2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

UVM RAL: Registers on demand

Elimination of the Unnecessary*

Sailaja Akkem

MicroSemi Corporation Pvt. Ltd.

Hyderabad, India

sailaja.akkem@microsemi.com

***Abstract*— This paper puts forward a novel approach of deploying UVM Register data base in verification environment. The supreme edge in utilizing this approach is that it improves the performance of the test on the whole (which is from compilation to simulation). This approach is perused, with examples, from idea to adoption in this paper.**

Keywords—UVM RAL, Register model, dynamic database creation.

I. INTRODUCTION

UVM-RAL is a powerful abstraction layer that is imbued with features which are relevant for modelling the registers in all kinds of environment while excellently aiding vertical re-usability: The register classes and the structure itself can be adopted into system from blocks.

Nonetheless, deployment of RAL in conventional approach adds a burden on the performance (which tends to deteriorate further when the approach is extended from block to chip level): Typically, it is not uncommon for today's complex SOC's to have tens of thousands of registers and fields. However, in a test, we may be needing only 10-20% of those registers depending on the feature it is required to cover. By using UVM RAL in the conventional method, we tend to create handles for all these register classes, we then compile them, load them and simulate them in every test. Simulator is laden with classes' handles and the compiled classes' prototypes, which, in most cases, are superfluous to the requirement which is being verified by that test.

This paper puts forward a dynamic approach that creates handles automatically only upon the requirement (making the database dynamic from static). This handle creation also facilitates any overrides if requested. This paper also proposes few guidelines which we have implemented in our verification environment so as to improve simulation performance, while registers' accesses were made from both test bench and test case.

The approach is examined with examples highlighting:

- Benefits of creating dynamic register data base: Dynamic creation of classes is the cardinal conviction around which current verification methodologies have evolved. Creating register data base statically does not provide this flexibility. The benefits of creating register class handles only when requested will be explored in this paper.
- Performance improvement: Improvement in performance using the dynamic register data base approach is underscored in this paper.

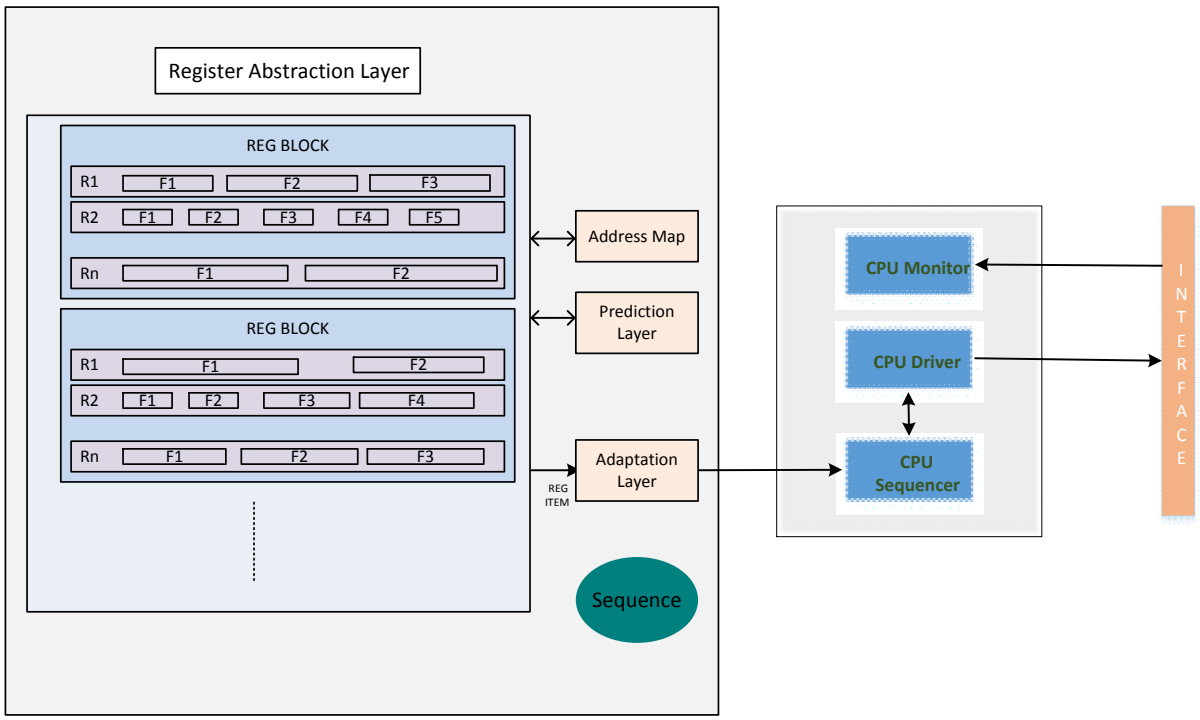
This paper explores the approach in this layout:

- First, it explores the versatility of standard base classes provided by UVM Register Adaptation Layer and also covers the documented approach of creating register model.
- Second, it highlights how the dynamic register restraints were unraveled.
- Subsequently, it explores with examples on how the custom core service was leveraged to honor register requests.
- Later it mentions metrics of performance improvement.

II. UVM REGISTER ADAPTATION LAYER

A. *UVM RAL*

The adaptation layer provides a convenient way of accessing registers in the DUT. In addition to this, the adaptation layer also keeps a track of register content of the DUT.



Accessing registers in the DUT is facilitated by adapters, sequence, register model, with interface specific drivers and monitors. Essential elements in implementing the functionality of a register model are:

Register Model: This model is the one which replicates the register information present in the DUT.

Adapter: The utilities provided by the RAL generate generic transactions, which contain relevant information essential for driving the bus. However, this transaction is non-specific as to any bus related additional information that may be needed by the bus driver. For example, the address formats may be different from interface (which may use 16-bit wide address bus) to interface (which may use 32-bit wide address bus) or additional information like channel ID may be needed to be driven on the bus based on the address of the register. Such conversion from generic transactions to bus/interface-specific transaction is performed by the adapter. Similarly, the response in the bus transactions is to be converted back to generic register transactions to be understood by the register model. Plugging in a bus-specific adapter and overriding bus2reg and reg2bus methods can achieve this.

Prediction elements: Any register access done through register model is predicted and updated inherently by the base classes. Prediction of the register values in the model can also be done explicitly using an external predictor which also takes care of interface bus transactions not occurring through register model.

Backdoor access: Facilitates direct access to signals of registers inside the DUT. uvm_reg_backdoor base class can be extended to perform project specific backdoor accesses as well.

B. *UVM RAL STANDARD BASE CLASSES*

The base classes provided by the UVM RAL are highly protean in that by creating a layer above these base classes or by instantiating these base classes directly, registers of highly complex designs can be modelled. This section reviews the utilities provided by base classes (especially the ones provided by register field, register and register blocks):

a) Creation of register model: Utilities like configure, Xinit_address_maps, add_map, set_offset etc., are available which allow variable register addressing formats to be handled i.e., using these utilities, registers positioned at different addresses in a register map can be modelled. Similarly, utilities like get_reg/field_by_name, get_parent, get_block etc., allow the user to reach to any point in the hierarchy of a register. This way, VIPs can be modelled to be irrelevant of the hierarchy changes of registers or field across projects.

b) Handling register processes: Utilities like update, mirror, write/read etc., are provided with adequate flexibility (in the form of virtual functions and call backs) to be used for different chip requirements. Chip specific classes can be plugged into the environment without disturbing the re-usable framework.

c) Mimicking DUT's behaviour: Utilities like XupdateX updates the field values effectively mimicking that in DUT. These base utilities are quite apposite in mimicking the DUT's behaviour in most of the field access types: "rw", "ro", "wo" etc., Any chip specific field access types can be still hooked into the register structure. Utilities are provided to reconstruct the value of a register based on individually configurable variables like msb, lsb positions of the field.

Such generic utilities, which could be just plugged into any verification environment with minimal or no changes (especially true for blocks which do not model any complex register functionality), definitely outweighed custom utilities we had in our environment prior to adoption of UVM RAL. Hence, it was contemplated to deploy these base classes efficiently to mimic DUT's behavior keeping the performance of the simulation from deteriorating and making the register model on the whole to be re-usable across projects. However, using statically created register database brings down performance of the simulation. This performance deteriorates further as the size of register database increases (i.e., from block to sub-block to chip). Hence, a dynamic data base approach was implemented in our environment(s), without compromising the features that UVM RAL standard base classes provided and as far as the test case writer is concerned, the register data base should look like a fully built data base similar to the conventional UVM register model.

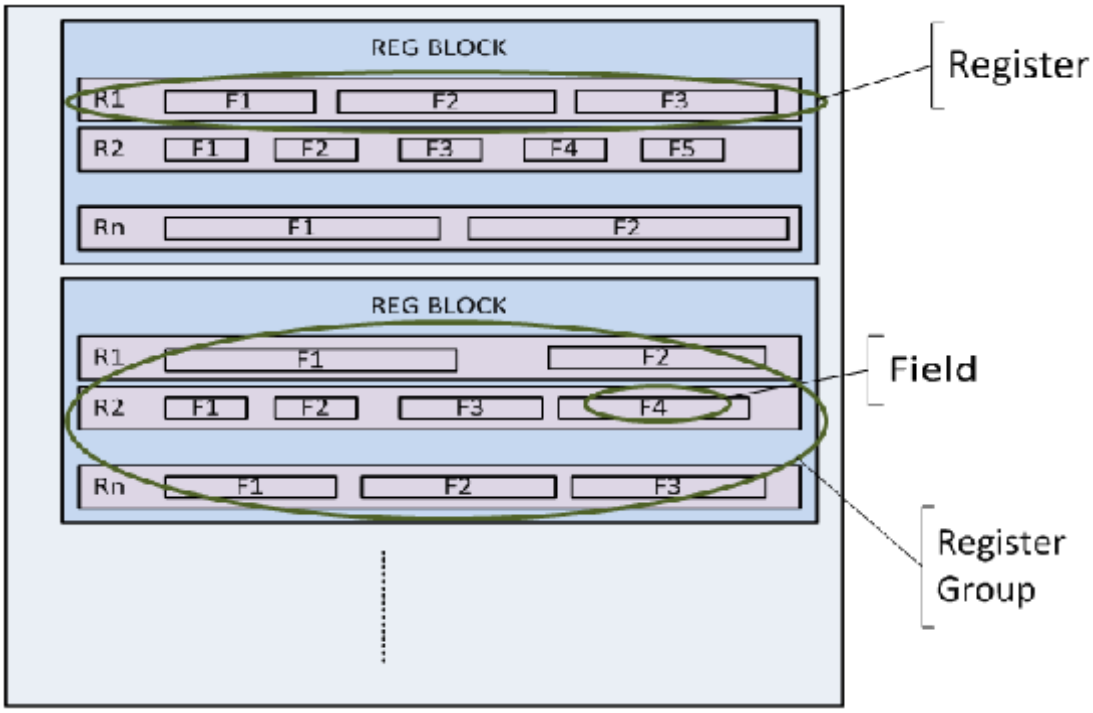
Decrease in performance when deploying conventional model occurs due to two overheads: One is the overhead due to static model and another is the overhead of expendable handles during the simulation. The approach tried in our environment was able to overcome these two overheads. Further sections provide more insight on how this was attempted at.

III. FRAMEWORK

This section provides high level overview of how the framework of the dynamic register model approach, which was implemented, differed from the framework of the conventional register database and highlights how this enabled to overcome performance reduction due to static model building. Simultaneously, this section also projects code examples on how dynamic register database approach was implemented in our verification environment.

A. CONVENTIONAL UVM REG MODEL

Conventionally built Register Model of a chip would look like this:



Register block is a collection of registers or sub-register blocks. Register, in turn, is a group of fields. In the conventional register model creation, a register block class extends uvm_reg_block base class, a register class extends uvm_reg base class, and a field class extends uvm_reg_field base class. Here, a class prototype is created for every register block and register. If a chip contains 5000 registers, 5000 class prototypes are created with registers, and similar number of class prototypes are created for register blocks. Practically, all the class proto types for registers would look like:

```
class enable_cfg extends CpuRegister;

  `uvm_object_utils(enable_cfg)

  rand CpuRegField tx_sw_rst;
  rand CpuRegField tx_ena;
  rand CpuRegField rx_ena;
  rand CpuRegField rx_sw_rst;

  function new(string name="enable_cfg");
    super.new(name,32,build_coverage(UVM_NO_COVERAGE));
  endfunction

  virtual function build();

    this.tx_sw_rst = CpuRegField::type_id::create("tx_sw_rst");
    tx_sw_rst.configure(.parent(this),
                        .size(1),
                        .lsb_pos(0),
                        .access("RW"),
                        .volatile(0),
                        .reset(0),
                        .has_reset(1),
                        .is_rand(1),
                        .individually_accessible(0));
```

Similarly class prototype for a register block would look like:

```
class fc_buffer extends CpuBlockRegs;

  `uvm_object_utils(fc_buffer)

  /////Registers///
  rand enable_cfg enable_cfg_inst;
  rand mode_cfg mode_cfg_inst;
  uvm_reg_map fc_buffer_MAP;

  virtual function void build();

    enable_cfg_inst = enable_cfg::type_id::create("enable_cfg_inst");
    enable_cfg_inst.configure(this,null,"");
    enable_cfg_inst.build();

    mode_cfg_inst = mode_cfg::type_id::create("mode_cfg_inst");
    mode_cfg_inst.configure(this,null,"");
    mode_cfg_inst.build();

    fc_buffer_MAP = create_map("fc_buffer_map",0,4,UVM_LITTLE_ENDIAN);
    this.default_map = fc_buffer_MAP;
    fc_buffer_MAP.add_reg(enable_cfg_inst,0);
    fc_buffer_MAP.add_reg(mode_cfg_inst,1);
  endfunction:build

endclass: fc_buffer
```

The model is hierarchically built by creating register blocks first, and then instantiating registers in those blocks, with those registers further instantiating fields in them. Hence, the class prototypes contain build functions to instantiate their respective child classes, and also configure functions to provide necessary information to a field: parent of the field, size of the field, position of the field in the register, access type, volatility, reset related information and most importantly if the field is individually accessible. Without creation of handles, and without configuration through those created handles, the register model is incomplete. In addition to this, optionally coverage related information is also present in the register model.

To accomplish these functionalities (i.e., building and configuring the classes), a register specification document is looked up (parsed): which is typically in the form of a markup language file, which contains all the information necessary to build the model. A snapshot of a markup language file used in our projects is:

```
 <reggrp name="CONFIG" repl_cnt="1" repl_width="38" base_addr="0">
   <reggrp_short_dscr>MACsec configuration registers</reggrp_short_dscr>
   <reggrp_public_text/>
   <reggrp_private_text/>
   <reg name="ENABLE_MODE_CFG" repl_cnt="1" repl_width="1" addr="0">
     <reg_short_dscr>MACsec enable register</reg_short_dscr>
     <reg_public_text/>
     <reg_private_text/>
     <field name="CLK_ENA" type="rw" default="0" width="1" pos="0">
       <field_public_text>Enables clock to MACsec receive path (Ingress path).</field_public_text>
       <field_private_text/>
       <field_enc>0: Block don't receive clock.&#13;
1: Block receive clock.</field_enc>
     </field>
```

All such class prototypes are then compiled into individual packages. Effectively, the overhead added by this compiled package will be very significant in a simulation, which brings down its performance. Please refer to the section on performance metrics.

We ruminated in our project that, if somehow, this prototyping of every register class and every register block could be circumvented, there will be huge savings in terms of compiled database and this would proportionately affect the simulation.

We were able to accomplish this in a dynamic register database model.

B. DYNAMIC REGISTER MODEL

Without class prototyping, essential information for building the model will be vanished (i.e., the information in the form of attributes parsed from markup language file will be lost) and without this information, bus transactions cannot be formed: because position of field in register, address of register, and any other bus specific information is given by these attributes. However, if the classes are prototyped to reflect the structure of the

markup language file, simulation performance is degraded due to additional overhead added by this compiled data base. To avoid this, a mid-way has been adopted in the dynamic register model.

In the dynamic register model approach, two levels of parsing has been adopted. First level of parsing of markup language file (containing registers specifications) is done through a custom script. And the relevant attributes of the markup language are extracted and stored into a text file. By the word relevant here, it is meant that those attributes which were otherwise existing in the conventional register model class prototypes. In all our projects, attributes that were needed to perform a bus transaction were: field position and its access type, register address, register group address, any device identifiers, reset value of the field and hdl paths for the register (for backdoor access). These were essentially similar attributes as passed to the configure function called on each field class's handle. Hence the text file which we needed for our environments looked like (shows only part of the file):

```
21 0  enable_mode_cfg    clk_ena        0 0   rw 0
21 0  enable_mode_cfg    sw_rst         4 4   rw 0
21 0  enable_mode_cfg    block_ena      8 8   rw 0
21 0  enable_mode_cfg    bypass_ena     12 12 rw 0
21 1  vlan_etype_cfg:0   vlan_etype     15 0  rw 0x88A8
21 2  vlan_etype_cfg:1   vlan_etype     15 0  rw 0x88A8
```

Every line in this text file contains information necessary to build the hierarchy of one field.[1]

Once this text file is available, compilation of the verification environment continues without creating register class prototypes. The compiled verification database will be small in size. In this dynamic register model approach, in the simulation, this text file is parsed once by system verilog file parser using a simple function, when any register access is requested to the custom core service from anywhere in the verification environment or from the test. This function stores the parsed information in associative arrays of structures, indexed by names of registers, register blocks and fields in a hierarchy. A snapshot of this structure elements:

```
regmap_entry_s              rm_entries[string][string][string]; /**< Indexed by block,reg,field( there will b
                                                                    per regmap.out entry). */
```

While individual structure element would look like:

```
typedef struct {
   bit   [TGT_ID_WID   - 1 : 0]   tgt_id;
   bit   [REG_ADDR_WID - 1 : 0]   reg_addr;
   bit   [BIT_POS_WID  - 1 : 0]   msb_pos;
   bit   [BIT_POS_WID  - 1 : 0]   lsb_pos;
   bit   [REG_DATA_WID - 1 : 0]   default_val;
   access_mode_e                  acc_mode;
   string                         reg_name;
   string                         fld_name;
   string                         access_mode_s;
   bit   [REG_DATA_WID - 1 : 0]   config_val;
} regmap_entry_s;
```

With this information available in the structures, custom core service can create any register block, any register and any field such that the hierarchy is nowhere compromised.

[1] The constituents of this text file are chosen to be adequate to build the register model in our verification environment. This example does not restrain others to implement their own custom text file with elements essential to other verification environments.

The overhead of parsing a text file during simulation is significantly very low when compared to the overhead added by the register data base class prototypes created as specified conventionally. This comparison can be seen in the section on performance metrics.

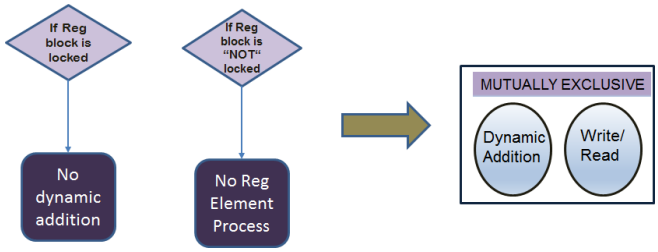
C. FUNCTIONAL IMPLEMENTATION

In the conventional UVM Register model approach, once the model is available, the registers are accessed from anywhere in the verification environment using hierarchical access to the register's handle after the model is built using build() function.

On the other hand, the functional implementation of registers using UVM RAL in the dynamic approach poses a catch-22 situation. The issue is that UVM RAL base classes are structurally rigid in that, they do not allow adding a new element in their hierarchy once the register model is locked.

```
function void uvm_reg_block::add_reg(uvm_reg rg);
   if (this.is_locked()) begin
      `uvm_error("RegModel", "Cannot add register to locked block model");
      return;
   end
```

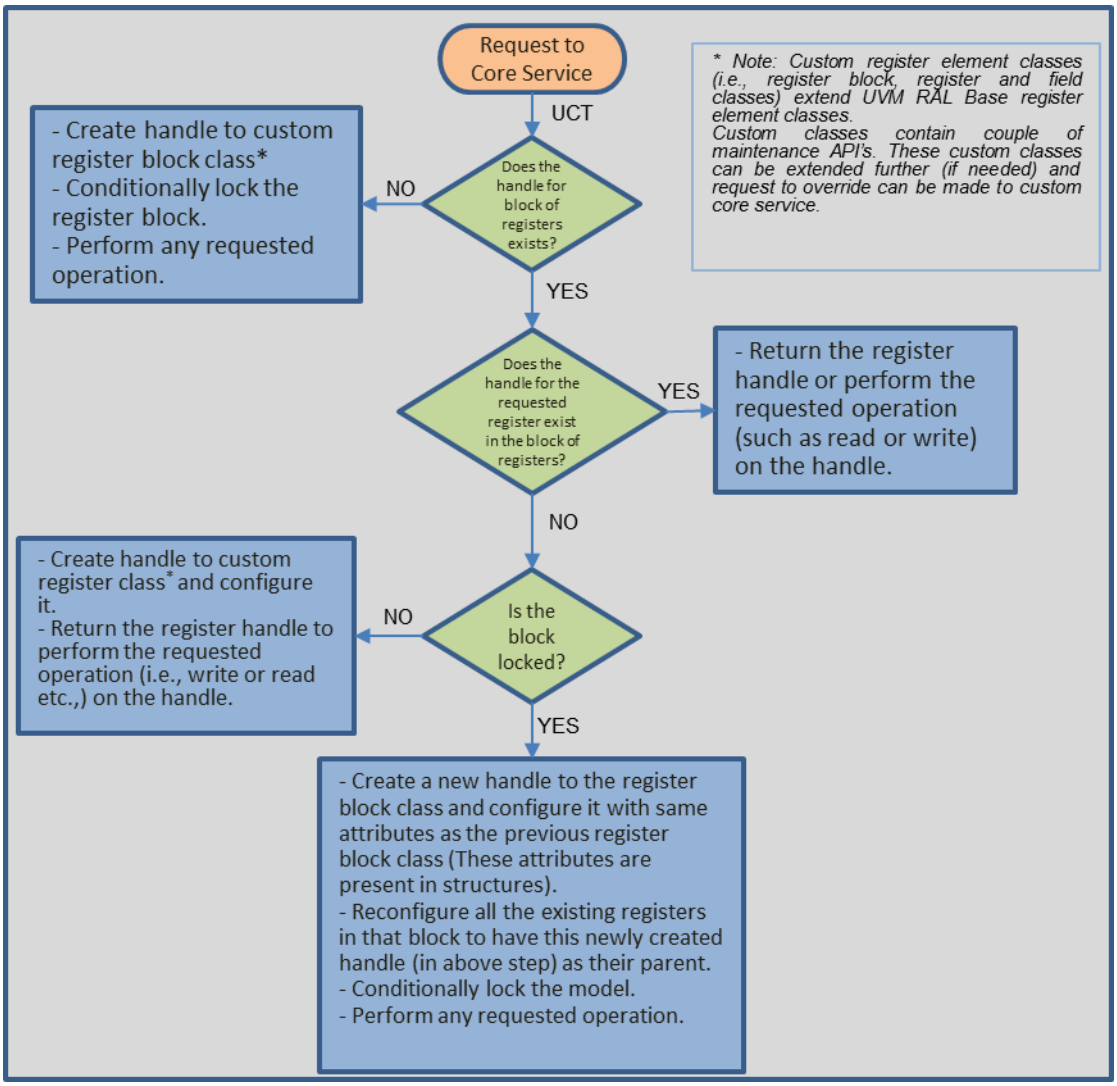
But locking of the register model cannot be avoided because to perform any register access (say write/read), the block has to be locked. Block lock condition is checked in the base classes in numerous places to ensure that block is locked before performing the access (This is done to prevent any accidental register access requests that come to the RAL before address for that register has been resolved). Hence, creating a register handle dynamically without compromising the hierarchy of the register element was a bit involved.



However, the custom core service plays a major role in resolving this quandary. This approach relies on the flexibility provided by methods in the UVM RAL base classes. Methods which are essential for configuring the hierarchy in the UVM RAL do not check whether the model is locked: configure() functions in uvm_reg_block, in uvm_reg, in uvm_reg_map, in uvm_reg_field. Similarly, utilities like set_parent, set_hdlpath, add_map etc., are flexible enough that the hierarchy can be rebuilt at any point in the simulation without statically building the model.

When the custom core service receives a request with respect to any register or register block or field, it first checks in the hierarchy of the register model if the handle of that element exists. If yes, it returns the handle as requested or performs operation on that handle. Otherwise, it creates the hierarchy, and inserts the element into that hierarchy and performs the operation.

The flow of operation can be illustrated through this flow chart:



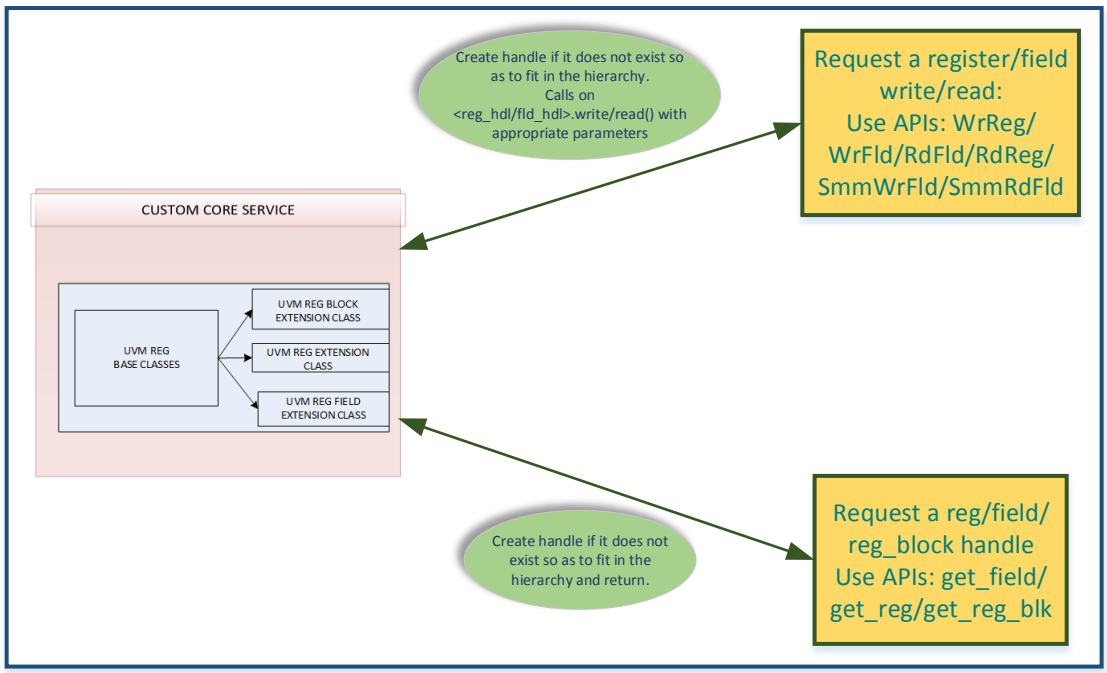
This flow is atomic at the register level, i.e., this flow need not performed for field accesses. This was because in our designs, fields were not individually accessible. Hence, to perform a bus access, the entire register's contents needs to be constructed first. Hence, even though a single field is requested in the register, the entire register is still constructed (so few superfluous fields handles may still exist in a simulation). Nevertheless, atomicity at field level choice is also provided in the custom core service to accommodate for any future changes in the designs.

The piece of code which re-configures the hierarchy is:

```
if (copy_e != COPY_NONE) begin
   foreach(cpu_regs[i]) begin
      CpuRegister r;

      $cast(r,cpu_regs[i]);

      r.configure(.blk_parent(b),
                  .regfile_parent(null),
                  .hdl_path()
                  );

      b.non_local_regs[r.get_name()] = r;

   end
end
map.configure(b,
              0,
              4,
              UVM_LITTLE_ENDIAN,
              1
              );

b.add_map(map);
b.set_backdoor(this.get_backdoor());
this.get_hdl_path(paths,this.get_default_hdl_path());
b.configure(null,
            paths[0]
            );
```

To handle requests from multiple threads while also not disrupting the hierarchy during any processing operation, semaphores and state variables are effectively maintained internally by the custom core service. Using these synchronizing elements, multiple threads can access same register (or register block or field) simultaneously through the custom core service, and all the requests will be honored intelligently by the core service and at the end of all the requests, the hierarchy of all the register elements, whose accesses have occurred till now, is still retained. That is, the handles of the registers and fields that have been created and accessed through till now will be unchanged, only the parent register block handle is replaced by a new register block handle with efficient synchronization among multiple requests.

A high level overview of custom core service approach:



This way, handles of registers are created only when core service is approached by a requestor which facilitates dynamic creation of handles. To further this whole process, the register accesses are to be performed through utilities provided by custom core service. Regardless, once register or field handles are got through custom core service (using utilities like cpu_cs.get_field_from_name()), they can be used incessantly by the

environment without any need for contacting custom core service again. This is because custom core service does not manipulate the handles of registers or fields once created. It just reconfigures the registers' hierarchy.

Next section explores on how these utilities were implemented and how they were used by the environment to perform register accesses.

D. Verification Environment using custom core service.

Vertical and horizontal re-usability were taken as prerequisites while our verification model was being contemplated. That is the block level environments (VIPs) should be easily accommodated into chip level environment with minimal or no modifications. Another rudiment is that these VIP's should be portable across different chips. For example: a VIP coded for IEEE 802.1AE MACsec should be all-purposeful to be adopted into a switch chip environment or into a PHY chip environment. The register layout would be different for these two chip level environments. However, the VIP should not be modified for every environment. To achieve such re-usability, the configuration and statistical elements of that VIP's are excluded from the register modelling layer.

Hence, the configuration class prototypes in the VIP's contain all the standard (IEEE 802.1AE MACsec considered for further illustration in this section) specified variables and constraints which are mentioned by the standards are also contained in these configuration classes. An example configuration class in this approach looks like:

```
^L
/*****************************************************************************//**
 *
 * @class MscSAParamCfg
 *
 *****************************************************************************/
virtual class MscSAParamCfg extends uvm_object;

   ^L
   /*****************************************************************************
    * @name Properties of class MscSAParamCfg
    *****************************************************************************/
   rand  ubyte8        packet_number;   /**< Packet Number as defined in IEEE 802.1AE: Clause 9.8. */
   rand  ubyte32       key;             /**< Key parameter required to protect frames. */
   rand  ubyte16       hash_key;        /**< Hash Key as defined in IEEE 802.1AE. */
   rand  ubit          sa_in_use;       /**< Indicates SA in use as prescribed in 802.1AE: Clause 10.7.14. */


   typedef struct packed {
      ubit2     sectag_an;
      ubyte8    sectag_sci;
   } sa_lookup_key_s;
```

Here, the variables packet_number, key, hash_key, sa_in_use are requisite for protecting and encrypting an Ethernet frame as per IEEE 802.1AE. The constraints specific to these variables as defined in the standard (for example: Version bit should be 0 etc.,) are captured in this class. This configuration class handle is used in the VIP as below (snapshot of only one access is mentioned below):

```
// ___ Key ___
key             =       {<< 8 {sa_entry.key}};
if (sc_entry.cipher_suite[0] == 0) begin // __ 128-bit cipher suite.

   key          =       new[16](key);

end

// ___ Hash Key ___
hash_key        =       {<< 8 {sa_entry.hash_key}};

// ___ 64-bit Packet Number ___
total_seq_num   =       (sc_entry.cipher_suite[1]) ? sa_entry.packet_number :
                                                     sa_entry.packet_number[31:0];
```

In the above screenshot, sa_entry is the handle to the aforementioned configuration class.

A layer above this VIP's configuration (i.e, MscSAParamCfg) lies project specific configuration map layer, where the standard configuration variables are mapped into project specific variables:

```
// __ Do All CPU Writes.
`CPU_WR(blk_name,    "sa_ram_reg_1",    "packet_number_0", packet_number[31:0],  shadow)
`CPU_WR(blk_name,    "sa_ram_reg_2",    "packet_number_1", packet_number[63:32], shadow)
`CPU_WR(blk_name,    "sa_ram_reg_3",    "sa_key_0",        key[31:0],            shadow)
`CPU_WR(blk_name,    "sa_ram_reg_4",    "sa_key_1",        key[63:32],           shadow)
`CPU_WR(blk_name,    "sa_ram_reg_5",    "sa_key_2",        key[95:64],           shadow)
`CPU_WR(blk_name,    "sa_ram_reg_6",    "sa_key_3",        key[127:96],          shadow)
`CPU_WR(blk_name,    "sa_ram_reg_7",    "sa_key_4",        key[159:128],         shadow)
`CPU_WR(blk_name,    "sa_ram_reg_8",    "sa_key_5",        key[191:160],         shadow)
`CPU_WR(blk_name,    "sa_ram_reg_9",    "sa_key_6",        key[223:192],         shadow)
`CPU_WR(blk_name,    "sa_ram_reg_10",   "sa_key_7",        key[255:224],         shadow)
`CPU_WR(blk_name,    "sa_ram_reg_11",   "gcm_h_0",         hash_key[31:0],       shadow)
`CPU_WR(blk_name,    "sa_ram_reg_12",   "gcm_h_1",         hash_key[63:32],      shadow)
`CPU_WR(blk_name,    "sa_ram_reg_13",   "gcm_h_2",         hash_key[95:64],      shadow)
`CPU_WR(blk_name,    "sa_ram_reg_14",   "gcm_h_3",         hash_key[127:96],     shadow)
 cpu_cs.SmmRelease(blk_name);
```

This approach of having a map layer facilitates re-usability: even when the register naming is changed or the way register mapping changes, i.e., say 256-bit key is mapped into sixteen 16-bit fields instead of eight 32-bit fields (as it was shown in the example above), the VIP or the base configuration object containing standard specified variables and their constraints remain generic.

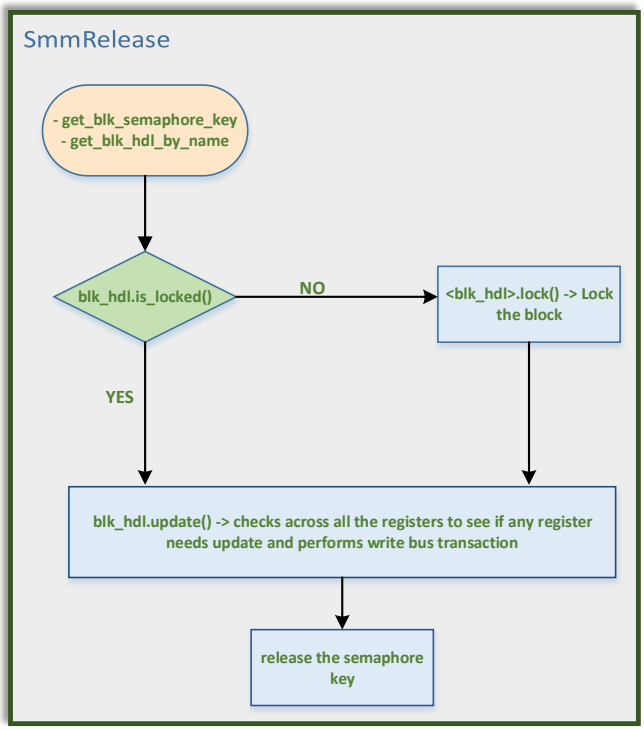
In the pre_configure phase of the environment, configuration objects of the VIPs are randomized and in the configure_phase, the randomized configuration variables are written into DUT using the `CPU_WR macro call as shown in the example above. The macro expands into:

```
^L
/*******************
 Any `defines here.
 *******************/
`define CPU_WR(BLK_NAME, REG_NAME, FLD_NAME, WRITE_VAL, SDW) \
 if (SDW == 0) begin \
    cpu_cs.WrFld(BLK_NAME, REG_NAME, FLD_NAME, WRITE_VAL); \
 end \
 else begin \
    cpu_cs.SmmWrFld(BLK_NAME, REG_NAME, FLD_NAME, WRITE_VAL); \
 end
```

“cpu_cs” is handle to the singleton class custom core service. For every call made via cpu_cs handle, the custom core service processes as per the flow chart above. If the shadow bit is set to one, then a set operation is performed via the field handle (i.e., <fld_hdl>.set()), which modifies the desired value of the register field. If the shadow bit is 0, then via the register handle a write operation is called (i.e., <reg_hdl>.write()) i.e., actual DUT write is performed.

In the above flow chart, it is mentioned that block is conditionally locked. The reason why block is not locked for every access is because the SmmWrFld function (implemented in the custom core service) internally calls <field_handle>.set() and these set (or get) functions which are implemented in the UVM RAL base classes do not check if the register block is locked. Custom core service takes advantage of this behavior and locks the model only when actual read/write methods are being called on the register/field handles. This way, all the SmmWrFld function calls can be clustered (shown in example above). And for every Smmwrfld function call, a new register is simply added into the register block without the necessity of creating new register block handle.

Once all the randomized values are set into the register database using SmmWrFld function calls, later in the configure phase, these values are written into the DUT using SmmRelease task call. SmmRelease is an API provided by the custom core service, which checks if the block referred by the block_handle is locked. If not, it locks the block and performs the update operation on the block handle. In the above mentioned example, SmmWrFld writes the value into the “m_desired” field via set function call, register block is locked only during SmmRelease task call as illustrated by the flow chart below. This minimizes the need of creating any new register block handle.



This way of clustering register set operations (using SmmWrFld operation above) is mentioned just as a way it is modelled in our verification environments. This does not preclude the user of custom core service to perform register access in any other method as they would have done using conventional UVM Register data base.

Even after this DUT configuration (or during this DUT configuration phase), if there are any requests to the register block from elsewhere, custom core service efficiently synchronizes all the requests by using semaphores which exist per register block in the custom core service and state variables in every register block.

Similarly, statistics verification done at the end of this test is made more user friendly by custom core service approach. In our verification environments, for most of the tests, relevant statistics verification is done at the end of a test. The environment predicts the statistics (especially octet counters, frame counters etc.,) during the entire test, and at the end this predicted value contained in the statistics class variables is compared against the DUT's value. Here also, two layered approach is followed wherein, the standard specified statistics variables are specified in a stats class, and the other VIP constituents (say xfers) update these variables as per the scenario. At the end of the test, these statistics are mapped into registers as below:

```
cpu_cs.SetPredict(block, "in_pkts_ctrl_lower"     , "in_pkts_ctrl_lower"     , ig_stat_cntr_out.InPktsCtrl_lsb[31:0]);
cpu_cs.SetPredict(block, "in_pkts_ctrl_upper"     , "in_pkts_ctrl_upper"     , ig_stat_cntr_out.InPktsCtrl_lsb[39:32]);
cpu_cs.SetPredict(block, "in_pkts_no_tag_lower"   , "in_pkts_no_tag_lower"   , ig_stat_cntr_out.InPktsNoTag[31:0]);
cpu_cs.SetPredict(block, "in_pkts_no_tag_upper"   , "in_pkts_no_tag_upper"   , ig_stat_cntr_out.InPktsNoTag[39:32]);
cpu_cs.SetPredict(block, "in_pkts_untagged_lower" , "in_pkts_untagged_lower" , ig_stat_cntr_out.InPktsUntagged[31:0]);
cpu_cs.SetPredict(block, "in_pkts_untagged_upper" , "in_pkts_untagged_upper" , ig_stat_cntr_out.InPktsUntagged[39:32]);
```

SetPredict is a utility which writes into the mirrored value of the fields. Similar to SmmWrFld, SetPredict does not lock the register block. At the end of mapping all the relevant statistics into the register blocks, similar to SmmRelease, MirrorBlk task is called on the custom core service handle. This mirror method locks the register block, and calls the mirror() method of the register block handle. This mirror method (as implemented in the UVM RAL base classes), iteratively goes through all the registers in the register block and performs read operation on all those registers. While iterating through the handles, configuration fields in that register block are also processed (since mirror function calls mirror on all its register handles). This may result in register

transactions for configuration fields as well, which is quite unnecessary at this point in the verification. In our verification models, for most of the tests, during this stage, configuration values are not needed.

Hence, before performing statistics mapping (i.e., before performing SetPredict function calls above), existing register block handles are deleted. This way, only those registers whose SetPredict functions are called will exist in the block. And mirror() method will iterate only through these registers which limits the number of bus transactions only to the necessity. Using conventional register model approach, if irrelevant handles are not deleted, even they would have been mirrored by the register model base classes and to delete those irrelevant handles additional methods would have to be written which is here accomplished using one single statement of making the register block handle = null.

E. Customization of register classes

Some register classes may have to be customized to include specific features. Using UVM register model approach, these are generated by the script itself. Custom core service approach also facilitates such usage by providing overrides. Custom core service maintains a queue of overriding registers, register fields and register groups. Since every access of register occurs through custom core service (at the least, creation should go through custom core service), the core service checks for the overriding list to see if there was any override request on this register element, and if there is a request, it creates the handle of new register element according to the overriding type.

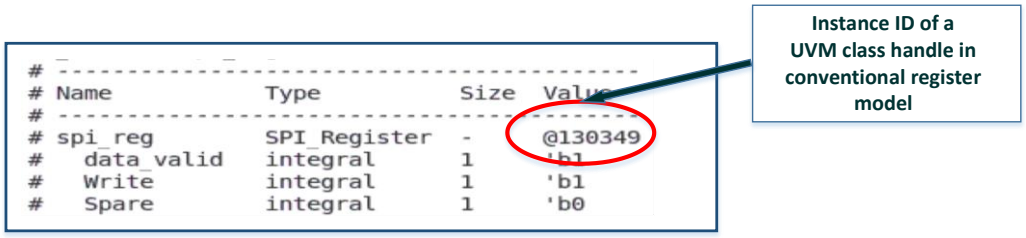
Furthermore, this method of overriding does not impede the user in performing factory specified overrides as well, because custom core service creates handles using create function call, and uvm_object_utils macro call is present in base class prototypes (i.e., uvm_reg, uvm_reg_field etc.,).

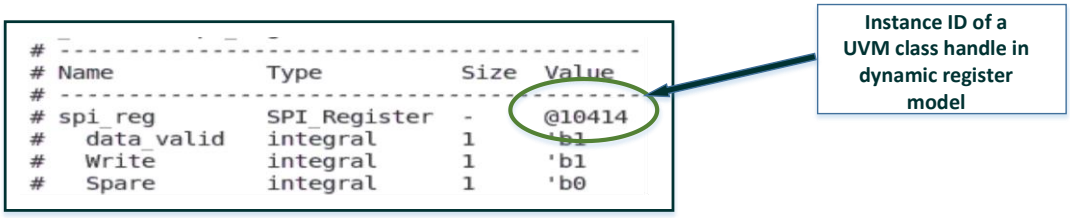
IV. PERFORMANCE METRICS

As the huge static data base is no more compiled and loaded in the simulation, the improvement in performance is quite significant. Simultaneously, since number of handles are significantly less, the performance improvement is noteworthy.

The chip level environment which was used to mention metrics in this section contained 2054 register class prototypes, with around 3960 fields. Questasim 10.0d simulator was used to simulate the environment.

This is a snapshot of comparison of number of handles created in the simulation of this test. Instance identifiers represent number of uvm class handles created in a test. This is internally maintained by uvm_object base class and this instance identifier is incremented when handle is created for any class extending from uvm_object (including components).





Below are the performance metrics for simulation statistics for 529 register writes performed in conventional register model deployment (i.e., Model is prototyped and built statically), and in dynamic register model deployment. The simulation ended at the same clock cycle in both the approaches with the difference in simulation time as mentioned below.

ATTRIBUTE	CONVENTIONAL REG MODEL DEPLOYMENT	DYNAMIC REGISTER DATABASE DEPLOYMENT
SIMULATION TIME	3845.0685 Seconds	3444.6392 Seconds
PROCESS MEMORY USAGE	721105428 Bytes	595249622 Bytes
SIM WORK DIR DISK USAGE	403 MB	282 MB
COMPILED DATA BASE DISK USAGE	17 M	4.2 M

Improvement which is seen only one time for all the tests:

- Compiled data base disk usage improvement from 17MB to 4.2 MB. This disk usage improvement is seen once for a verification environment.
- Compile time improvement.

Statistics improvements which can be seen for every test:

- Simulation time improvement for 529 register writes is around 400 seconds which is roughly 6.7 minutes. Aforementioned statistics are from a test run on CPU server which is not loaded with any other jobs. However, when the server is fully loaded with jobs (which is very common when test regressions are fired), the improvement is much higher with dynamic register data base deployment.
- Process memory usage: i.e., memory consumption by the simulation is improved by roughly around 126 Megabytes.
- Similarly, the improvement in disk space consumption of the work directory of the test is roughly 121 Megabytes.

Furthermore, these improvement statistics are mentioned for a single test. The improvement augments when a regression of hundreds of tests are simulated, which would save hours of simulation time. Similarly, the disk usage improvement with dynamic register data base creation would be in terms of gigabytes for a test regression.

Various iterations are run under different scenarios (i.e., when CPU server is free, when CPU server if fully loaded etc.,), and improvement was observed in all those cases. At chip level, for around 140 register accesses, improvement metrics are:

ATTRIBUTE	CONVENTIONAL REG MODEL DEPLOYMENT	DYNAMIC REGISTER DATABASE DEPLOYMENT
SIMULATION TIME	1197.0863 seconds	1177.90979 seconds
PROCESS MEMORY USAGE	707768340 Bytes	568642006 Bytes
SIM WORK DIR DISK USAGE	397 MB	276 MB
COMPILED DATA BASE DISK USAGE	17 M	4.2 M

As can be seen, the performance improvement increases as number of writes done is increased (from 140 to 529). At the block level environment, for around 50 register accesses, improvement in terms of process time observed is around 20 seconds.

V. CONCLUSION

The main motive behind projecting the approach deployed in our verification environments is to advocate exploring a dynamic register data base model than a static model to truly exploit the benefits of the verification methodology. This approach provides a comprehensive solution independent of the complexity of the register data base. This approach eliminated the dependency on a complex code generator for creating register model and this semi static model could be deployed into our verification environment with minimal effort. The code presented in this paper is being deployed into a live project.

ACKNOWLEDGMENTS

My sincere thanks to Sailesh Rupani, Manager at Microsemi Corporation for his continuous support throughout this proposal's implementation.

I also thank Mobeen Syed, Manager, Sivakranth Varanasi and Sreekar Kokkonda for their brainstorming effort put behind this proposal.

I further thank Uma Athuluri, Component Design Engineer at Intel Corporation, with whose discussions, this idea was put to practice.

REFERENCES

[1] Mark Litterick, Marcus Harnisch, "Advanced UVM Register Modeling - There's More Than One Way to Skin A Reg", DVCon 2014.