Leveraging RAL and alternate automation (cocotb) techniques to improve Register Verification in UVM

Sougata Bhattacharjee,
Senior Staff Engineer, Samsung Semiconductor (SSIR), Bangalore, India
Email: Sougata.b@samsung.com

Introduction

- The utility of **Register Abstraction Layer (RAL)** is that the sequences and the code used can be reused even if there is a change in the DUT address map related to the physical registers which is not the case with the address-based verification

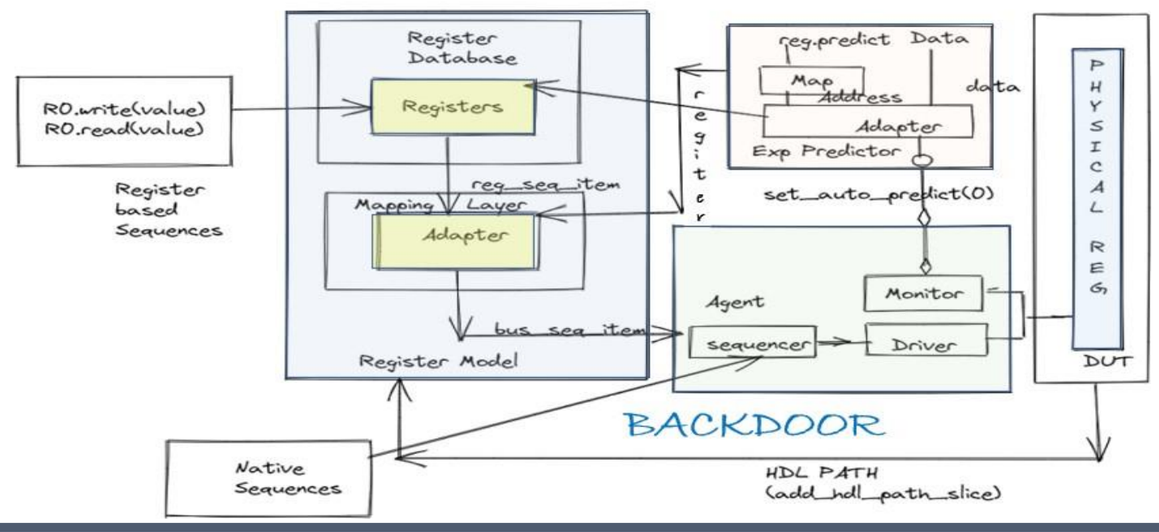
Register Verification - Categories:
Read/Write Test
Default Value or Hard / Soft Reset Test
RO and WO Test
Negative Testing
Aliasing
Broadcasting or Shared Test
Special Register Test

- Although RAL provides reusability but some of the sequences take time to complete, and this problem can be solved by **further automation** using Cocotb.
  - **Cocotb** is a coroutine co-simulation testbench implemented in python with support for SV/UVM constructs by utilizing the inbuilt libraries of cocotb in python (pyuvm).

RAL Implementation Steps

Step – 1:

Any register can be called by its name instead of the address from the sequence, and when it is triggered it maps the content of the register through the address map in the register model. The register model has all the information regarding the attributes and other access functionalities of the register through the register database. FRONTDOOR



RAL Implementation Steps

Step – 2:

It then produces the reg_sequence_item and with the help of an adapter and converts it into a bus sequence item. An **adapter** is bidirectional in nature and has two functionalities in the name of bus2reg and reg2bus.

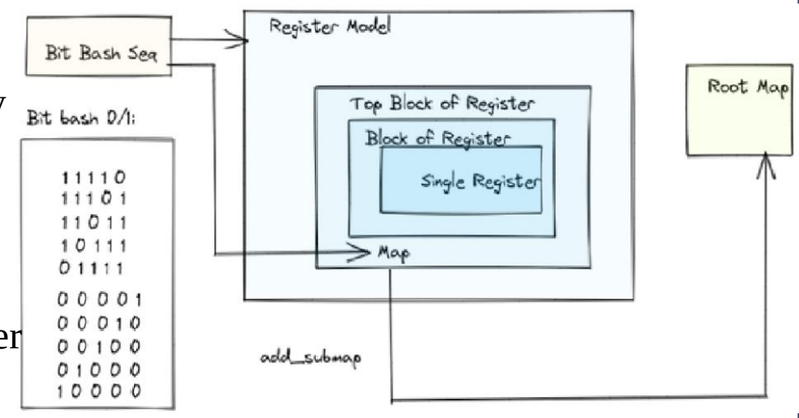
Step – 3:

- The transaction then reaches the Agent interface through which it accesses the contents of the physical register from the DUT. The response then routes back with the help of the predictor path and then reaches the adapter to update the contents of the mirror value. The mirror value is sitting in the register model, contains the current state of the DUT register, and is updated by the predictor after each write and read cycle. It is important to note that the mirror value should not be out of date. This process described above can be termed as **Frontdoor**.
- Alternatively, the register contents can be accessed directly with the help of HDL paths by setting up the add_hdl_path_slices in the reg model, and this type of access is known as **Backdoor** access.

Different Inbuilt Sequences in UVM

uvm_reg_bit_bash_seq:

a] Thorough and rigorous verification - checks every field of the register like a stream of walking 0/1 counter and hence can unearth critical bugs.



b] Ability to check whether any of the RTL signals are stuck at 0 or 1.

c] Automated and part of UVM RAL- saves a lot of coding efforts.

uvm_reg_hw_reset_seq:

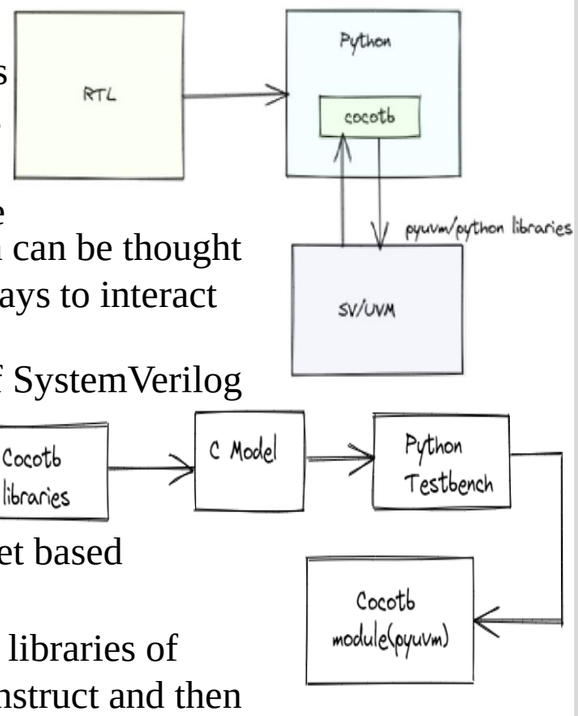
This sequence checks the default value of the register specified in the register model. It resets the DUT and reads all the registers in the address map range and then compares it with the mirror value.

uvm_reg_shared_access_seq:

The register model has some specific group of registers which has an effect on other sub-blocks within the address map through which it can be accessed.

Cocotb

Cocotb is a co-routine co-simulation environment that enables to write codes in verification-like software and it does this by connecting the RTL codes implemented in any HDL code with the python testbench through cocotb which can be thought of as a VPI. In general, there are two ways to interact with cocotb in python.



- The first is to build the interaction of SystemVerilog with the **C model**, and then the C model will interact with **python** with the help of Client-Server and Socket based connections.

- The other way is to utilize the inbuilt libraries of cocotb supporting the UVM RAL construct and then instantiate it with the **python** Testbench with the help of libraries called **pyuvm**.

Conclusion

1. The register access sequences present in the existing Testbench comprised of 453 lines of code, but with the implementation of Bit-bash algorithm the coding lines have been reduced to 278. So, the significant 61% decrease in the coding length has been very helpful for increasing the quality and saving time of verification.
2. Implementing RAL-based functionalities related to the inbuilt sequences has notably increased the efficiency of the existing Testbench. The difference in result can be observed in reusability, reduction in 30% of the coding effort, and increase in around 40% of the functional completeness in Register verification as compared to the existing Testbenches.
3. To reduce the simulation time and to check the HDL path, Backdoor checks have been introduced in addition to Frontdoor algorithms.
4. Cocotb has further enhanced the process of register verification by implementing the register access sequences in python which makes it 20% less verbose and further reduces the debugging effort.

REFERENCES

[1] Universal Verification Methodology (UVM) 1.2 User's Guide by Accellera
[2] How to connect SystemVerilog with Python by Amiq Consulting
[3] Cocotb: a Python-based digital logic verification framework by Benn Roser, University of Pennsylvania.