) is specific to each design and for this design the following SVA sequence represents the reset sequence for our design. This design required PRESETn to be driven low for 20 clocks and p_rst to be driven high for 20 clocks. There is a requirement for the reset sequence to terminate with one clock of the reset deasserted. The last term below represents both reset signals deasserted.

```
((!PRESETn & p_rst)[*20] ##1
(PRESETn & !p_rst))
```

The sequences are combined with the register IP-XACT to create and extended IP-XACT file as illustrated in Figure 2.

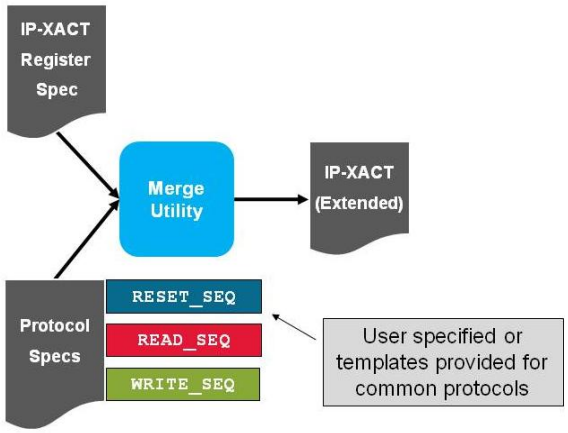

Figure 2: Extended IP-XACT Creation

Generation of appropriate assertions is the next step in the register validation flow as illustrated in Figure 3. The result is a BFM to drive the design interface along with the checks for each access policy in the design. The register generator utility reads the IP-XACT specifications, and generates a set of output files which are compiled and run using Incisive Enterprise Verifier (IEV) from Cadence Design Systems. The generated files include an SVA file which embodies properties, along with bus sequences to verify configuration register policies; .tcl files for executing each type of access policy checks (Reset, RW, RO, etc.); and a text file with information of how to bind the created SVA file to the top-level design module.

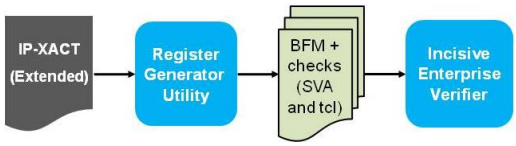

Figure 3: Assertion Generation and Execution

By default, all checks run all sequences back to back. In order to find all bugs, the concept of gap activity has been introduced. The gap activity induces chaos between read and write sequences in the checks in the case of the read-write policy check. It controls the additional free activity regions where any activity except writing to the address it is checking can be done. Checks support both front and back door accesses. The front door checking happens through interface protocol, whereas, back door checking happens through direct register access. Figure 4 illustrates the concept of gap activity, in this example for Read/Write checks. For this Read/Write example, no target write is issued during the gap activity period.

While formal technology by default can explore all possible sequences of interface/design behavior the methodology takes an incremental

approach to controlling this behavior on the IP register access interface. In this way you can control the injection of "chaos" and use the opportunity to understand a design's sensitivity to this chaos. Knobs are provided in the flow to control the injected chaos.

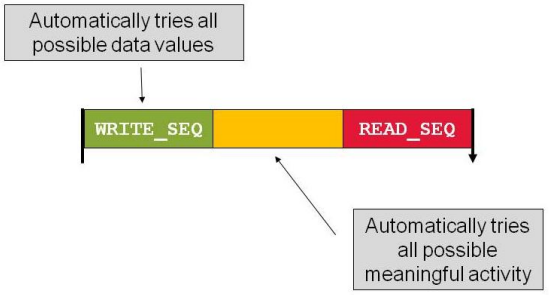

Figure 4: Read/Write Checks Showing Gap Activity

The list of checks for which assertions were generated is given below:

- Reset Checks
- Read-Write Checks
- Read-Only Checks
- Write One Set Checks
- Write One Clear Checks

The following examples illustrate how the generated checks in the respective files are executed using IEV. We specified the IP-XACT Component description XML file to IEV for executing the checks as follows:

```
iev -f ./apb_subsystem.f
+regval+component.xml
+rv_check+<check type>
[+rv_alias_covers]
```

where:

- `+regval` option specifies the component xml file to be processed. This is a mandatory option required for running the register validation flow in IEV.
- `+rv_check` option specifies which check to execute. The *<check_type>* can be `rw`, `ro`, `rst`,`w1c`, or `w1s`. This is a mandatory option and a single check needs to be specified for execution.
- `+rv_alias_covers` option creates read-write check covers that verify aliasing can occur in read-write checks when the middle gap is enabled.

The component.xml file was processed and the following output files were generated by IEV:

- `rvf_reg_check.svp`
- `rvf_ext_bind.txt`
- check-specific tcl files will be generated

5. Results

We ran the Register Map Validation Flow on multiple different IP blocks in our storage SoCs. The run time was around 2-3 hours each for IP blocks with 200-300 registers. The tests were run before RTL freeze and results were fed back to design and documentation teams for required updates. We experienced no limitations in the complexity of register IP blocks that could be described using IP-XACT. In fact, as anyone experienced with formal analysis might expect, the limit on complexity is rather set by the number of states in which the assertions need to be explored. To mitigate this, and since the checks are independent assertions, we could leverage multiple processors to evaluate them in parallel. While this parallel approach is not unique to

formal methods and has been supported by simulators for a long time, we found that the tool in this case supported parallel execution natively and conveniently. In fact, we found that the methodology scaled successfully to a subsystem consisting of a number of instances of the IP blocks described earlier, connected on an APB bus to an APB bridge.

The flow has helped clean up many design-versus-specification mismatches and register access policy (RW vs RO) errors. One example mismatch we were able to find was a register duplication issue. Here, two registers had the same address, and were located inside the same sub-IP but the read mux was hooked on to only one of the two registers at the SoC level. So a write essentially went through to both, but the read occurs from the wrong register. For the directed test, the read-compare always passed as the return value is always correct. However, the formal test failed in reset value check after gap-activity was introduced before the first reset value read cycle. We realized that we had been reading out the value from the wrong register all along in the directed tests.

All other issues found could also be covered in directed tests but required specific scenarios to be created to hit the failure (out-of-range, decoding errors etc.)

6. Future Work

We plan to extend the environment in IEV to run a single-shot test that drives the CPU AXI interface that has access to all IP level registers. This saves more time and resolves any address decoding issues at the lower levels that are caused by the design hierarchy of the register blocks.

7. Conclusions

The formal register map validation flow was able to meet our goals to accelerate register map verification, both in terms of setup speed and execution, and to improve completeness of verification. Fast setup was achieved through the convenience of using IP-XACT for register specification; through the availability of pre-existing read and write protocol libraries for our interface protocol; and through the flow's capability to automatically create assertions to check the register access policies. The goal for fast execution was met with runtimes of 2-3 hours for our IP blocks. Achievement of the completeness goal was demonstrated by finding many design-versus-specification mismatches, some of which could have been found if directed tests had been created for them, and others that were missed by existing directed tests.

References

[1] IEEE-Std 1685-2009, IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows, IEEE Standards Association, 2010

[2] Verification Apps User Guide, Product Version 13.1, Cadence Design Systems Inc., 2013