2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Step-up your Register Access Verification

Nisha Kadhirvelu, Cypress Semiconductor Technology India Pvt. Ltd., Bangalore, India (*nish@cypress.com*)

Sundar Krishnakumar, Cypress Semiconductor Technology India Pvt. Ltd., Bangalore, India (*skb@cypress.com*)

Rimpy Chugh, Mentor Graphics India Pvt. Ltd., Bangalore, India (*Rimpy_Chugh@mentor.com*)

Wesley Park, Mentor, A Siemen Business, Inc., Fremont, United States (*Wesley_Park@mentor.com*)

***Abstract*—Even small IPs have dozens of memory-mapped control & status registers, many with complex access policies. Additionally, verifying a retention registers' robustness early-on can save many simulation cycles down-the-road. While simulations leveraging UVM_REG can exercise the primary use cases, numerous corner-cases are left uncovered, creating the risk of show stopper bugs going undetected. In short, a formal-based solution is needed to ensure exhaustive coverage of the state space [1]. Hence, in this paper we will show how we replaced our simulation-based flow with an automated formal-based flow to reduce our setup and run time from weeks down to minutes, simultaneously enjoying exhaustive results.**

Keywords—memory-mapped registers, register access policy, retention registers, formal solution

I. INTRODUCTION

The configuration and run-time behaviors of all our IPs are governed by memory-mapped registers. They reside in address spaces that are accessible with interfaces like AMBA, and they have a wide variety of bit widths, internal fields, and access policies (W, RW, RO, W1C, etc.) These control & status register banks are effectively the heart of each IP, so if there is a bug in them, the corrective surgery (with firmware patches and such) can usually cure the patient, but the issue will be expensive to diagnose, and the ongoing health / performance of the patient will be forever compromised.

Consequently, we must diligently verify that the RTL implementation of the registers meet their specifications. As such, our verification checklist starts with:

- Checks for register accessibility
- Address correctness
- Correct default values on reset
- Basic Write-to/Read-from checks
- Tests for the access policy implementation
- Correctness of all the fields
- Stability of the register
- Front-door and Back-door access

Multiply this by a dozen to a thousand in a larger IP, and it's easy to see that the scalability requirements of our register verification can threaten our resource capacity and project schedule.

II. PRIOR METHODOLOGY AND CHALLENGES

A. Simulation approach

We originally employed a conventional, constrained-random, coverage-driven testbench simulation approach. Specifically, once the overall DUT testbench was far enough along we would integrate in the register modeling to the environment, write parameterized tests and coverage points for each register, run the simulations, analyze failures, track code and functional coverage, etc.

All of this must be effectively repeated if the given DUT has multiple operational configurations. In short, it was a state-of-the-art testbench simulation flow.

B. *Challenges*

The problem for us was that this simulation flow started breaking – becoming un-scalable when the DUT configuration changes & IPs complexity raises (due to increase in number of registers). For starters, even for teams that are very experienced in SV/UVM and are expert script writers, the number of tests and sequences for the "easy" registers were getting to be unmanageable (in-spite of automation built around it).

Adding to this was the time-consuming (and error prone) process of understanding and developing tests for the unconventional, customized register access APIs, longer run-time & coverage closure. Inevitably some corner cases were initially missed and not caught until late in the design cycle.

Further challenges with our IPs include lesser reusability of the setup for different configurations of the IP, modelling & verification of interdependency between registers, accounting of hardware latency in updating the registers plus the need of separate agents & test cases for retention checks. Above all, the major downside is that the flow must be iterated for various register interfaces like AHB5, APB, etc., that are standard protocols (or) any custom interface.

III. NEW PROCESS

A. *Formal approach*

Being aware of formal analysis' ability to deliver exhaustive verification of control logic, we adopted a formal-based flow to exhaustively verify our registers. Questa RegCheck tool was used for this approach. The benefits were immediate: there was no requirement to create a testbench/register model/test setup – the formal-based register checking application takes-in the XML or CSV register spec as input directly, automatic negative checking capability, support for front-door & back-door accesses.

Along with the DUT RTL, the formal tool generates all it needs under-the-hood and illustrates any discrepancies if it finds between the spec and the DUT with "counter example" waveform showing how the spec can be violated by the current DUT implementation.

B. *Flow*

As shown in the Figure 1, this simple flow takes a register spec, an interface spec, and any necessary constraints along with the DUT RTL as input, generates the properties and binds them to the in-built assertion module (in case of standard AMBA interface) / custom interface module & stores the results in the form of status logs and waveforms.

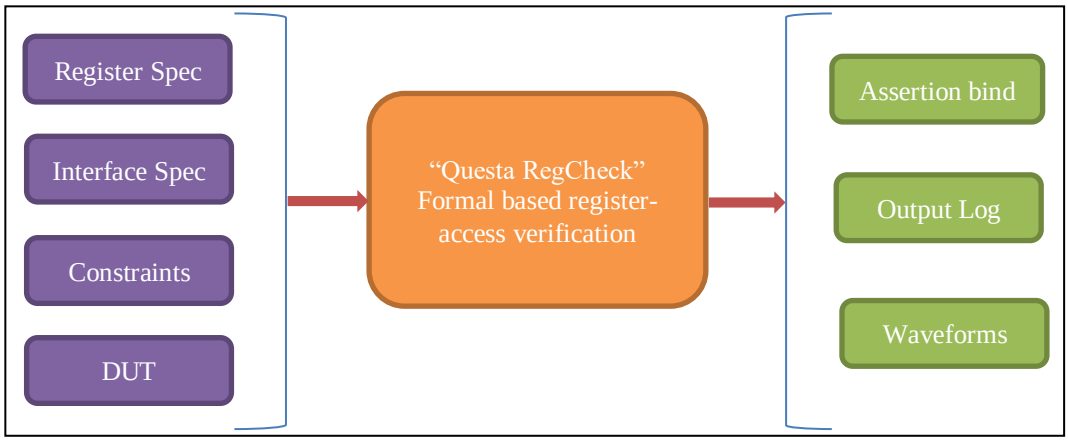

Figure 1. Formal approach – A Pictorial representation

C. *Inputs*

The first input for this is the register spec in CSV format that followed IP-XACT register access policy descriptions. It is described in the specified format below:

```
Offset,Position,Title,Identifier,Array,Access,readAction,modifiedWriteValue,volatile,Type,Reset Value,Reset Mask,Description,.memmap_write_internal,.memmap_resetn_global,.memmap_resetn_local_as
ync,.memmap_write,.memmap_write_address,.memmap_write_data,.memmap_write_mask

0x004,,comp0_status,comp0_status,,read-write,read,write,TRUE,register,0x00000000,0x0000ffff,"comparator_structures_comparator_0_status"
,[15:0],comp0_out,comp0_out,,read-only,read,write,TRUE,configuration,0x00000000,0x0000ffff,"Active_comparator__comp0_out____outputs_"

0x700,,intr,intr,,read-write,read,write,TRUE,register,0x00000000,0x0000ffff,"interrupt"
,[15:0],comp0,comp0,,read-write,read,oneToClear,TRUE,configuration,0x00000000,0x0000ffff,"This_interrupt_cause_field_is_activated__HW_sets_the_field_to__1___when_a_comparator_0_event_is_generat
ed",([mxevtgen_top.act_0.mmio_0.mmio_int_set_comp0_sw_set)
```

Figure 2. Partial Snippet of CSV format

While the definition of the CSV is simple, the effort in creating it for different configurations of the IP increases with increase in registers. To overcome this, an in-house script was developed to automate the CSV generation from our register specification format, which is only a ONE-TIME effort.

```
-register        act_0.mmio_0.mmio_regs_0.$register_$field
-interface       amba_ahb
-base_addr       0x403f0000
-spec_type       ipxact
-signal_match    nocase,prefix,postfix
-interface_port hselx = mmio_hsel
-interface_port haddr = mmio_haddr[31:0]
-interface_port htrans = mmio_htrans[1:0]
-interface_port hwrite = mmio_hwrite
```

Figure 3. Interface definition

Next input is the interface spec which is to be stated in a simple .txt format as in Figure 3. All it includes is the path of the register module in the design, IP AMBA/custom interface, the base address of the IP, format of register specification used (IPXACT/UVM), mapping of the DUT's interface to the tool's inbuilt assertions module ports and the pointer on back-door (or) front-door access.

Third is the constraint file as shown in Figure 4, that incorporates any constraints on the reset signals, clock frequencies, specific inputs/assumptions to the DUT IO's for the formal run, black-boxing of modules along with formal compile & verify commands.

```
onerror {exit}
netlist clock clk_ip -period 10
netlist constraint rst_ip_n -value 1'b1 -after_init
netlist property -name haddr_used -assume {mmio_haddr[11:0] < 12'h800 && mmio_haddr[11:0] > 12'h000}
formal compile -d ip_top -cuname BIND_QFL_MMREG_ip
formal verify -timeout 60m -sanity_waveforms -init qft_files/init.seq
exit
```

Figure 4. Constraint input file

And finally, the DUT compilation file list should be passed together with the top-level configuration parameters.

After these inputs were provided, the tool generated all the necessary verification environments including target properties to verify the register behaviors and then the exhaustive formal engines assure the complete behavior verification of the given registers. The target properties are designed to cover complete behaviors of individual registers that comprises of:

- Global/local reset behaviors
- Bus read/write operations
- Any conflict behaviors
- Volatility
- No operation (where the register value should not be modified by any other transactions on the bus)

D. *Outputs*

Assertion bind output (Figure 5) that is generated from the tool contains information on the in-built assertion module that is instantiated for every register field, with a instance name created as a concatenation of assertion module name, register base + offset address, register/field name, access policies.

```
qfl_memmap_reg_ipxact_backdoor
#(
  .ADDR_WIDTH          ( ADDR_WIDTH ),
  .ADDR_MASK           ( 0 ),
  .ADDR_VALUE          ( baseAddr0 + 'h0004 ),
  .DATA_WIDTH          ( DATA_WIDTH ),
  .DATA_LEFT           ( 15 ),
  .DATA_RIGHT          ( 0 ),
  .DATA_RESET_GLOBAL_MASK  ( 'h0000ffff ),
  .DATA_RESET_GLOBAL_VALUE ( 'h00000000 ),
  .REG_WIDTH           ( 16 ),
  .REG_OFFSET          ( 0 ),
  .ACCESS_POLICY            ( "read-only" ),
  .ACCESS_READ              ( "read" ),
  .ACCESS_WRITE             ( "write" ),
  .ACCESS_VOLATILE     ( "true" )   )
QFL_MEMMAP_IPXACT_BACKDOOR_0x403f0000_0x0004_comp0_status_comp0_out_read_only_read_write_volatile
(
  .memmap_register     ( ip_top.act_0.mmio_0.mmio_regs_0.comp0_status_comp0_out ) ,
  .*
);
```

Figure 5. Assertion bind to a register field

Once the formal engine is run, the status of the properties is logged as one of the 6 types of status below:

- Assumed – The assumptions as passed in the constraints file
- Proven – Number of passing properties (like read, write operation)
- Covered – Number of properties that are covered
- Inconclusive – Properties that could not be concluded by the formal engine as proven/dis-proven
- Fired – Number of failing properties
- Uncoverable – Number of properties that could not be covered in anyway

Any failures could then be debugged with a simpler Visualizer GUI (Figure 6) that dumps all related signals causing failure, failing timestamp, clear transaction on the failure point for all the assertions.

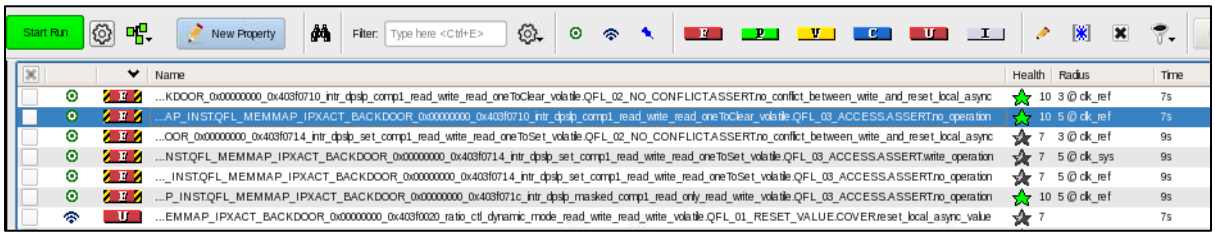

Figure 6. Assertion bind to a register field

IV. RESULTS

We deployed this new process in 3 of our IPs – 2 with APB/custom interface & 1 with AHB-Lite interface. The result of this was a drastic improvement in register verification that helped us in faster IP-level verification. Table I, II represents the results of IPs that uses APB interface. Since a simpler implementation of APB interface is supported by the design rather the standard protocol, a custom interface was declared & used for these IPs.

Table I. Results - APB IP#1

S. No	***Number of registers: 8***	
	Property	***Count***
1	*Assumed*	*2*
2	*Proven*	*220*
3	*Covered*	*127*
4	*Inconclusive*	*0*
5	*Fired*	*0*
6	*Uncoverable*	*33*
7	***TOTAL***	***382***
TIME TAKEN: 1 minute 14 seconds!!!		

Table II. Results - APB IP#2

S. No	***Number of registers: 5***	
	Property	***Count***
1	*Assumed*	*1*
2	*Proven*	*112*
3	*Covered*	*64*
4	*Inconclusive*	*0*
5	*Fired*	*0*
6	*Uncoverable*	*16*
7	***TOTAL***	***193***
TIME TAKEN: 1 minute 8 seconds!!!		

Table III below represents the result of the IP using AHB-Lite protocol. The time taken for this IP is little higher not because of the increase in registers compared to the above IPs, but due to the registers spread across multiple modules and so the formal was run separately for each module. The result is the consolidated one that has a overhead of formal compilation/loading/optimization time for each of the individual runs.

Table III. Results - AHB-Lite IP

S. No	***Number of registers: 64***	
	Property	***Count***
1	*Assumed*	*39*
2	*Proven*	*1596*
3	*Covered*	*873*
4	*Inconclusive*	*0*
5	*Fired*	*0*
6	*Uncoverable*	*267*
7	***TOTAL***	***2775***
TIME TAKEN: 18 minutes!!!		

A. Verification of Retention Checks

Additionally, the AHB-Lite target design has a separate power island in it, and hence, has state retention registers. At a RTL level IP verification this only means the toggle of retention/non-retention reset and verification of registers for retained/reset value respectively. This was much tougher with simulation due to the need of separate reset agents, proper handshaking of transactions to align with the reset toggle, etc., With the new process, we only toggled the non-retention reset keeping retention reset in de-asserted state via the constraints file, added a new column in the CSV to provide the retention reset name for the retention registers, and the retention registers could then be completely verified.

Table IV. Results - AHB-Lite IP retention registers

Number of registers: 56
TOTAL properties: 2332
TIME TAKEN: 14.7 minutes!!!

B. Verification of some complex registers

Some registers were designed to have complex volatility behaviors that are affected by other registers through variable delays. These complex behaviors cannot be described using IP-XACT register access policies. Instead, we could describe them intuitively using the tool provided debug signals. This allowed us to update the original register spec and we could finish our register verification completely with a push-button solution.

C. Issues caught

With this new process, issues with address decoding, overlapping of hardware & software modification to the registers and problems with definition on number of cycles between software write initiation and hardware update to the register were seen.

Clearly, a big benefit of this exhaustive complete register verification is that it enables early retention checking before starting compute and time-intensive power-aware sims. Ultimately, for a comparatively sized DUT that took months to verify, after a few of days of initial setup this new flow exhaustively verified the registers in under seconds to minutes!

V. Future Work

With successful usage with AHB-Lite & APB IPs, the process is also being tested on IPs with AHB5 interface. The next step on this deployment is to explore on the way to include all the registers spanned across different modules in a one formal run to reduce the time taken for IPs of type listed in Table III (there is a support in the tool already to pass registers present in different modules in a single run). Later to this will be the automation of creation of the full setup based on the standard protocols supported by the tool, while also supporting the custom interface.

VI. Conclusion

With this new formal-based flow, beyond the delivery of exhaustive results in an incredibly short amount of run-time, we now enjoy dramatically less user effort in environment setup with early retention checking. The setup itself is more portable and reusable, the compute resource needed is substantially less, and the failure analysis / debug is also quick and efficient. Consequently, this flow is now our plan-of-record.

Acknowledgment

I would like to thank Rimpy Chugh, Bathri Subramanian, Wesley Park from Mentor, A Siemens Business., for their support during this execution.

References

[1] “Questa Register Check User Guide”