Taming UVM 1.1d RAL in a Multi-Interface, Multi-Mode Environment

Steven K. Sherman

AMD
Boxborough, Massachusetts, USA

www.amd.com

ABSTRACT

A verification environment that features multiple interfaces to registers that include multiple access modes introduces challenges to UVM 1.1d RAL (and not yet addressed in UVM 1.2 RAL). This paper discusses features of legacy RAL not ported to UVM 1.1d. It demonstrates how to support multiple interfaces to registers and extend RAL to access registers in multiple modes via predictors, suggesting future enhancements for UVM RAL.

Table of Contents

Table of Figures

1. Introduction

During the past several years, RAL has emerged as a standard for expressing and processing register information. It was integrated into UVM, but the integration is imperfect. Specifically, it currently lacks support for registers that may be accessed through multiple interfaces and via multiple modes. However, RAL in UVM can be used in a multi-interface, multi-mode environment with some modifications. The modifications presented here also suggest directions for future development and enhancement of RAL support in UVM.

2. Register Access Challenges

RAL as implemented in UVM readily supports single interfaces to registers as well as single addresses and access modes to individual registers through the use of submaps. Challenges arise when one attempts to introduce multiple interfaces or multiple modes of access to individual registers, such as with the design shown in Figure 1. This simplified design features registers that can be accessed via "BUS" and "USB" interfaces. For this example, assume also that the registers may be written in "boot" mode. In "run" mode, the registers are protected. They may then be read but not written.

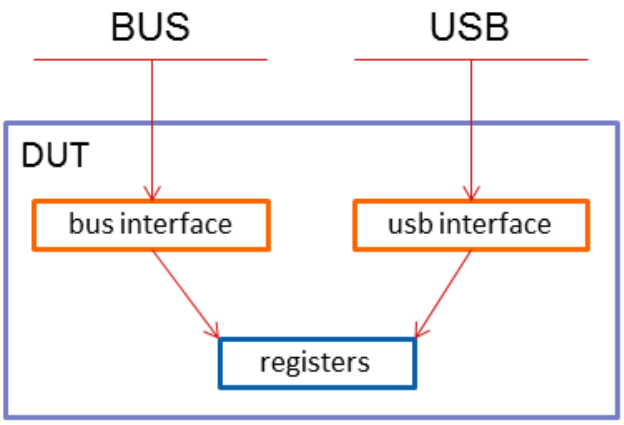

Figure 1 - Example of Two Interfaces

Multiple Interface Challenge

A verification environment that features multiple interfaces to a register model introduces challenges to UVM 1.1d RAL. A few years ago, attention was drawn to the issue of how to support multiple interfaces to a register in UVM RAL [1]. This issue was closed and not accepted for UVM RAL. The issue was again iterated [2] and appears to be still pending [3]. In short, there is demand for support for multiple interfaces to registers, but this is not readily supported in UVM RAL 1.2 or earlier versions.

Hereafter in this paper, UVM RAL version 1.1d is used, but version will be omitted and the term will simply be "UVM RAL." The term for VMM RAL or for potentially other versions of RAL that support multiple interfaces will be "legacy RAL."

In theory, access to multiple interfaces should be supported in UVM RAL [4]. The example of Figure 1 is similar to an example cited in [4] where a block may specify two physical interfaces (or "domains") for a register. However, "domain" has a separate meaning in UVM, related to the phasing schedule [5]. What was "domain" now appears to be replaced with "submap," albeit not entirely.

In legacy RAL, a block may contain multiple domains. In UVM RAL, a block may contain multiple address maps with multiple submaps per address map. In both legacy and UVM RAL, a block may contain a set of registers.

In legacy RAL, a block can readily support more than one physical interface [4]. In UVM RAL, only one parent address map is supported per "submap" [3]. This can pose a challenge to verification of a design with UVM RAL with multiple interfaces such as that shown in Figure 1. This limitation precludes assigning a register submap to multiple address maps (parents) in a block.

The enhancements discussed will be put into predictors (passive components) instead of sequences or drivers (active components) in order to be consistent with the policy of exporting passive IP to clients [6]. Additional policies include using the source code pre-processing provided with IP given to us. Another policy is to simplify the pre-processing required of our clients with our exported code. Thus, running the RAL code generators provided to us, we use the generated RAL register model "as is." We then try to limit the amount of processing required of our clients with our exported code.

Multiple Mode Challenge

Within a single interface, a register may be assigned to several submaps, enabling register access via these submaps. A submap controls several aspects of register access including offset (for address) and rights (such as "RW, "RO" and "WO"). With multiple mode support, a register is accessible via multiple address ranges and multiple definitions of rights.

It may be desirable to support multiple modes for a register accessed via single interface but with the same address. For example, in a "configure" mode, access to a register at an address may be "RW" so that it may be configured. Then, in "run" mode it may necessary to have access restricted to "RO" for the register at that address. If a design flips between modes, such as during a "warm reset" then it may be necessary to dynamically flip between modes.

There is no difficulty in assigning multiple modes to a register via submap in UVM RAL. The challenge here is subtle in that a problem may arise in UVM predictors when transferring a packet from a bus to the RAL environment. The default predictor invokes the internal RAL register function, "get_local_map" which may return the first submap that has as its parent the map assigned to the predictor. As there may be multiple submaps assigned to the same parent map for this register, this can result in selection of the wrong map.

A predictor may support all register addresses and modes and be assigned to a bus interface. That interface has a corresponding root address map assigned to the predictor during the connect phase. The root address map may be the parent for all submaps associated with the register, hence the issue. One may elect to use several submaps to enable register access via multiple modes. This issue may then result in errors due to the wrong address being detected or to application of incorrect rights. Thus, support for multiple modes is a challenge in UVM RAL.

Dual Hierarchies

The challenges of multiple interfaces and modes become more complicated with the imposition of dual hierarchies necessary for register instancing and access. A block serves as a container for blocks, maps and registers. Among the maps of the top block, one is designated as the "default."

Maps and blocks contained within the top block comprise two hierarchies at the bottom of which are the register model instances (objects). One hierarchy determines the usual testbench path to the register object. This includes the root block and lower level blocks, the lowest of which contain the register objects as shown in Figure 2.

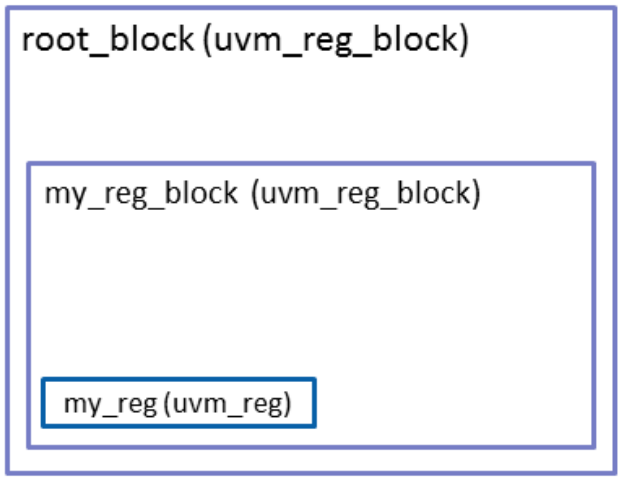

Figure 2 - Register Block Hierarchy

Within RAL a second hierarchy is imposed based upon register maps. This hierarchy of maps is traversed to determine, for example, specific register address and access rights. It is comprised of a root (or default) map at the top level with lower level submaps, the lowest of which completes the addressing and other details necessary for register access as shown in Figure 3.

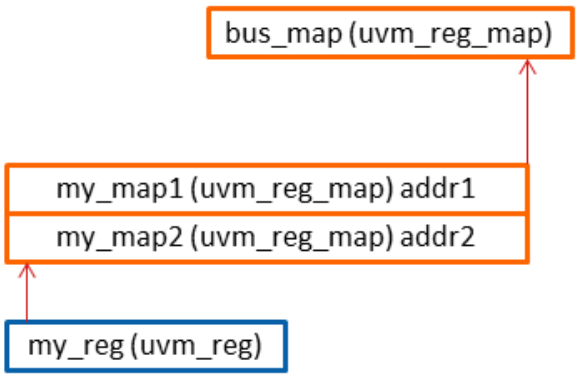

Figure 3 - Register Map Hierarchy

Maps are added as submaps to parent maps with the uvm_reg_map "add_submap" function. A map may only be added once to a parent map in a parent block and may only have one parent.

Registers are added to maps with the uvm_reg_map "add_reg" function. A register may only be added once to a map and the map must be contained in the same block as the register. However, a register may be added to several maps as parents.

3. Interfaces and Modes

Single Interface and Mode

UVM RAL readily supports registers with single interfaces and single modes with a hierarchy that expresses register structure. A register with a single interface is defined as a register that may be accessed via only one hardware path or port. UVM RAL as currently implemented seems to assume that all registers will be of the single interface, single mode type. This is shown

in Figure 4 where there is a single interface (via "bus_map") to a register ("my_reg"). The register may be accessed via non-equivalent addresses, "addr1" and "addr2."

Submaps support address aliasing by layering a second hierarchy upon the initial register structure. There is no issue with assigning a register to more than one submap in the hierarchy with the block holding the register. There is also no issue with having both submaps assigned to the parent map, "bus_map." This register may be accessed over the interface with a predictor.

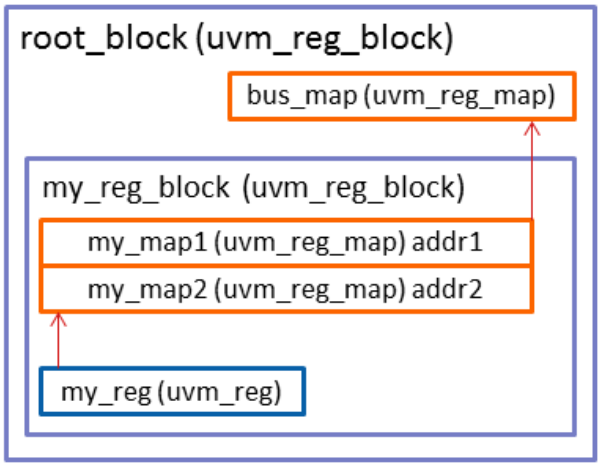

Figure 4 - Single Interface

UVM Predictor

A predictor customized for a bus interface (via extension of uvm_reg_predictor) can receive bus transactions and handle related register operations and updates. The predictor uses function "write" to accept the bus transaction (which may actually be a write or a read), call the "bus2reg" adapter function for any needed conversion, look up the register object, and do appropriate updating and checking of data in the register object.

The register object is found based on the address passed in, using the top-level map set in the predictor during the connect phase and calling map function "get_reg_by_offset". However, this function does not return the "local_map" for the register, needed to perform checks and to access operations for the register object. The "local_map" is normally obtained with a call to the register function, "get_local_map".

```
local_map = rg.get_local_map(map,"predictor::write()");
map_info = local_map.get_reg_map_info(rg);

…

foreach (map_info.addr[i]) begin
  if (rw.addr == map_info.addr[i]) begin

  …
```

Function "get_local_map" recursively searches the list of maps attached to the register object, returning the first that ultimately has as a parent that matches the top-level map passed into the predictor during the connect phase. There is only one "local_map" when there is a single interface and a single mode for the register, so this should be the correct "local_map" in this case. Within the respective "map_info" there may be several addresses collected from several submaps attached to this register object.

The predictor relies upon the map passed in for access to UVM RAL. If the bus interface is supported by a client in a custom fashion, then the submap type may also be provided and must be supported within the UVM RAL infrastructure.

Multiple Interfaces

Multiple interfaces occur when there are two or more hardware paths to a register object. This could happen if, for example, a module contains registers that can be accessed via two independent ports, perhaps differing only in the top-level offsets as set by "bus_map" and "usb_map". Figure 5 shows a UVM RAL implementation to support interfaces "bus" and "usb." This is illegal because UVM RAL does not support submaps with multiple parents. In addition, this forces use of the same submap types in accessing the registers during simulation. This is a problem if, for example, a vendor requires submap extensions in support of a particular interface.

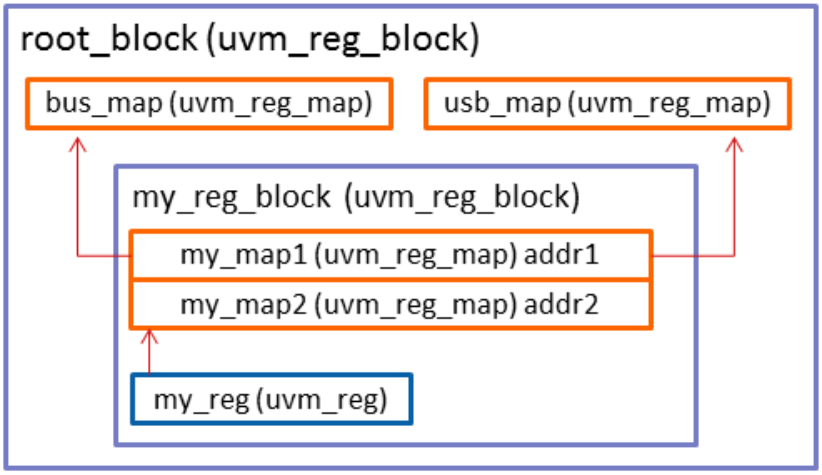

Figure 5 - Illegal Multiple Interfaces

Figure 6 shows a legal UVM RAL implementation that supports interfaces "bus" and "usb" with respective maps at the top level. This also has the two top-level maps but differs in that a new submap type is included to support the vendor-provided "usb" interface which requires extended submap type "usb_reg_map."

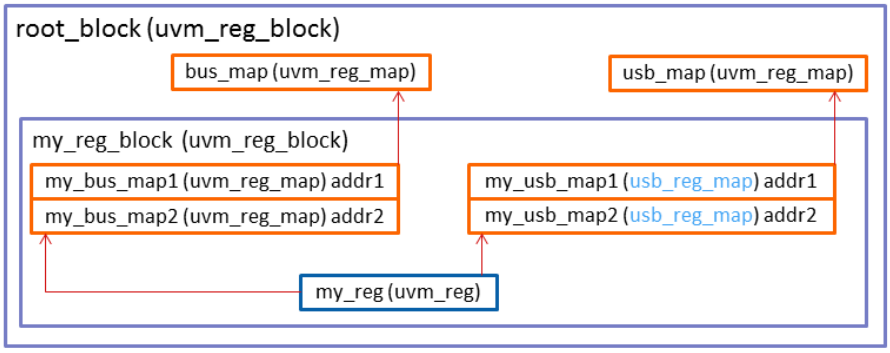

Figure 6 - Legal Multiple Interfaces

This is an imperfect implementation. It results in duplication of submaps within the "my_reg_block," some of "uvm_reg_map" type and some of "usb_reg_map" type. Though the submaps may be virtually identical, separate instances are required because maps may only have single parents. This is likely to have overlapping or duplicate addresses resulting in UVM RAL warnings. Though ugly, these warnings will be permitted here.

Multiple Modes

Multiple register modes can occur when a register object may be accessed with more than one set of rights. Each set of rights may require that a unique submap be created to reflect each access mechanism. A submap can only hold one definition of rights. Rights are designated as a register is added to a submap (with "add_reg"). Also, a register may be added only once to a submap. Thus, multiple submaps are required to support multiple access modes for a register.

Within the context of this paper, multiple modes of a register can exist when a register object can be accessed via multiple submaps with typically overlapping addresses and differing rights within the same interface as shown in Figure 7.

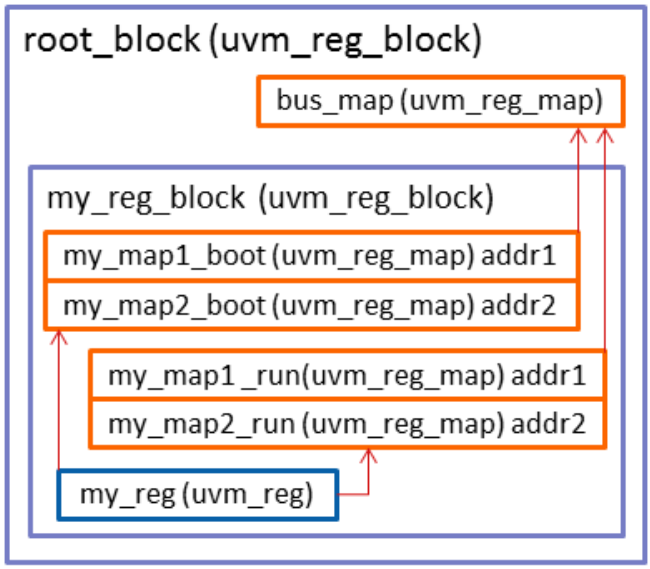

Figure 7 - Multiple Modes

Multiple register modes are likely not explicitly supported in legacy RAL due to potential conflicts as a result of matching address offsets among the submaps in the block. They are certainly not supported in UVM RAL and result in warnings due to address overlap.

As with the single-mode case of Figure 5, the register object of Figure 7 is found based on the address passed in, using the top-level map set in the predictor during the connect phase and calling map function "get_reg_by_offset". This function searches for the first "local_map" for the register. At this point, it is assumed that address offset is unique, so it should find the correct register, regardless of whether there is more than one way to find it due to inclusion of modes with overlapping addresses.

Unlike the example of Figure 4, "get_local_map" if used for the example shown in Figure 6 may return an incorrect map. This is a situation that warrants the type of warning about overlapping addresses ignored in Figure 6. This can happen because the function will return the first with matching address that ultimately has as a parent the top-level map. This is the correct top-level map for the interface. But, as there may now be more than one matching address, this could pick the wrong submap, resulting in incorrect rights.

The configuration depicted in Figure 7 accurately represents the configuration expected and used by clients. As far as clients are concerned, there is no multiple interface problem because that is a simulation issue. In fact, they expect this configuration as it fully represents the address mapping of the hardware including the various modes supported. They expect to see just the "bus_map" at the top and are disinterested in other, virtually identical maps added simply to support verification.

Multiple Interfaces and Modes

An environment may feature both multiple interfaces and multiple modes, compounding the aforementioned challenges to UVM RAL, as shown in Figure 8.

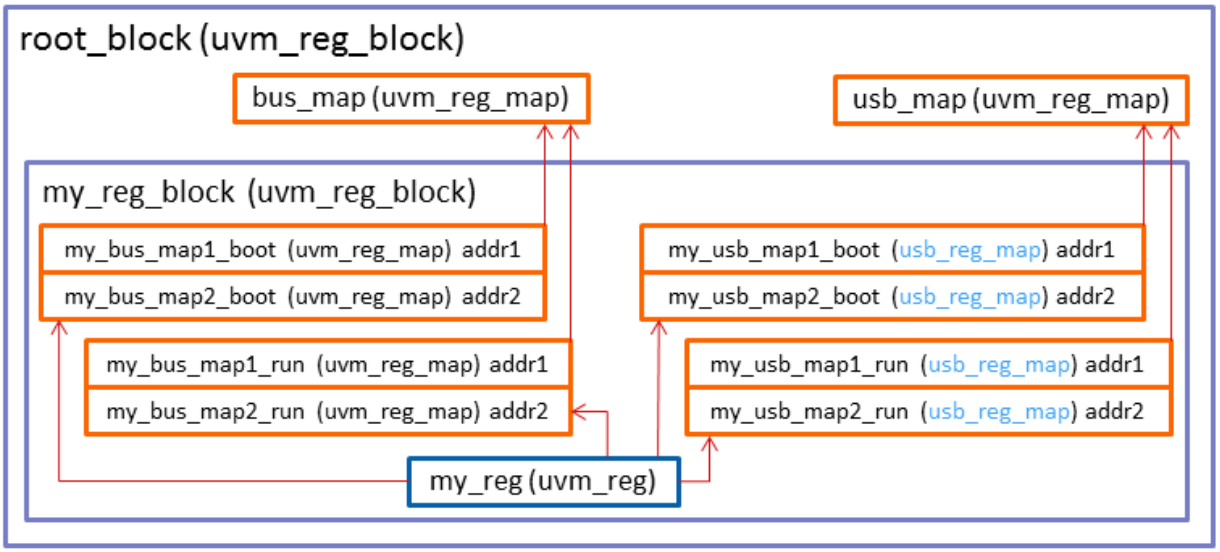

Figure 8 - Multiple Interfaces and Modes

Before the blocks, maps and registers of a register model can be used, the register model must be created, built and "locked." In UVM RAL, this is how the register blocks, maps and registers are initialized. It is typically done in the build phase with a call to the register model "lock" function. This function starts at "root_block" and recursively sets an internal "lock" bit for all the components of the register model including maps and registers. If this bit is not set successfully, critical RAL functions will not be available.

The register model lock procedure can support the two top-level maps; bus_map and usb_map. Note that there is a rule that to add a map as a submap (child) to a parent map, the block of that submap must already be a child to the block holding the parent map. In other words, a block may only hold one level of map hierarchy. Thus, there must be at least two blocks here.

The hierarchy shown in Figure 8 can support both multiple interfaces and multiple modes. It does so at the expense of adding additional submaps, creating an independent map hierarchy per interface type. Unfortunately, the configuration of Figure 8 is not acceptable. The client expects the configuration of Figure 7 and objects to the "usb" map at the top level in Figure 8.

It is okay with the client to have more than one instance of "root_block" used in simulation. This implies that the submaps for "bus" and for "usb" must be in separate instances of "root_block" so that the "usb" submap types may be overridden. Separate instances may also be required if adapter or sequencer types attached to the register model must also differ for each interface.

Further, UVM RAL requires that a register be assigned only to submaps within its own block. Thus, if we have submaps in separate block instances the register must also be duplicated in each block. As there are now multiple instances of the same register, synchronization of registers is required, as depicted in Figure 9 with "sync_regs."

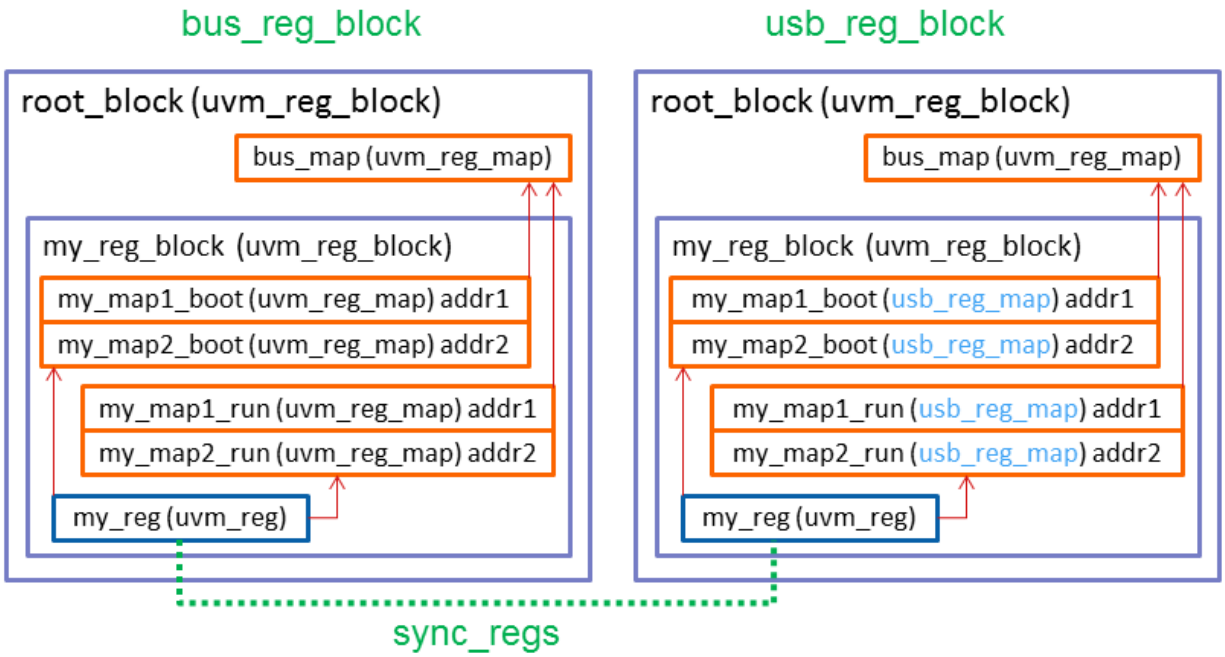

Figure 9 - Multiple root_block Instances

Though imperfect, this configuration will satisfy client requirements while supporting multiple interfaces and multiple modes in simulation. It will require special considerations to distinguish addresses and modes, to overwrite submap types and to synchronize registers.

In reality, our client also requires support for multiple instances of registers within "root_block." That is, there exist multiple copies of registers with identical offsets. This requires additional considerations, including changes to predictors, that are beyond the scope of this paper. For this paper, it is assumed that each register instance in "root_block" has unique offset.

4. Distinguishing Multiple Addresses and Modes

The key to resolving the multiple interface issue is to connect the proper top-level maps to the respective predictors. The key to resolving the multiple mode challenge is a bit more complex. This is solved by having submaps named per mode and by providing mode information to the predictor for the interface.

This requires an approach that does not use “get_local_map” and, instead, gets the list of maps within the register object and searches for a “local_map” that has both matching mode and matching address. This is done by copying and modifying the UVM internal function “write” in an extension of “uvm_reg_predictor” as shown in “bus2reg_rsp_predictor.svh” (in the code samples section).

External to the class, “check_map_mode” is defined to compare the given map against a list of potentially matching modes (see the code samples section). This is necessary so that the current mode can be referenced when finding the proper submap. Normally, variable “modes” will have only one mode listed, updated by the predictor as modes change. The string corresponding to the current mode should match the ending map name. The mode string list can be filled in with all modes to accept any mode. Alternatively, if there are no modes defined for the interface, the function can always return a true value. Of course, if no map is found there will be an error generated by UVM RAL.

5. Overwriting Submap Data Types

It is possible to address the requirement of differing submap types per interface by instructing the factory to overwrite submap types within the respective interface when the “root_block” instance is built. In this example, the “usb_map” submap types can be overwritten with a type unique to the “usb” interface as shown in the code below. Here, “reg_blk_list” is preloaded with all of the names of the submaps in the register model. This data type for the maps is required to support the predictor imported for the “usb” interface before the register model is built and locked.

```
foreach (reg_blk_list[i]) begin
  // Overwrite the USB instance address submaps with
  // usb_reg_map type (prior to building the reg model)
  // This sets some submaps to type needed for USB intfc.
  `uvm_info("BUILD_REG_MODEL_COMPONENTS",
           $sformatf(
           "Overwriting with USB reg objects for blocks: %s",
           {usb_reg_model.get_full_name(),
           ".",reg_blk_list[i],"*"}),UVM_LOW);
  uvm_reg_map::type_id::set_inst_override(
   usb_uvc_pkg::usb_reg_map#(`MY_USB_INTF_VALUES)::get_type(),
   {usb_reg_model.get_full_name(),
   ".",reg_blk_list[i],"*"});
end
```

6. Synchronizing Registers

Synchronization of the values of multiple instances of the same register is accomplished by calling a synchronizing routine (“sync_regs”) from predictors as each instance is updated. This is an external function for the predictor class and is only partially included in the code samples section. It has been optimized to create an initial hash table containing register objects of matching name to speed processing.

Specifically, the "write " function in the predictor includes the addition of the call to "sync_regs" which does the synchronization. If it detects that registers with this name have not yet been instanced in the hash, it searches all the "reg_model" instances for instances of registers with matching name. As each is found, it is added to the hash. Note that this approach takes advantage of the capability of indexing to sets of register objects by register name. Thus, when synchronization is actually done, it is done by using the passed in register object to get the name which is then used to retrieve pointers to the register objects to be updated from the hash.

7. Conclusions

Through use of multiple register modes and extensions, we are able to support our multiple interface, multiple mode environment as described in this paper. Our experience suggests at least two possible enhancements for RAL now as well as potential future directions.

Enhancement for Multiple Interfaces

Support for multiple interfaces in RAL is not trivial. However, one approach may be to allow for multiple parents of submaps only at the top level. This may enable support for multiple interfaces where submap types are shared with all interfaces. There may be a way to effectively hide multiple, top-level submaps from clients that wish only to see one submap at the top level, perhaps limiting client visibility to the default map at the top level.

Enhancement for Multiple Modes

RAL could be updated to enable distinguishing of register modes within submaps. Perhaps an efficient sideband communication approach could enable RAL to automatically track and support modes within submaps. The work may involve developing usage descriptions and to ensure that this does not have adverse impact on existing designs would not be trivial.

Further Development

Per feedback from Janick Bergeron courtesy of Synopsys, there are further considerations with respect to any such enhancements. Though this approach seems to work, there are potential hazards. Deployment cannot be recommended until such hazards are determined and resolved.

Map objects were created with the intention of eventually supporting multiple maps with register models. Though this approach does not seem to violate UVM rules, enhancements to make this easier would need careful consideration.

Adding such enhancements is not a trivial effort. Many corner cases should be considered before a solution can be generally recommended. For example:

- If two submaps with the same parent map and different offsets are assigned to a register, might the solution break some forms of register address access?
- Might there be conflicts introduced with more complicated block and map hierarchies?
- Are there functions visible to the user that might be broken by the solution?
- Are there internal UVM RAL functions that might be broken by the solution?

Proper definition and consideration of a comprehensive list of such issues may be overwhelming. Even so, successful development could be presented in detailed form to the UVM Technical Steering Committee as a contribution to the respective Mantis topic [3]. Barring that, it may be possible that with some limitations, such as limiting hierarchy depth, register model complexity and number of register access permutations, a local implementation of enhancements may prove useful.

8. References

[1] Bergeron, J., "Support for shared address maps", EDA.org Mantis 4009, January 2012 <http://www.eda.org/svdb/view.php?id=4009>

[2] Erickson, A., "Support for address submaps with multiple parents", EDA.org Mantis 4305, August, 2012 < http://www.eda.org/svdb/view.php?id=4305>

[3] Simm, U., "cant add an addressmap to another parent map despite the comment suggesting it", EDA.org Mantis 4497, March, 2013 < http://www.eda.org/svdb/view.php?id=4497>

[4] Synopsys, *UVM Register Abstraction Layer Generator User Guide*, p. 2-16, March 2014

[5] VIP Technical Committee. *Universal Verification Methodology (UVM) 1.1 Class Reference.* Accellera. June 2011 <http://www.accellera.org/downloads/standards/uvm/UVM_1.1_Class_Reference_Final_06062011.pdf>.

[6] Sherman, Steven K. "Considerations for Development and Support of Exportable UVM IP." SNUG Boston, 2013

9. Code Samples

bus2reg_rsp_predictor.svh

```
class bus2reg_rsp_predictor extends
uvm_reg_predictor#(bus_seq_item);

  ...

  uvm_reg_block blocks[$]
  protected uvm_reg sync_reg[string][$];

  ...

  extern virtual function bit check_map_mode(uvm_reg_map map);
  extern virtual protected function void sync_regs(
                 uvm_reg reg_obj, uvm_reg_bus_op rw);
  string modes[$];

  ...

  // Function: write
  // write method called by bus_in port
  virtual function void write(bus_seq_item tr);
```

```
    uvm_reg_map local_maps[$];

    ...

      // local_map = rg.get_local_map(map,"predictor::write()");

      // NOTE: The above line is commented out and this section
      // is reworked to support selection of a submap by name
      // (to support multiple modes) and by address (to support
      // multiple addresses within a mode).

      rg.get_maps(local_maps);
      ir = ($cast(ireg, rg))?ireg.get_indirect_reg():rg;

      foreach (local_maps[j]) begin
        local_map = local_maps[j];
        if(check_map_mode(local_map)) begin
          map_info = local_map.get_reg_map_info(rg);
          foreach (map_info.addr[i]) begin
            if (rw.addr == map_info.addr[i]) begin
              found = 1;
                ...
                if(reg_item.kind == UVM_WRITE) begin
                  ...
                  sync_regs(rg, rw);
                end
```

check_map_mode.svh

```
function bit bus2reg_rsp_predictor::check_map_mode(uvm_reg_map
map);
  string  map_name = map.get_name();
  int     map_name_len, mode_len;

  map_name_len = map_name.len();

  foreach (modes[i]) begin
    mode_len = modes[i].len();
    if(map_name_len > mode_len) begin
      if (map_name.substr((map_name_len - mode_len),
                          (map_name_len - 1)) == modes[i]) begin
        return 1;
      end
    end
  end
  return 0;
endfunction : check_map_mode
```

sync_regs.svh

```
//=============================================================
//
function void my_bus2reg_rsp_predictor::sync_regs(
             uvm_reg  reg_obj, uvm_reg_bus_op rw);

  string block_name;
  string reg_name;
  string block_reg_name;
  uvm_reg_block block;
  uvm_reg regs[$];
  int temp; // will ignore result returned
  ...

  if(reg_obj == null) begin
    `uvm_error("SYNC_REGS_NO_REG_OBJ",
     $sformatf("No valid register object provided!"))
  end

  `uvm_info("SYNC_REGS_START",
    $sformatf("Syncing with register \"%s\" \
            and data 0x%0x per write to address 0x%0x.",
            reg_obj.get_full_name(), rw.data, rw.addr),
            UVM_HIGH)

  block = reg_obj.get_parent();
  block_name = block.get_name();
  reg_name = reg_obj.get_name();
  block_reg_name = {block_name, reg_name};

  if(sync_reg[block_reg_name].size() == 0) begin

    foreach(reg_models[i]) begin
      ...
      // Build sync_reg for this reg_obj,
      // which always includes the original reg_obj.
      if(blocks.size() == 0) begin
        reg_models[i].get_blocks(blocks);
      end

      // Traverse blocks to update sync_reg array with
      // the new block_reg_name entry.

      foreach (blocks[i]) begin
        blocks[i].get_registers(regs);
```

```
        foreach (regs[j]) begin
          if({str_truncate(blocks[i].get_name()),
          regs[j].get_name()} == block_reg_name) begin
            sync_reg[block_reg_name].push_back(regs[j]);
          end
        end
      end
    end
  end

  if(sync_reg[block_reg_name].size() == 0) begin
    `uvm_error("SYNC_REGS_NO_REG_OBJ",
     $sformatf("No sync register matches found for \"%s\"!",
     block_reg_name)) // Must have at least the original.
  end
  else begin
    // sync registers per this reg_obj and the given data
    foreach(reg_models[i]) begin
    ...

      foreach (sync_reg[block_reg_name][i]) begin
        // Do not attempt to update the source register to avoid
        // "Trying to predict value of register ... while it is
        // being accessed" warning.
        if(sync_reg[block_reg_name][i] != reg_obj) begin
          temp = sync_reg[block_reg_name][i].predict(rw.data);
          `uvm_info("SYNC_REGS_SYNCED_REG",
            $sformatf("Synced register \"%s\" with data 0x%0x.",
            sync_reg[block_reg_name][i].get_full_name(),
            rw.data),
            UVM_HIGH)
        end
      end
    end
  end

endfunction : sync_regs
```