THE INTERNATIONAL JOURNAL OF SCIENCE & TECHNOLEDGE

Co-verification of Registers Using UVM RAL

Shraddha Pal
Student, Department of Electronics and Communication Engineering,
NorthCap University, Gurgaon, Haryana, India
Neeraj Kr. Shukla
Associate Professor, Department of Electronics and Communication Engineering,
NorthCap University, Gurgaon, Haryana, India
Puneet Goel
Principal Consultant, CoVerify Systems Technology, Gurgaon, India

Abstract:
The technological development has led to rapid increase in the complexity of system design, whereas the system design gap is also increasing because of Moores law. Hardware dependent software (HdS) is addressing system design gap challenge by providing close interface between software and hardware. The co-verification of hardware and software thus becomes a critical task which is needed to be performed at system level for functional verification. Hardware registers are responsible for initial configuration of Design Under Test and hence their verification is vital for creation of error free test-bench environment. In System Verilog UVM Register Abstraction Layer (RAL) provides flexibility and reusability of register models but some run time overheads are incurred in System Verilog which limits the effective exploitation of UVM RAL. Multi core verification language Vlang has been used in this paper to efficiently employ UVM RAL without any overheads of speed and memory. AMBA AHB –Lite Master interface with I2C Core is used for co-verification of I2C register.

Keywords: *UVM, RAL, register model, interface, Vlang*

1. Introduction

Today's complex SoC has made Hardware-dependent Software an indispensable part of system design. Complex hardware needs to be controlled using software rather than manually managing it. Registers are part of DUT which are required to be configured for initial reset of DUT or for meeting any change in specification. Thus registers are required to be verified to ensure DUT are configured correctly. Functional verification of an embedded system or SoC requires the verification of both hardware and software, co-verification of hardware and software thus becomes indispensable. Building a truly verified computing system such as compiler or operating system requires both software and hardware verification. A major verification challenge is the multiple levels of abstraction it spans in a complex design. Hardware verification and software verification which are two disparate fields are needed to combine if a computing system needs to be truly verified [7]. Faster growth of hardware capacity than its productivity in design has created the hardware design gap. The reason is the Moores Law according to which the capacity of chips double in every eighteen months whereas increase in the productivity of hardware design is less than the growth of capacity of chip by more than twenty percent. Similarly, the software design productivity should double every ten months but it takes around twenty-four months to double the productivity of software design. This has created software design gap along with hardware design gap which cumulates into system design gap. The real challenge of this gap in system design is to tackle the close interaction and tight dependency between software and hardware domain. Hence a layer of complexity is created because of interfacing between software and hardware. This close interface requires Hardware –dependent Software (HdS) to be the main part of system design and makes co-verification indispensable for verification of large and complex system.

To meet the changing specification and constraint of time to market, re-configurability and flexibility becomes an important aspect of system design. A flexible system is one which can offer multiple feature and functions through different variations. For hardware design an effective way to introduce flexibility is by reusing and circulating its resources which can be achieved only through programming its hardware content using a software. Thus the concept of HdS becomes quite clear which requires hardware to be able enough so that it can be programmed using software. The configuration of hardware components is another important aspect in large and complex system. The configuration of a small sub-system can be done manually to some extent but for large system this process of configuration requires a software backup to reduce both time consumption and probability of error. Multiple layer of software is used to implement HdS. RAL is a Hardware Abstraction Layer (HAL) which gives way for abstract register models to configure and verify hardware registers.

Implementation of UVM RAL using Vlang is the main focus of this paper. The basic idea which makes Vlang highly desirable is the advantages it offers in comparison to System Verilog. Each register is treated as an individual class in System Verilog. Thus although register abstraction layer reduces the task of manual configuration but for a very large complex system a large overhead is earned in System Verilog. So the presence of register abstraction increases the memory footprint as well as compile time. Vlang on the other hand takes register as pointer and do not need instantiations, thus providing high speed and less memory consumption.

A survey has been on recent work done on register abstraction layer to make it more fruitful. In [1] a dynamic register database model is explored to exploit the UVM RAL efficiently. [2] provides solution to common modelling problem using active and passive with predictable and volatile modelling techniques. In [3] a methodology employing RAL layer is described to provide zero time, low maintenance and reusable register design and verification system. [4] highlight the importance of hardware software co-verification for a reliable system design. [5] Discusses the challenges of hardware software co-verification with multiple levels of abstraction. [6] Forms the basis of this paper by discussing the shortcomings of UVM RAl and different types of ways that can be employed to make UVM RAL safer, better and smarter.

The paper is organised as follows: section 1 gives the introduction of the need of co-verification and role of RAL. Section 2 gives the detailed explanation of UVM RAL. In section 3 next generation system level verification language Vlang is introduced and its effectiveness in implementation of UVM RAL is highlighted. Integration of interface UVC and RAL in test bench environment is explained in section 4. Simulation result is shown in section 5 and conclusion and future scope is discussed in section 6.

2. Register Abstraction Layer

In modern system design thousands of registers are present with quite a large number of register bits having multiple properties and functionality. Registers are critical for proper functioning and verification of device and they should be verified before tape-out. The abstraction layer provides accesses to DUT and also keeps a track of register content of DUT. UVM RAL is a UVM application package which can be used to automate the creation of high level, object oriented abstraction model of registers and memory in DUT. In any verification environment registers are used for initial setup of reset and read write tests of DUT. In absence of abstraction layer registers are required to be maintained manually.

The use of register layer makes the register abstraction and access of its contents independent of the bus protocol which is used to transfer data in and out of registers inside the design. Hierarchical model provided by RAL makes the reusability of test bench components very easy. The changes in initial configuration of registers or specifications can be easily communicated in the entire environment. RAL layer supports both front door and backdoor access. Front door access is the conventional verification technique where access is done using the same access ports as of design logic. The backdoor access does not use the bus interface rather it uses the HDL defined paths for direct communication with the device. Thus in zero simulation time the registers of device can be reconfigured using the backdoor access and verification can be started. One more advantage of backdoor access is that it can be used for verify if the access through front door are happening correctly. To achieve this the front door, write is verified using backdoor read.

UVM RAL is thus a standard modelling approach which provides a mean to control, check and cover DUT registers.

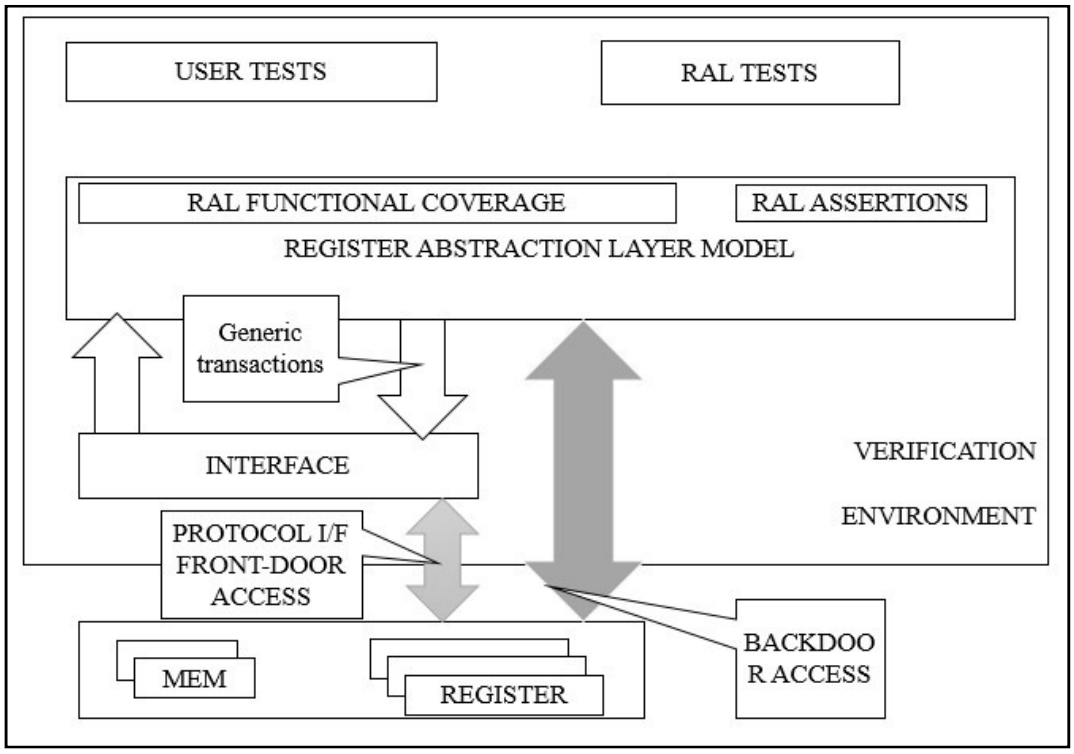

Figure 1: RAL Architecture

3. Vlang

An open-source programming language, Vlang is built using the D programming language as the base language. It meets the demands of System Level Verification by providing multi-core verification capability. Vlang addresses the challenges of System Level Verification along with keeping interoperability with RTL simulations. A multi-pronged approach is taken by Vlang for fast stimulus generation. Both Vlang and System Verilog compiles to native machine code but Vlang creates optimized code that run faster for most algorithms. This is so because System Verilog as a language has some overheads. Vlang completely avoids such overheads and creates executables that can potentially match the speed of any algorithm coded in C or C++. Secondly, Vlang enables multi-core test benches. In fact, Vlang is the only Verification Language that provides for a multicore capable port of UVM. One major issue with System Verilog is that the language does not come with a standard library. System Verilog also lacks data structure compatibilities and function overloading which are important to enable generic programming. It does not support multicore, with the coming of multicore processors, the lack of parallel execution ability of System Verilog may become the major limitation of this language.

System Verilog provides a backward compatibility with the Verilog but introduces monothicity in test-benches. The problem associated with monothilic test benches are:

- Gotchas are propagated
- Exposure to malpractices
- Reuse is limited
- Large compilation and elaboration time

System Verilog do not support any aspect of system programming. A language having the following features support the system level verification.

- It must be a modern system level programming language so as to support hardware software co-verification.
- Must be open source
- Should allow library level extension of language
- Provide ABI compatibility with C/C++.

Vlang built on the top of d offers all the advantages of software and provide highly efficient classes. The basic problem with UVM Register package is the huge source code for register layer which in itself comprises around nineteen percent of total UVM files and thirty-two percent of total number of lines of source code [white]. Ram size is another issue in UVM which makes it difficult to model 32-bit class. Everything is compiled in System Verilog so every code is included during compilation even if it is not required. Vlang has its origin at c and in C addresses are linked during compilation but in Verilog which looks modules on the basis of input output ports, linking process becomes much more complex. The linking and elaboration task of each and every register in UVM register layer is modelled as separate class. So the millions of registers present in system increase memory footprint exponentially during compilation. Thus the biggest flaw lies in compilation model of System Verilog which increase memory footprint and compile time. Vlang provides a solution because in C each file compiles separately and at the time of linking only memory addresses are required and no classes. The amount of source code for register package is also reduced by a large amount in Vlang UVM. Efficient memory utilization is made possible in Vlang. Between System Verilog and software memory mapping is direct whereas in Vlang software pointer is created which allows same memory location become accessible for Vlang and software. The open-source multicore solution provided by Vlang with added advantage of lesser source code base, efficient memory usage and high speed makes it a desirable system level varication language.

4. Testbench for RAL using UVM-Vlang

4.1. Interface UVC

Interface UVC is used for DUT and it either verify the implementation of design protocol logic or to program the DUT. Use of Vlang is done to create a verification IP to overcome the challenges faced by System Verilog. Interface UVC is created for AMBA AHB lite bus which acts as master for slave DUT I2C. Driver, monitor, sequencer are the various components which are encapsulated within a single agent. Multiple agents are possible within a single testbench environment. DUT is driven by Driver which pulls data items generated by a sequencer and drives it to DUT by sampling and driving DUT signals. Randomization of stimulus is executed by sequences. Sequencer generates the stimulus that is responsible for items provided to driver for execution. A sequence captures meaningful stream of transaction. Various functions performed by monitor includes collection of coverage information and transactions, performing checking, extracting events and optionally printing trace information [8].

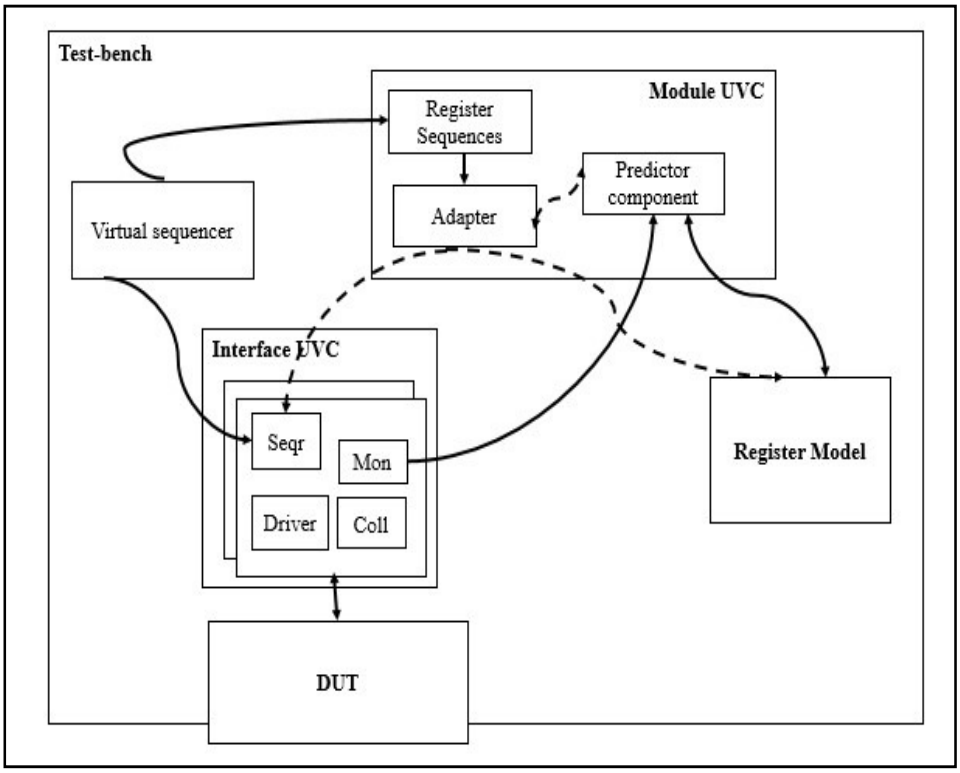

Figure 2: Integration of RAL and interface UVC [8]

4.2. Register model

A register model (or register abstraction layer) is a set of classes that model the memory mapped behaviour of registers and memories in the DUT in order to facilitate stimulus generation and functional checking. The UVM provides a set of base classes that can be extended to implement comprehensive register modelling capabilities.

As the number of registers are large in a design and many minor details are also associated with them so these registers are generated automatically. The model thus generated contains register block which assembles many registers and memories. Every register contains varying number of individual fields, each with a functionality and hence a specific duty of hardware register is defined by applying different values to fields of register. Modelling the register and its field is achieved by extension of the base class uvm_reg and uvm_reg_field. Basically the register models are like a mirror image for the real registers inside the DUT.

4.3. Register Layer

Presence of register model, register abstraction layer and interface UVC forms a Register Layer inside testbench. Function of adapter and predictor components are explained below.

4.3.1. Adapter

An adapter is created by extension of base class uvm_reg_adapter and this base class contains two functions bus2reg () and reg2bus (). The bus specific adapter here performs translation of register operation to AHB transaction using reg2bus () and the other function bus2reg () translates AHB transaction to registers operations. The access of hardware register is done by using either front door access or backdoor access which are also responsible for reconfiguration of register model. Codes for converting generic transaction to bus specific and vie-versa is shown below.

```
class reg_to_ahb_adapter: uvm_reg_adapter reg2bus(const ref uvm_reg_bus_op rw) {
     ahb_transfer transfer =
ahb_transfer.type_id.create("ahb_transfer");  transfer.hwrite    = (rw.kind == UVM_READ) ? 0 : 1;
transfer.haddr = cast(ubyte) rw.addr;   transfer.htrans = trans_e.NONSEQ; transfer.hwdata = cast(ubyte)
rw.data;   return transfer;
  }
  override void bus2reg(uvm_sequence_item bus_item, ref uvm_reg_bus_op rw) {
    ahb_transfer transfer = cast(ahb_transfer) bus_item;     if (transfer is null) {       uvm_root_fatal("NOT_REG_TYPE",
"Incorrect bus item type. Expecting ahb_transfer");       return;     }
    rw.kind = (transfer.hwrite == 0) ? UVM_READ : UVM_WRITE;     rw.addr = transfer.haddr;
    if(rw.kind == UVM_WRITE) rw.data = transfer.hwdata;       else rw.data = cast(Bit!8) transfer.hrdata;
rw.status = UVM_IS_OK;
```

4.3.2 Predictor

Predictor is created by extending the base class uvm_reg_predictor is used to make changes in register model after every access
Transaction class are created to pass grouped data items to DUT. The DUT is driven by bus functional model which is a driver. Driver receives data item and then data items are driven by driving the DUT signals. Sequencer in the testbench forms a pre-structured stimulus pattern and monitor performs coverage by collecting transaction. This bus UVC is then connected to memory mapped register model using adapter and predictor. The entire testbench environment thus provide a reusable and flexible verification component and register model.

5. Simulations and Results

Configuration and verification of DUT registers are achieved by creating an abstract register model which is then configured by writing user defined sequences. The code showing the configuration sequences created in register layer is shown below.

```
class ahb_reg_seq: uvm_sequence!ahb_transfer
{  reg_block model;    mixin uvm_object_utils;
------------//------------------------
   model.PRERlo_h.write(status, 0x81, UVM_DEFAULT_PATH, null, this);
   if(status == UVM_NOT_OK) {
     writeln(" cannot write into register PRERlo_h");
   }
   model.PRERlo_h.read(status, data, UVM_DEFAULT_PATH, null, this);     if(status == UVM_NOT_OK) {
     writeln(" cannot read from register PRERlo_h");
   }
   writefln("read data is %h", data);
------------//------------------------
uvm_info("Reg_seq", "Sequence completed", UVM_LOW);
  }
};
```

Using the above code format of Vlang lower and upper bits of prer register and ctr register are configured by using the write function in register layer. The verification is performed by using the read operation for DUT registers and displaying the values of registers.

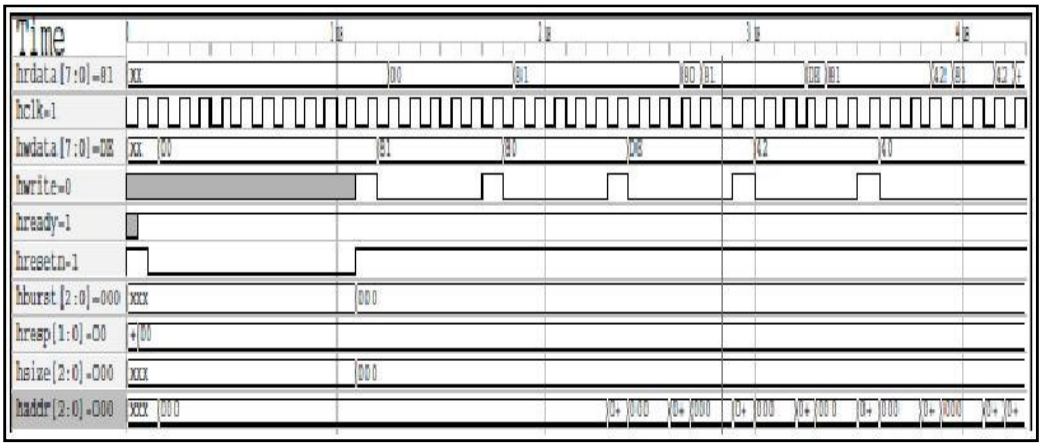

Figure 3: Waveforms for read and write transactions

6. Conclusion and Future Scope

Co-verification of registers are done by configuring the registers using the UVM RAL and verifying the resultant waveforms of different registers. Use of Vlang for implementation of UVM RAL has made UVM RAL much more desirable in comparison to its implementation in System Verilog. As in highly complex system only a selected set of registers are used and configured without RAL to avoid the overhead obtained in System Verilog. Whereas Vlang has made it possible to exploit UVM RAL features without any compromise with speed and memory.

The present work is implemented for front door access only. The future scope of this work is to implement the backdoor access for register abstraction layer and thus further reduce the simulation and verification time.

7. References

i. Sailaja Akkem, “UVM RAL: Registers on demand Elimination of the Unnecessary*”, in proceedings of Design and Verification conference (DVCon), 2015.

ii. M. Litterick, M. Harnisch, “Advanced UVM Register Modeling – There’s More Than One Way to Skin a Reg,”, in proceedings of Design and Verification conference (DVCon) 2014.

iii. Amruth Kumar J, Arun Kumar M, Kundu Priyabrata, "Functional Verification of Register using UVM RAL Methodology", IJLTEST – Vol. I, Issue 5 (May-June 2014), pp. 1-5 [4] Milan Purohit, Shantanu Bhattacharyya, Puneet Goel, "Software Driven Hardware
iv. Verification – A System Verilog DPI/UVM based Approach", in proceedings of Design and Verification conference (DVCon), India 2015, Bangalore
v. Panagiotis Manolios, "The Challenge of Hardware-Software Co-Verification", Published in Verified Software: Theories, Tools, Experiments, pp. 438-447.
vi. Rich Edelman and Bhushan Safi, "Beyond UVM Registers, Better, Faster and Smarter", in proceedings of Design and Verification conference (DVCon) 2015.
vii. Wolfgang Ecker, Wolfgang Muller, Rainer Domer, "Hardware-dependent Software Principles and Practice", Springer, 2009.
viii. Kathleen A Meade and Sharon Rosenberg, "A Practical Guide to Adopting the Universal Verification Methodology (UVM)", Second Edition