2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

In pursuit of Faster Register Abstract Layer (RAL) Model

Anmol Rana, R&D Engineer, anmol@agnisys.com
Bhagwan Jha, R&D Engineer, bhagwan@agnisys.com
Harjeet Singh Sanga, R&D Engineer, harjeet@agnisys.com
Agnisys Technology Pvt. Ltd. Noida, India

Abstract - **RAL is used to model the registers and memories present in the design. UVM comes with register package which is used to model these registers and memories. With its numerous advantages, there are also some disadvantages. The UVM RAL model is good for small testbenches but when one moves to large system level testbench which contains thousands of registers, it impacts the performance and adds significant load to the simulator. To overcome these performance issues, we have developed and benchmarked several alternative RAL models. The underlying approach in the first two models (SV RAL with lookup and C based RAL with lookup) is to optimize the number of handles to register or field classes and to store register information in associative-array based lookup tables. There is also a third model which uses the register and field classes written in C++ instead of "uvm_reg" and "uvm_reg_field" classes.**

I. Introduction

The speed of the RAL model has become important as more and more SoCs are built using existing IPs – which in turn have thousands of programmable registers. There is some literature available to guide the verification engineer, but it requires painstaking research that is difficult when the project deadlines are looming large. In this paper we give an overview of the landscape based on our research. From simple things as removing factory registration for SV UVM, to creating lookup based models that reduce the number of handles which degrades the simulators during simulation and impacts the performance. We also explore C++ based RAL models and their shortcomings and advantages. Three of these approaches are discussed below.

A. SV RAL with lookup

This model is based on the approach of having static databases for storing registers information instead of having separate class handle for storing each register information, which is generally followed in traditional UVM RAL model. This model will have two classes, one is register class and another being the top-level class. The top-level class will behave like a "uvm_reg_block" class. For 10K registers, this model will have only one register class handle which will handle all the read/write operation for the 10K registers as compared to the UVM RAL which will have 10K register class handles for read/write operation.

B. C based RAL with lookup

This model is also based on the approach of having databases for storing registers information. In this approach the databases are created in C. With the help of DPI-C these databases are accessed whenever a read/write transaction is performed. Just like the above model, this will also have two classes one is top level and another being the register class which will handle the read/write operations for all registers.

C. C++ based RAL

This model is based on the traditional approach of having one register class handle for storing each register information. In UVM RAL model the register and field classes are extended from uvm_reg and uvm_reg_field respectively but in C++ based RAL the register and field classes will be extended from "agni_reg.cpp" and "agni_reg_field.cpp" class, which are written in C++. The dynamic methods of agni_reg.cpp and agni_reg_field.cpp are wrapped in static methods that uses the object handle, passed from System Verilog. Wrapper class "agni_reg.svh" and "agni_reg_field.svh" are used to hide the object of "agni_reg.cpp" and "agni_reg_field.cpp" class respectively.

With some changes these register/field class can be used inside the UVM RAL model. Inside the uvm_reg_block the register and field classes can be extended from "agni_reg.svh" and "agni_reg_field.svh". User can also create its own top class and use it instead of the uvm_reg_block class. Wrapper classes are written in System Verilog, so the user has the ability to add constraints and coverage model.

II. SV RAL WITH LOOOKUP

This model is based on the idea to reduce the number of handles for register read/write operation. This model uses a single class handle to perform read/write operations for all the registers present in address map. All the register information are stored in associative arrays which are manipulated while accessing registers.

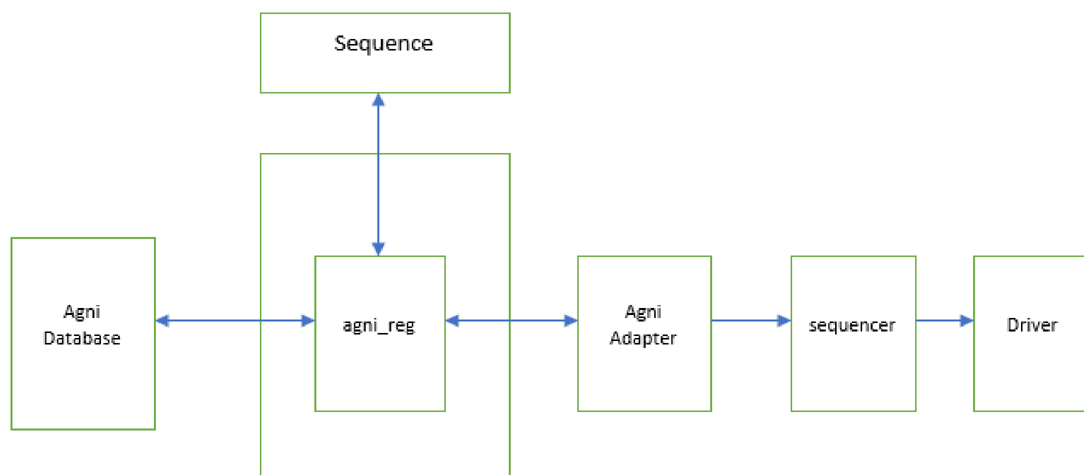

Figure 1 Overview of SV RAL

A. Agni Database

Agni Database class is the heart and soul of the SV RAL model. This class contains the static databases which have stored the register information. The databases are in the form of associative arrays name of the register with keys as. These databases will be accessed by the agni_reg class methods for read/write operations.

B. Agni Register class and Agni Adapter class

agni_reg class will provide the means for register read/write operations. Only one instance of this class will be created which will be used for accessing all registers present in address map. agni_reg class internally uses the agni_adapter class methods for converting the register information into bus specific transaction and vice versa. User has to override the agni_wr and agni_rd methods in the derived class extended from agni_adapter for converting the register information into bus specific transaction.

Methods provided by the Agni Register class: -

Methods	Example	DUT	Database
Write	Model.regx.write(32'hffffffff,"reg0")	This method will write the values to dut	Writes value to the database
Read	Model.regx.read(rd_data,"reg0",1/0)	This method will read the value from the DUT	If third argument is 0 :- update the model 1 :- compare the value with database and update the database with generating the uvm_error
Get	Model.regx.get("reg0")	No effect	Return the database value

C. Pros & Cons

1). As a single handle is used to deal with all register transactions. This model can solve the problem of moving the RAL to SOC level. The write method can be enhanced for supporting this flexibility. As we know that for block level the address of register will be the register offset itself and when we want to access this register from the SOC level it become register offset + block offset. The write method can be enhanced like write (data,"reg0"," block0",1/0) or write (data," block0.reg",1/0), the last argument will be used for changing register address like if it is 1 then address used from SOC level point of view else from block level. Block name string can be used to access the block offset which will be stored in the form of associative arrays.

2). Signature for writing/reading will be different from traditional RAL model.

3). Field access is not supported for this model. This can be done by using the associative arrays, but this can make the RAL troublesome for having so many associative arrays and may result in increase in memory. User can deal with this problem by using field masks.

III. C based RAL with lookup

This model is also based on the idea to use one handle for accessing all the register of address map. All the register information stored in associative arrays which are defined in C language. This information is accessed through DPI-C for register transactions. The idea was to move all associative lookup calculations to C side rather than doing it in System Verilog in order to save the calculation timings as we know C language compute faster than the System Verilog.

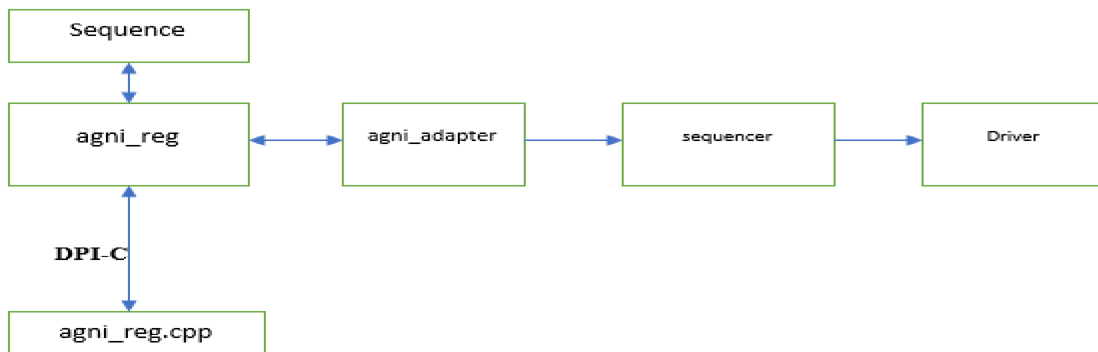

Figure 2 Overview of C based RAL with lookup

A. Agni Register class (C++)

agni_reg.cpp class stores the information of registers in associative arrays with register name as the key of the associative arrays. This information is accessed for register transactions through DPI-C.

Figure 3 shows the static methods that act as wrapper functions for the dynamic functions of agni_reg.cpp and uses the object handle passed from System Verilog side imported functions so that the System Verilog classes will able to access the agni_reg.cpp class functions.

```
extern "C" void* reg_new() {
        return new agni_register();
 }
 extern "C"  int  address(void* inst,const char* name ){
      agni_register *reg11 = (agni_register *) inst;
     return reg11->get_address(name);
 }
 extern "C"  void wrdata(void* inst,int wr_data,const char* name ){
       agni_register *reg11 = (agni_register *) inst;
       reg11->write_data(wr_data,name);
 }
 extern "C"   int rddata(void* inst,const char* name){
       agni_register *reg11 = (agni_register *) inst;
       return   reg11->get_data(name);
}
```

Figure 3: - DPI-C Functions for sharing information b/w C and SV

B. Agni Register Wrapper Class (SV)

Agni Register Wrapper class is a System Verilog class which provide methods for register transaction. This class act as wrapper class for the agni_reg.cpp so that the user gets clean interface just like traditional ral model. Just like the SV model, there will be only one instance of this class which will be used for accessing all register present in address map. The methods used for register manipulation will be same as that of the SV RAL model. The only difference is that they internally use the DPI-C functions for accessing register information.

Methods for read/write transactions: -

Methods	Example	DUT	Database
Write	Model.regx.write(32’hffffffff,“reg0”)	This method will write the values to dut	Writes value to the database
Read	Model.regx.read(rd_data,“reg0”,1/0)	This method will read the value from the DUT	If third argument is 0 :- update the model 1 :- compare the value with database and update the database with generating the uvm_error
Get	Model.regx.get(“reg0”)	No effect	Return the database value

C. Pros & Cons

Pros & Cons will be same for as in SV RAL model. As in this model only the lookup tables are moved to C side rest remains the same.

IV. C++ based RAL

The model uses the traditional way to have one class handle per register for storing information and for register's read/write transactions. The approach behind this model is to have the register information stored in classes written in C++ language. So, for every register there will be an instance of this class which store the address values, default value and the current values which are updated through tests.

```
class block1_reg0 extends agni_register;
        agni_reg_field f0;
        agni_reg_field f1;
    function new(string name = "", int def, int addr);
          super.new(name, def, addr);
          f0 = new(this, "f0", 16, 0);
          f1 = new(this, "f1", 16, 16);
    endfunction
endclass
class block1_block;
      block1_reg0 reg0;
  function new();
       reg0 = new("reg0", 0, 0);
  endfunction
endclass
```

Figure 4: - Basic example of C++ RAL model

A. Agni Register Class (C++)

Agni Register class is C++ class which will store the information of registers in variables rather than storing them in associative arrays. This register information is accessed through the DPI-C. Figure 5 shows the static DPI-C methods used by the agni register and agni register field class for accessing the register information.

```
extern "C" void* reg_new(int def,int addr) {
       return new agni_register(def, addr);
}
extern "C" void set_reg(void* inst, const int def,const int addr ){
      agni_register *reg11 = (agni_register *) inst;
      reg11->set(def,addr);
}
extern "C"  void  write_reg(void* inst,const int data){
       agni_register *reg11 = (agni_register *) inst;
       reg11->write(data);
}
extern "C" int get_regdata(void* inst){
      agni_register *reg11 = (agni_register *) inst;
      return reg11->get_data();
}
extern "C" int  get_regaddr(void* inst) {
       agni_register *reg11 = (agni_register *) inst;
       return reg11 ->get_address();
}
```

```
extern "C" int get_mask(void* inst,const int bits,const int lsb){
          agni_register *reg11 = (agni_register *) inst;
          return reg11 ->mask(bits,lsb);
}
extern "C" int data_fld(void* inst,const int fld_data,const int maskbits,const int field_lsb){
         agni_register *reg11 = (agni_register *) inst;
         return reg11 -> field_data(fld_data,maskbits,field_lsb);
}

 extern "C" int fld_read(void* inst,const int maskbits){
                    agni_register *reg11 = (agni_register *) inst;
                    return reg11 -> get_fielddata(maskbits);
}
```

Figure 5 Static DPI-C functions

B. Agni Register Class (SV)

This class is just a wrapper class for the C++ class so that the user will have clean interface and does not have to bother about the things happening in background. As the wrapper classes are written system Verilog so user can have the advantage of using System Verilog things like constraints or coverage models. This class provides the basic read/write method which in turn uses the imported methods for information transfer.

C. Agni Register Field Class (SV)

This class will provide the methods for field manipulation just like the uvm_reg_field. This class will use its parent class methods which in turn uses the DPI-C methods for accessing register information. A field write uses the read modify write algorithm for updating the fields.

D. Pros & Cons

1). All the necessary functionality of UVM RAL can be ported into this RAL model, as in this model we have moved all the computation work to C side and uses DPI-C functions to access them. So, it may be possible to port certain set of functionalities that are necessary from test point of view without changing the traditional UVM RAL signature. Porting whole model will be troublesome.

2). As discussed in above sections the register will be extended from agni_register class rather the uvm_reg class. A mix RAL model can be created with the normal register extended from agni_register class and other special registers from uvm_reg. But this will result in loss of certain privileges like get_registers() functions from uvm_reg_map class which provide handles of all registers.

V. Performance Comparison

We have benchmarked traditional UVM RAL model with all of these alternative models viz. SV RAL with lookup table, C based RAL with lookup and C++ based RAL. To compare performance of these alternative RAL models with UVM RAL model, we gave simple read/write operations using same sequences under same UVM environment with standard simulators. Similar tests are run on 100, 1K, 3K,5K and 10K, register models and their data is compared for overall time taken (compilation + elaboration and the run time). We have used three different simulators and for this paper purpose they will be named as SimX, SimY and SimZ.

Graph Legend

UVM Performance curve represent UVM RAL model.
SV RAL Performance curve represent SV RAL model with lookup table.
C RAL Performance curve represent C++ based RAL
CT RAL Performance curve represent RAL with lookup table in C++
X-axis: - Register Count
Y-axis: - Number of registers read/wrote in 1 sec.

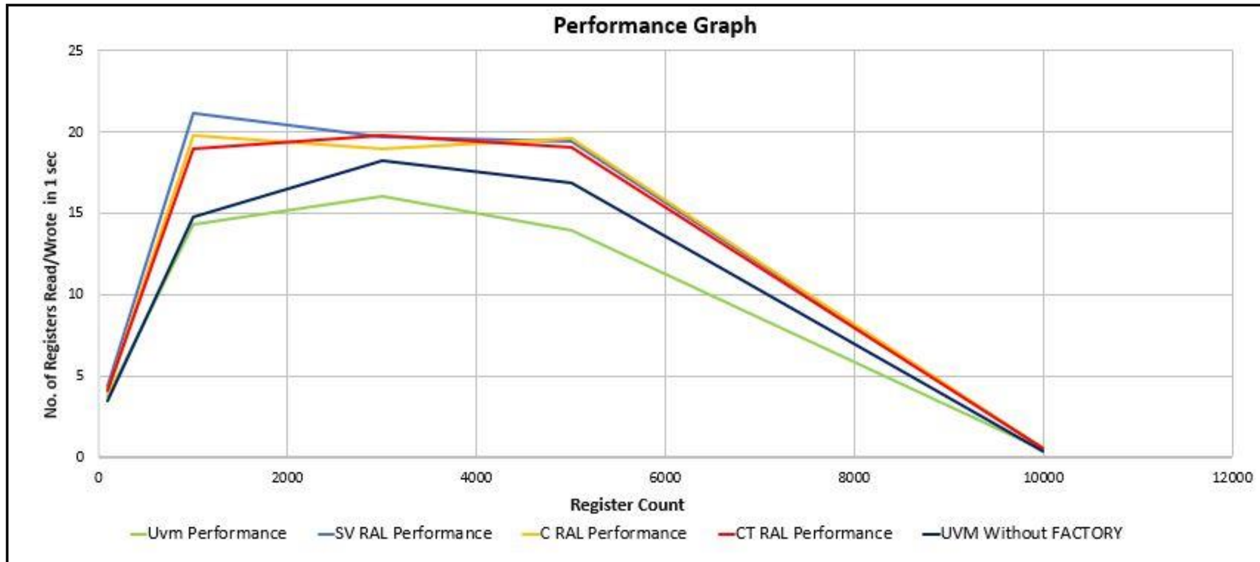

Figure 6 SimX

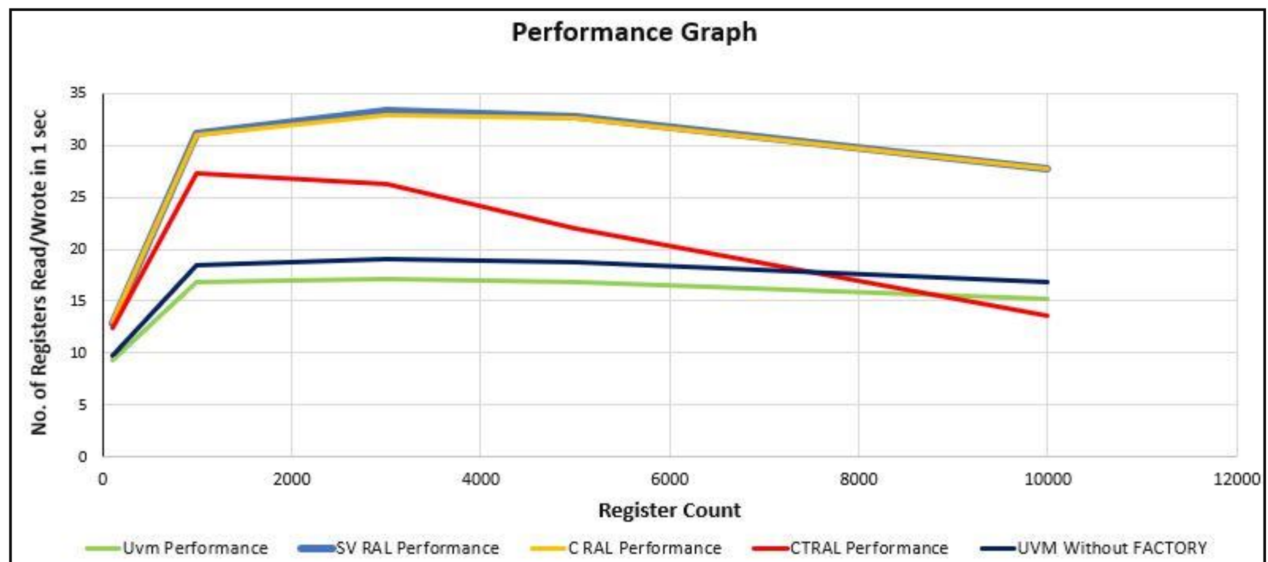

Figure 7 SimY

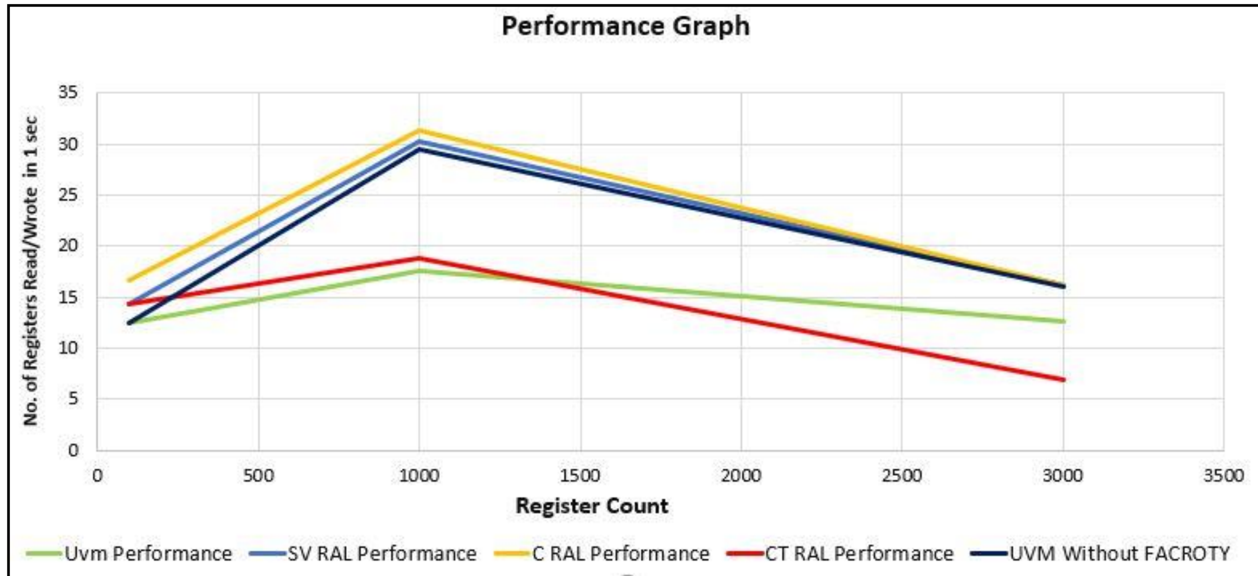

Figure 8 SimZ

Same test and same environment run on different simulators. From above graph it is clear that all the defined alternate models have higher performance than the traditional UVM RAL model till certain count of registers. It is important to note that after certain count of registers the performance of models started to decrease.

For SimX: - the performance difference is not so huge but still alternate models are faster than the UVM RAL to certain extent.
For SimY: - the performance difference is double for C++ RAL and SV RAL, but the CT RAL performance comes out to be worse than the UVM RAL.
For SimZ: - the performance difference is huge till certain count of registers and eventually all the models will have same performance at some point at a certain register count. CT RAL performance degraded most as compared to others.

VI. SUMMARY

This paper introduces the new approaches for register layer. These models are developed in order to overcome the performance issues of the traditional UVM RAL model. We are somewhat successful in increasing the overall performance, but the performance degrades when register's count crosses certain point. We have used different simulators for our experiment, but these simulators have different performance curves, so reader will need to conduct their own performance checks.

We have concluded the results based on above performance graphs.

1. Better result can be achieved for maximum performance by taking register count below 5000.
2. There is no need to explore on C language because after 5000 registers the results will be same as in SystemVerilog(SV RAL).

As the paper title suggest we are still in pursuit to increase the performance of register layer.

REFERENCES

[1] Simpler Register Model https://github.com/Juniper/simple_reg_model
[2] Register on Demand by Sailaja Akkem https://dvcon-india.org/sites/dvcon-india.org/files/archive/2015/proceedings/24_UVM_RAL_Registers_On_Demand_paper.pdf
[3] Litterick http://www.verilab.com/files/litterick_register_final_1.pdf
[4] SystemVerilog UVM http://accellera.org/downloads/standards/uvm
[5] cluelogic.com http://cluelogic.com/2012/10/uvm-tutorial-for-candy-lovers-register-abstraction/