2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA

Leveraging IEEE 1800.2-2017 UVM for improved RAL modelling

Vikas Sharma, Mentor, A Siemens Business (vikas_sharma@mentor.com)
Manoj Manu, Mentor, A Siemens Business (manoj_manu@mentor.com)
Ankit Garg, Mentor, A Siemens Business (ankit_garg2@mentor.com)

Abstract **— The implementation of UVM-RAL** [1] **verification environment which contains multiple processors or masters, and access the same registers (*uvm_reg_block*) with dynamic address mapping is a challenging task, especially when no standard solution is available in the traditional UVM. Until now, the industry was solving such challenges by implementing home-grown techniques that involved less efficient codes and low reusable quotients. A year ago, Accellera introduced new reference implementations aligned with the IEEE 1800.2-2017 standard for UVM with many improvements and new features. This paper aims to provide IEEE 1800.2 UVM-based solutions to complex register blocks or structures that cannot be modelled statically and require multiple mapping and remapping of registers (*uvm_reg*) dynamically. The proposed solution also tries to accommodate any potential vertical reuse and future expansions of dynamic address mapping.**

Keywords— System Verilog; UVM; IEEE 1800.2-2017; Register Abstract Layer RAL; uvm_reg; uvm_reg_block; multiple address mapping; dynamic address mapping;

I. Introduction

Before discussing paper intent and its schema, let's delve into SystemVerilog [2], which is a powerful language primarily developed to facilitate effective verification of hardware designs. The SystemVerilog language standard includes hardware descriptions, test bench construction, assertions, coverage, constrained randomization, and object-oriented programming features. Although SystemVerilog includes a plethora of features for hardware design verification, building a verification environment using SystemVerilog is challenging for anyone especially the beginners. This is due to the absence of a standard verification architecture for creating SystemVerilog test benches. The Universal Verification Methodology (UVM) addresses this gap and helps in overcoming the verification-related challenges.

UVM [1] provides one of the best verification architectures for creating SystemVerilog-based test benches, thereby eliminating the effort and time spent in creating complex test benches. To facilitate the development of efficient verification environments, UVM provides various building blocks that include a set of base classes, reusable components, and steps to create a test bench from reusable components.

UVM [1] also provides an abstraction layer known as the register abstraction layer (RAL) for conveniently accessing registers and their contents and memory locations within a DUT. The UVM-RAL provides a base library for modelling the register space and abstracting the read and write operations to the corresponding mapped register space in the DUT. It not only helps in tracking the register content of the DUT, but also helps in mirroring them with the ones which are modelled inside UVM register model. The traditional UVM-RAL meets most verification requirements upfront except some exceptions, such as static locking and registration of a register model. To support dynamic address remapping, the IEEE 1800.2 UVM [3] offers new APIs and methods.

The IEEE 1800.2 UVM-RAL [3] allows register blocks and mappings to be unlocked, unregistered, and remapped at different addresses before relocking them again. This paper aims to address the challenges faced in dynamic address remapping and multiple mapping of shared registers and their applications using the IEEE 1800.2 UVM-RAL. The techniques described in this paper cover remapping of registers along with new IEEE 1800.2 UVM [3] references that are described with their advantages and potential use cases.

II. Traditional UVM-RAL

A typical register model consists of a hierarchy of blocks which contain sub-blocks, registers, register files, memories, and register map. The register model data structure must be organized in a way that it completely reflects

the DUT hierarchy and aids in the writing of abstract modelling. After its integration, the register model is used by a test bench user to create reusable sequences that can access hardware blocks and memory. In the case of a block-level register model, the register block is most likely to be mapped to a single address map. The register map defines the address space offsets of one or more registers or memories as specified by the specific agent or bus interface. There is no mechanism of changing the address map once it is built and locked.

Presently, CPU subsystems/interconnect require additional capabilities in their register models that include dynamic address mapping of shared registers. There could be multiple register blocks containing other register model blocks for each of the subcomponents in the subsystem, and their register maps might be associated with different address maps during runtime. Therefore, it is a challenge to fulfil such conditions because it requires locking and unlocking of the register models. The traditional UVM-RAL modelling is static, and does not provide any standard API or recommended guidelines to address the aforesaid challenges. Further, the traditional UVM-RAL modelling is often coded ineffectively to meet verification requirements. In addition, the traditional UVM-RAL offers only the static ***lock_model()*** API and allows static locking and registration of the register model at the build time. This makes it harder to unlock the register mappings, which could be required in an advanced verification plan. Attempting to reuse register mappings in the subsystem level or in multiple mapping environments becomes difficult because of unavailability of relevant APIs. Currently, no standard method is available to unlock the register mappings and implement them for further vertical reuse. This is also true for the verification environments with multiple processors or masters that access the same registers or memories with different register maps.

- The ***lock_model()*** API checks if a register model is locked. If the model is unlocked, the API recursively locks the entire register model and builds the address map. When the model is locked, no further structural changes can be made. Therefore, all the sub-blocks, maps, and registers must be created before the ***lock_model()*** API is called. This API finalizes the address mappings

```
// Once locked, no further structural
// changes, such as adding registers
// or memories, can be made.
function void uvm_reg_block::lock_model();

   if (is_locked())
     return;

   locked = 1;

   foreach (regs[rg_]) begin
      uvm_reg rg = rg_;
      rg.Xlock_modelX();
   end

   foreach (mems[mem_]) begin
      uvm_mem mem = mem_;
      mem.Xlock_modelX();
   end

   foreach (blks[blk_]) begin
      uvm_reg_block blk=blk_;
      blk.lock_model();
   end
```

III. IEEE-1800.2-2017 UVM-RAL

Several online forums [4] promoting UVM education and discussions are witnessing an increasing number of queries on dynamic and multiple address mapping. The solutions [6] for such queries propose different implementations that involve the same set of steps, which include extending the base class and using APIs which do not check whether the register model is locked or not.

Last year, Accellera launched an upgrade to UVM 1.2 in conjunction with IEEE by providing a new 1800.2 UVM [3] standard. As revealed [5], the new standard comes with some bug fixes and new, modified, and deprecated APIs. The deprecated APIs can be activated again by using +UVM_ENABLE_DEPRECATED_API. Now, all UVM register base classes are abstract (virtual) classes. A significant feature of the IEEE-1800.2 UVM standard that differentiates it from the traditional UVM-RAL is its register layer APIs, which allow removal or addition of registers and memories along with an association to different address maps.

IV. DYNAMIC ADDRESS MAP

We should look for potential usage of dynamic register map in cases where a system requires to change its register map on the fly because of many reasons, such as:

- Reconfigurations, wherein an address map has to be reconfigured
- States, wherein different states have different access policies
- Hot-Join, wherein a new host joins the system on the fly
- Access policies, wherein a change is required in access policies of sub-blocks
- Multiple address mapping, wherein a register space is to be accessed by more than one master with different address offsets

Contrary to the traditional UVM-RAL [1], the IEEE 1800.2 UVM [3] register modelling RAL is dynamic and provides new functions, ***unlock_model()*** and ***unregister()***, to unlock and unregister a register model that is already locked and registered. The IEEE-1800.2 UVM-RAL allows users to easily rebuild the address hierarchy (maps) from registers as needed at runtime by locking the register model again in the simulation dynamically.

- The ***unlock_model()*** function brings back the register mode to a state before locking so that a subsequent ***lock_model()*** API can be issued at any time during the simulation. It recursively unlocks the entire register model and sub-blocks, and moves the model to a state where the locked bit is set to 0.

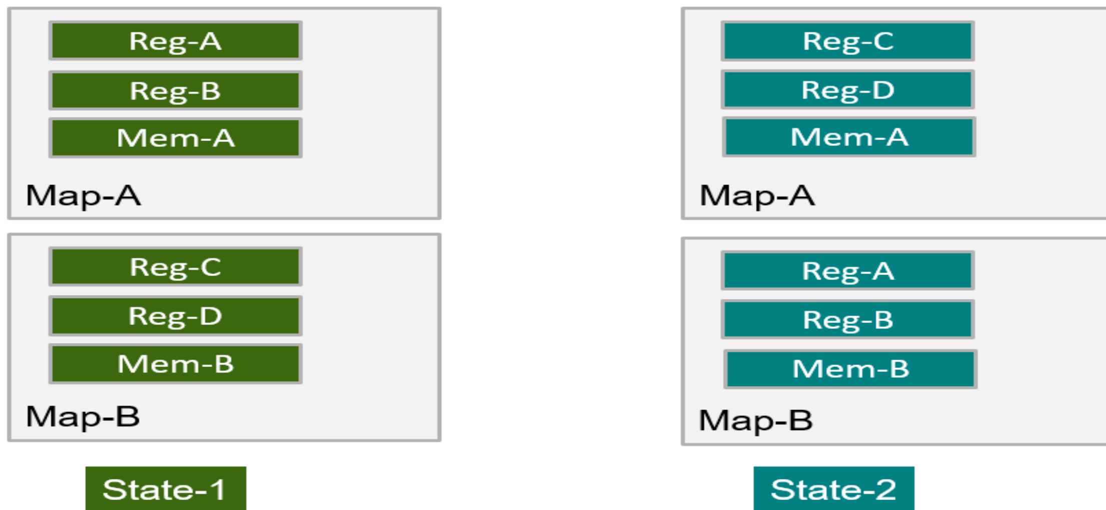
```
// Brings back the register mode to a
// state before lock_model() so that a
// subsequent lock_model() can be issued
//
virtual function void unlock_model();

  bit s[uvm_reg_block]=m_roots;
  m_roots.delete();

   foreach (blks[blk_])
        blk_.unlock_model();

   m_roots=s;
   foreach(m_roots[b])
     m_roots[b]=0;

   locked=0;
endfunction
```

- The ***unregister()*** function unregisters all the content from the map recursively and removes all knowledge of the map recursively from other objects, such as regs, mems, and vregs. This enables the reuse of the map content and objects with a fresh map instance by using the add_** APIs. If ***this.unregister(map)*** is called then a full register map will get unregistered. When only a specific register or memory address map needs to be changed, the ***map.unregister(reg/mem)*** function is called.

```
// Removes all knowledge of map m and all
// regs|mems|vregs contained in m from the block
//
virtual function void unregister(uvm_reg_map m);

  foreach(regs[idx]) begin
   if(idx.is_in_map(m))
     regs.delete(idx);
  end
  foreach(mems[idx]) begin
    if(idx.is_in_map(m))
      mems.delete(idx);
  end
  foreach(vregs[idx]) begin
    if(idx.is_in_map(m))
      vregs.delete(idx);
  end
  maps.delete(m);

endfunction
```

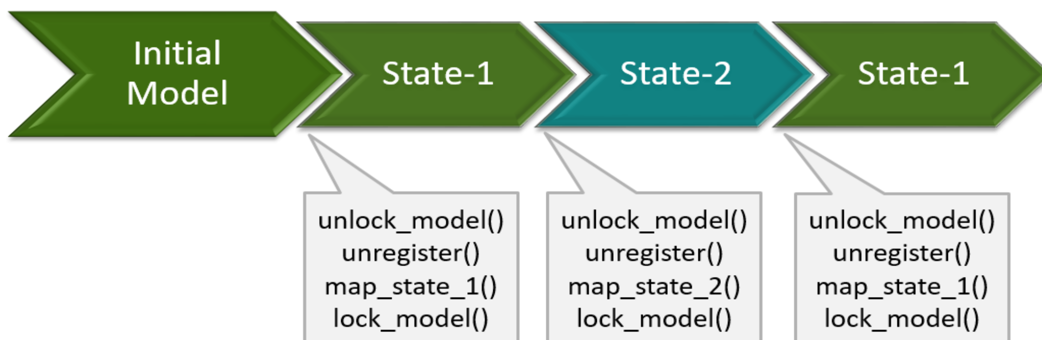
V. Implementation

The following block diagrams describe a sample requirement [5] of a system with two states, State-1 and State-2, and two sets of register maps, Map-A and Map-B.



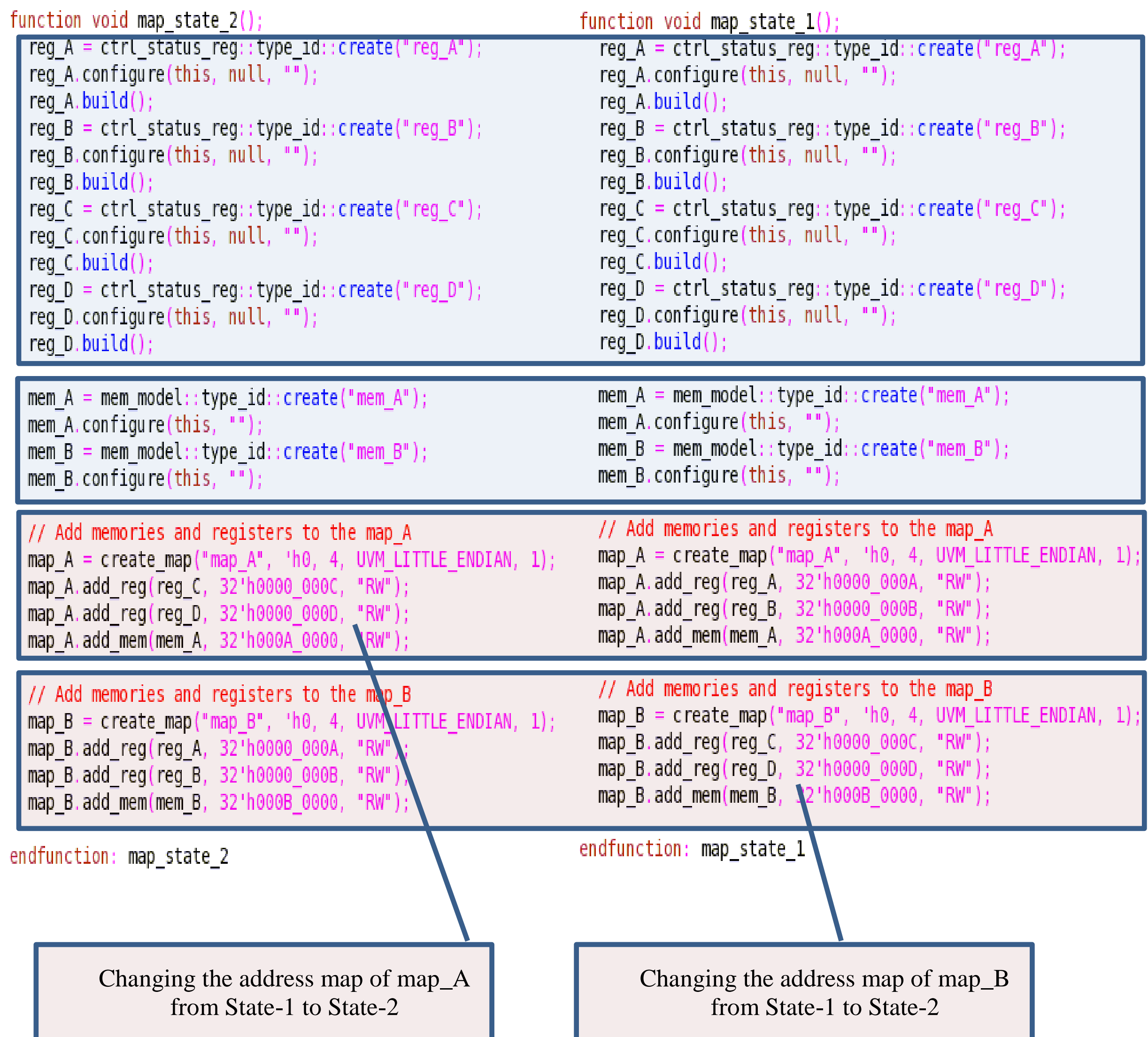
Let's suppose that there is a test requirement [5] where:

1. State-1 and State-2 specify different register structures and address maps as explained in the above figure.
2. The simulation starts from the initial model and moves the system from State-1 to State-2, and after some time the system returns to State-1 as explained in the figure below.
3. At every stage, user needs to reconfigure the model, but it cannot be done because the system is already locked during initial model setup.



```
// Remapping the register model as per states
function void map_as_per_states(state_e state);

  unlock_model();     // Unlocks the model

  unregister(map_A); // Unregisters the address map_A
  unregister(map_B); // Unregisters the address map_B
  map_A = null;      // Delete the address map_A
  map_B = null;      // Delete the address map_B

  case(state)
    STATE_1 : map_state_1();
    STATE_2 : map_state_2();
  endcase

  // Locks the model
  lock_model();

endfunction: map_as_per_states
```

Now, let's examine the code shown in the figure above. It demonstrates the ***map_as_per_states()*** function, which handles the register mapping based on the states of the system. This function is called every time the system state changes. In this example code, the function unlocks the register model and unregisters the register maps. Thereafter, it calls the ***map_state_1()*** and ***map_state_2()*** functions shown in the figures below. To ensure that the register model is not altered, the ***map_as_per_states()*** function locks the model at the end.



Once you consider the above mentioned steps, adopting the IEEE 1800.2 UVM-RAL [3] can significantly reduce the amount of time it takes to achieve dynamic addressing and multiple mapping. The IEEE 1800.2 UVM-RAL provides various built-in APIs to handle many of the common to-dos used in the following process. To achieve a dynamic address map, there is a need to understand some of the basic concepts explained above and perform the following steps for its development.

1. Build a register model in the same way as in traditional UVM.
2. Register the register map using the ***add_reg()*** API.
3. Do not perform any register read or write operation without locking the register model.
4. Lock the model by calling the ***lock_model()*** API.
5. Unlock the model during simulation by calling the ***unlock_model()*** API.
6. Unregister the register map by calling the ***unregister()*** API.
7. Create new maps, add registers, connect into blocks.
8. Unmap existing registers and remap them again.
9. Lock the register model again by calling the ***lock_model()*** API.

VI. SUMMARY

The IEEE 1800.2 UVM-based solution proposed in this paper achieves the following objectives:

- Identified the problem of static nature of the RAL model in traditional UVM.
- Introduced new IEEE 1800.2 UVM RAL model.
- Proposed solutions targeting dynamic address mapping.
- Illustrated an example changing the register model dynamically using system states.
- Demonstrated the usage of new APIs provided in the IEEE 1800.2 UVM RAL model.

VII. REFERENCES

[1] Accellea UVM: https://www.accellera.org/downloads/standards/uvm
[2] IEEE 1800-2017: SystemVerilog (SV)
[3] IEEE 1800.2-2017: Universal Verification Methodology (UVM)
[4] Online Forum: https://verificationacademy.com
[5] DVCon US-20I8: IEEE-Compatible UVM Reference Implementation and Verification Components
[6] DVCon INDIA-2015: UVM-RAL: Registers on demand Elimination of the unnecessary