UVM Register Modelling at the Integration-Level Testbench

Wayne Yun
Advanced Micro Devices, Inc.
1 Commerce Valley Drive East
Markham, Ontario, Canada L3T 7X6

***Abstract*-When design IPs and blocks are integrated together, their configuration registers are connected by decoding logic or register control blocks so as to compose larger address space(s). Similarly in a testbench, IP register models are stitched together to facilitate verification and reuse at the integration level. UVM [1] provides fundamental elements for register modelling. It defines SystemVerilog classes [2] for registers, fields, blocks, maps, etc. They can be extended and model many simple designs, especially IPs and design blocks. Address space composition is expressed by a parent-child relationship of maps in UVM; however, UVM has several restrictions. For example, multiple parents are not allowed. A few common types of register space composition are not directly supported. This paper describes solutions using user defined maps and register for register alias, duplication, and indirection. Some general consideration of an integration-level UVM register model is also presented.**

I. INTRODUCTION

Universal Verification Methodology (UVM) standardizes architecture and elements of simulation testbenches. Various reuse strategies leverage such standardization to improve quality of verification, shorten project cycle, and reduce overall cost. One of them is vertical reuse in which a higher level testbench reproduces the work of a lower level. It starts from reusing the lowest IP-level testbench at the first integration level. The UVM register model is an important part of a testbench to be reused.

A UVM register model is the verification counterpart of configuration registers inside a design. Typically, design IP organizes its configuration registers in a single address space. At the integration level, decoding logic or a register control block composes one or many address spaces from those of multiple IPs. The simplest composition is concatenation (i.e., a single larger address space is divided into many non-overlapping segments) and each IP occupies one segment. Design specifications often command more than the simplest method. The base address of one IP could be dynamically decided by a configuration register. IP address space could be accessed from multiple address segments from the same address space, as "(b0) alias" shows in Figure 1, or from another address space, as "(c) duplication" shows in Figure 1. Even a composed address space could be accessed from an indirect data/index register pair located in another address space, as "(a) indirect" shows in Figure 1.

This paper presents techniques for modelling integration-level configuration registers, which address the above design requirements based on UVM register model.

A. UVM Register Model

UVM supplies classes of register field, register, register block, register map, etc. They can be extended, configured, and assembled together to model design configuration registers. Detailed documents can be found in [1].

A UVM register map models a segment of, or a complete address space. A lower level register map is added to a higher level register map as a child. Each child occupies a part of its parent's address space. A register map having no parent is the root or the top map. It represents a complete address space.

One UVM register can be added to multiple register maps as long as the register maps and the register are instantiated in the same register block. As such, the same register can be mapped into multiple address spaces.

At the integration level, composition of address space is a parent-child relationship between the integration-level register map and lower level ones. Any parent can have one or many children; however, current UVM

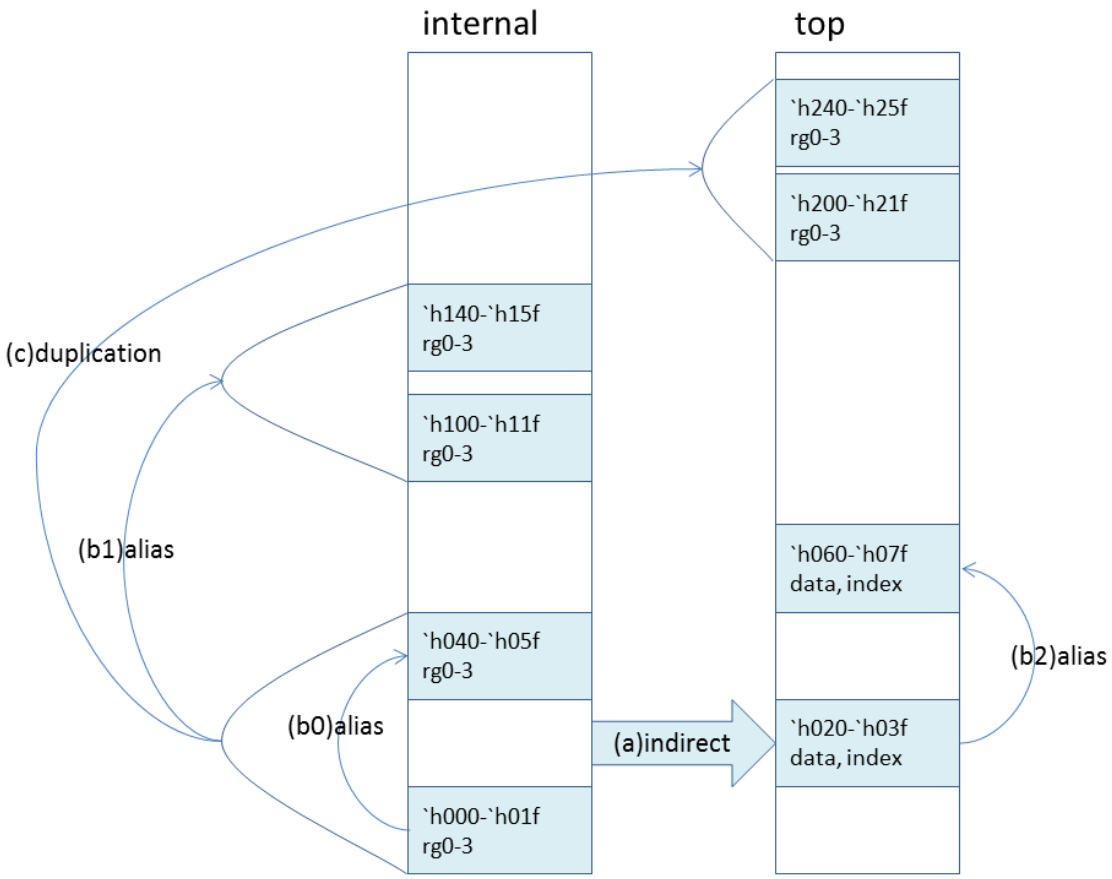

Figure 1 Examples of Address Space Composition

implementation only allows a child to have one parent. The creators of UVM register model obviously noticed the necessity of multiple parents, and some initial work has already been completed.

The rule of the same register block for the register map and the register also has consequences at the integration level. For example, an IP level defines only one address space, but two are required at the integration level. IP registers are not allowed to be added to a register map of an integration level register block. The only legitimate solution is to update the IP-level register model to have more address spaces. For practical reasons, it is often not a simple task.

B. Requirements of Integrating Register Models

A higher level register model must provide the same programming interface as the lower level for every IP and the intermediate integration levels to facilitate reuse of UVM verification components and sequences. This applies to complicated address space compositions just as it does to the simplest one. The most important guidelines for the integration-level register model are listed below.

1) Name, properties, and structure of register fields, registers, register blocks, and register maps from lower levels should be kept the same

2) Prototypes and functionality of tasks, functions, and class member variables of lower levels should be kept the same

3) Features provided at a lower level should be kept the same

4) Any feature introduced at a higher level should be transparent to the lower level

Object Oriented Programming capability provided by SystemVerilog classes can satisfy these reuse requirements while implementing various compositions defined by a design.

II. Structure of Integration-Level Register Model

Design integration existed long before UVM. So did integration-level register models. UVM emphasizes reuse to meet productivity challenges, but the focus could have been quite different in earlier days. The previous structure of integration-level register models may not satisfy UVM reuse.

Even for newly created pure UVM testbenches, guidelines are still needed in terms of how an integration-level testbench organizes pieces from lower levels.

When any of previously stated reuse requirements are broken, the IP sequence has to be updated before being reused at the integration-level. A higher workload incurs when such a structure is chosen. Some examples are examined below.

A. Flattened Structure

When the integration-level register model follows a flattened structure, all registers from IPs are instantiated in one integration-level register block. New and longer names are given to each register to avoid any potential conflict. Pseudo code in Figure 2 demonstrates this structure.

```
// IP level register model
class ip_register_block extends uvm_reg_block;
  ip_register regA;
endclass

// IP level sequence
class ip_sequence extends uvm_sequence;
  ip_register_block ip_registers;

  task body();
    ...
    ip_registers.regA.write();
    ...
  endtask
endclass

// Flattened structure of integration level register model
// It cannot be used by ip_sequnece
class integration_level_register_block extends uvm_reg_block;
  // registers from first instance of IP
  ip_register ip_0_regA;
  // registers from second instance of IP
  ip_register ip_1_regA;
endclass
```

Figure 2 Flattened Structure of an Integration-Level Register Model

Flattened structure results in a register model that is incompatible with the lower level one. The lower level sequence cannot be reused. UVM integration-level testbenches should not choose this structure.

B. Repackage Structure

The integration-level register model could instantiate lower level register sub-blocks. This is a repackaging of the lower level register models. Pseudo code in Figure 3 demonstrates this structure.

```
// IP level register model
class ip_register_subblock extends uvm_reg_block;
  ip_register regA;
endclass

class ip_register_block extends uvm_reg_block;
  ip_register_subblock sub0;
endclass

// IP level sequence
class ip_sequence extends uvm_sequence;
  ip_register_block ip_registers;

  task body();
    ...
    ip_registers.sub0.regA.write();
    ...
  endtask
endclass

// Repackage structure of integration level register model
// It cannot be used by ip_sequnece
class integration_level_register_block extends uvm_reg_block;
  // registers from sub-blocks of first instance of IP
  ip_register_subblock ip_0_sub0;
```

```
    // registers from sub-blocks of second instance of IP
    ip_register_subblock ip_1_sub0;
  endclass
```

Figure 3 Repackage Structure of an Integration-Level Register Model

The repackaged structure results in a register model that is incompatible with the lower level one. The lower level sequence cannot be reused. UVM integration-level testbenches should not choose this structure.

C. Hierarchical Structure

When every lower level register model is instantiated directly in an integration-level register model, a hierarchical structure is created. Any lower level register model is a branch in this hierarchy. Pseudo code in Figure 4 demonstrates this structure.

```
// IP level register model
class ip_register_subblock extends uvm_reg_block;
  ip_register regA;
endclass

class ip_register_block extends uvm_reg_block;
  ip_register_subblock sub0;
endclass

// IP level sequence
class ip_sequence extends uvm_sequence;
  ip_register_block ip_registers;

  task body();
    ...
    ip_registers.sub0.regA.write();
    ...
  endtask
endclass

// Hierarchical structure of integration level register model
// It can be used by ip_sequnece
class integration_level_register_block extends uvm_reg_block;
  // registers from first instance of IP
  ip_register_block ip_0;
  // registers from second instance of IP
  ip_register_block ip_1;
endclass

// Integration level sequence
class integration_level_sequence extends uvm_sequence;
  integration_level_register_block all_reg_blocks;
  ip_sequence ip_action = new();

  task body();
    ...
    ip_action.ip_registers = all_reg_blocks.ip_0;
    ip_action.start();
    ...
  endtask
endclass
```

Figure 4 Hierarchical Structure of an Integration-Level Register Model

A hierarchical structure results in a register model that is compatible with the lower level one. The lower level sequence is able to be reused. The hierarchical structure can be applied at every level of integration recursively. UVM integration-level testbenches should choose this structure.

There could be more types of structures that can satisfy reuse requirements at the integration-level testbench. For practical reasons, one simple solution is sufficient.

III. Dynamic Base Address

A design could dictate that the offset of one configuration register or base address of a group be changed by certain settings. This feature could be created at any level. Reference [1] did not document how this feature should be supported. But UVM code actually provides everything needed.

A. Changing Offset of One Configuration Register

Function uvm_reg_map::m_set_reg_offset() provides such capacity. It has three arguments. The first is the register of which the offset would be changed. The second is the new offset. The third flags if this register should be unmapped.

This function provides offset control at a granularity level of an individual register. Therefore any grouping of registers can be implemented.

B. Changing Base Address of a Register Map

Function uvm_reg_map::set_base_addr() provides such capacity. It has only one argument which is the new offset for the register map.

This function provides offset control at a granularity level of a register map. Therefore, all of the registers added to the map will change their address simultaneously. However, if the design functions exactly in the same way, it is a simpler implementation.

IV. Indirect Registers

In Figure 1, "(a) indirect" shows that "internal" address space is accessed via a data/index register pair located in "top" address space. All registers located in "internal" address space are indirect registers. UVM contains an implementation of indirect data registers. It serves a certain design pattern, but at the integration-level testbench, it has a few difficulties.

First, it requires indirect registers to be in same register map as the indirect data/index pair. If the design requires indirect registers in one IP (one register map), and an indirect data/index pair in another IP (another register map), UVM indirect data register does not support this.

Second, it requires the offsets of indirect registers to be consecutive and start from 0. Again, the design could just designate scattered offsets.

Third, it is inconvenient to add more data/index pairs. Every data/index pair needs a dedicated register map. If an IP register model does not provide any extra register map or data/index pair, it is not simple to add one at the integration level.

Integration-level testbenches need a more generic solution for indirect registers. One obvious solution is to define a user register map implementing indirect access using address information provided by the "internal" address map. This user register map is named as proxy_map.

Figure 5 is pseudo code for proxy_map and usage. Figure 6 explains how proxy_map works. Proxy_map is a local map of indirect registers. When proxy_map is specified as the map to be used in a register access, this access request will be passed to proxy_map. Then proxy_map calculates the index from the register address in indirect space, and initiates accesses to the data/index pair.

```
// Proxy_map implements generic indirect register access.
//   It overcomes restrictions of class uvm_reg_indirect_data.
//     a) Indirect registers have to be in the same map of data/index pair
//     b) Offsets of indirect registers have to be consecutive and start from 0
//     c) Inconvenient to addd more data/index pairs
//
// Since proxy_map extends uvm_reg_map, it can be used as a uvm_reg_map.

class proxy_map extends uvm_reg_map;
  // the map to access data and index registers
  uvm_reg_map data_index_map;

  // the map in which all indirect registers are added
  uvm_reg_map indirect_space_map;

  // the data register of the indirect pair
  uvm_reg data_reg;

  // the index register of the indirect pair
  uvm_reg index_reg;

  // the sequence requesting data/index pair
  uvm_sequence indirect_access_seq;
```

```
  task do_write;
    index_reg.write(rg.get_address());
    data_reg. write(data[0]);
  endtask

    ...
  endclass

  // IP level register model
  class ip_register_block extends uvm_reg_block;
    // IP level registers
    ip_register regA;

    // IP level register map
    uvm_reg_map ip_map;

    ...
  endclass

  // Integration-level register model
  class integration_level_block extends uvm_reg_block;
    // Instantiate IP level block
    ip_register_block ip_0;

    // Top block has data and index register
    Indirect_pair_block top_block;

    // Internal address space
    uvm_reg_map internal_map;

    // Top address space
    uvm_reg_map top_map;

    // Proxy_map creating indirect address space
    proxy_map indirect_map;

    // Build function of register block
    function void build();
      ...
      // data_index_map of proxy_map is top
      indirect_map.data_index_map = top_map;

      // data_reg of proxy_map is in top_block
      indirect_map.data_reg = top_block.data_reg;

      // index_reg of proxy_map is in top_block
      indirect_map.index_reg = top_block.index_reg;

      // indirect_space_map of proxy_map is internal_map
      indirect_map.indirect_space_map = internal_map;

      ...
    endfunction
    ...
  endclass

  // Integration-level sequence
  class integration_level_sequence extends uvm_sequence;
    // Integration-level register block
    Integration_level_block all_registers;

    task body();
      ...
      // Indirect write to register
      all_registers.ip_0.regA.write(..., .map(all_registers.indirect_map));
      ...
    endtask
  endclass
```

Figure 5 Pseudo Code for Proxy_map and Usage

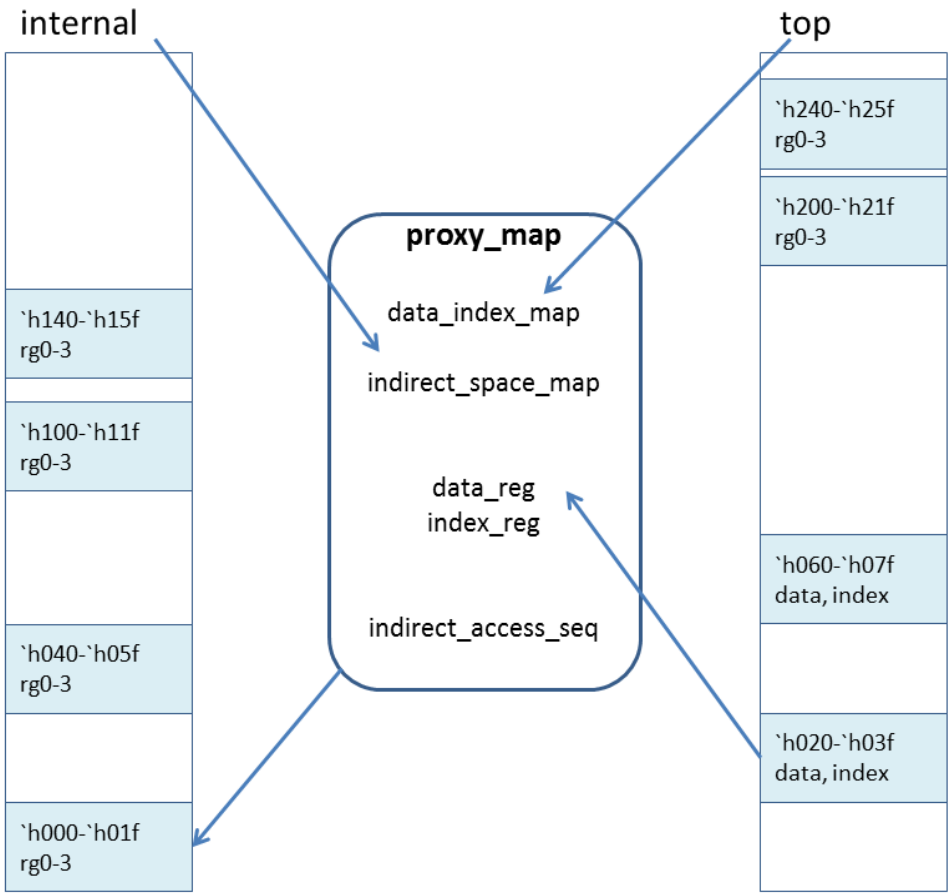

Figure 6 Proxy_map

Since the data/index pair is separate from the indirect registers, they can be in different register maps. Therefore, the first difficulty above is resolved. There is no restriction to the offsets of indirect registers, because the register map supports any arbitrary offset. The second difficulty is resolved too. Finally, adding a new data/index pair only needs a new proxy_map. The third difficulty is resolved also.

The proxy_map is created at the integration level. No change to the lower level register model is needed. The proxy_map satisfies all requirements listed in section I.B.

V. Alias Registers

At the integration level, a design could specify a segment of configuration registers that can be accessed from two or more address ranges in same address space. Figure 1 shows 3 examples – (b0), (b1), and (b2). If one address range is considered to be main, then the others are aliases. Decoding logic of the design is rather simple for aliasing; however, its counterpart of the UVM register model is not straight forward. Reference [1] states alias registers should be declared as independent registers, and callbacks synchronize values. It works but requires manual work and consumes simulation resources.

The complexity of alias registers comes from the UVM register map. The UVM register model expresses address space composition by parent-child relationships between register maps. From a topology point of view, an alias can be expressed by multiple parents of a lower level register map. Inline comments of UVM source code states clearly that multiple parents is not supported. Current projects need a gap filling solution before multiple parents are supported by UVM register map.

Replica_map is the name of the solution presented by this paper. It is a user defined register map, shown in Figure 7. It is a local map of IP registers. It has a handle to the IP level register map which represents IP address space. It has a function offset_cal, which calculates an alias offset from the IP address space offset and vice versa. It also has dummy registers which occupy an alias address range when replica_map is added to the integration level map.

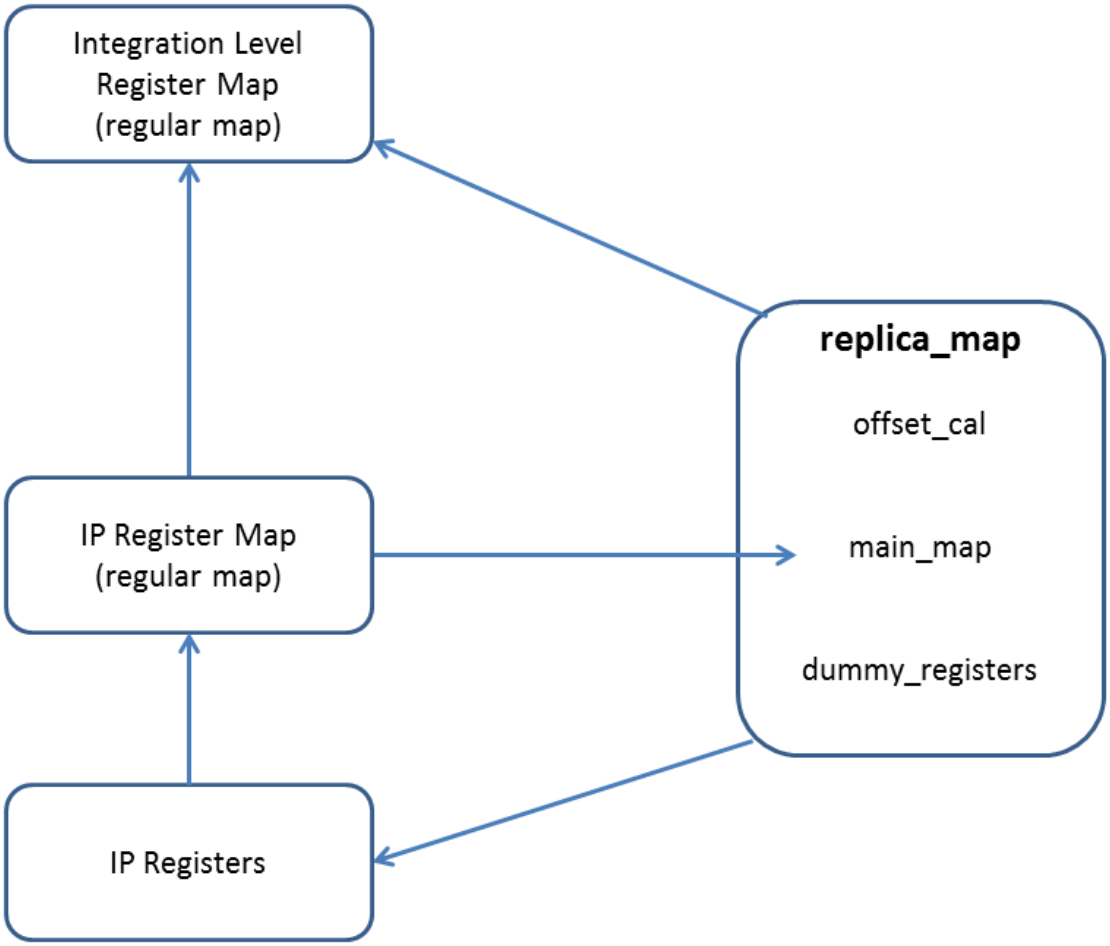

Figure 7 Replica_map

When access to an IP register is specified to use a replica_map, the access request will be passed to the replica_map. Then the offset of the register in alias address range is calculated from its offset in the IP register map, and a new request is sent through the root map – Integration-Level Register Map in the figure. Figure 8 shows the pseudo code for replica_map and its aliasing usage.

```
// Replica_map enables a register to be accessed from another address.
// Following pseudo code demonstrates aliasing usage.
//
// Since replica_map extends uvm_reg_map, it can be used as a uvm_reg_map.

class replica_map extends uvm_reg_map;
  // the map to be aliased
  uvm_reg_map main_map;

  // calculates address in target map
  function uvm_reg_addr_t offset_cal();

  task do_write;
    parent_map_write(.addr(offset_cal()), .data(data[0]));
  endtask

  ...

endclass
```

```
// IP level register model
class ip_register_block extends uvm_reg_block;
  // IP level registers
  ip_register regA;

  // IP level register map
  uvm_reg_map ip_map;

  ...
endclass

// Integration-level register model
class integration_level_block extends uvm_reg_block;
  // Instantiate IP level block
  ip_register_block ip_0;

  // Integration-level address space
  uvm_reg_map internal_map;

  // replica_map map registers to another base address
  replica_map alias_map;

  ...

  // Build function of register block
  function void build();
    ...
    // data_index_map of proxy_map is top
    alias_map.main_map = ip_0.ip_map;

    // internal_map has child maps
    internal_map.add_submap(ip_0.ip_map);
    internal_map.add_submap(alias_map);

    ...
  endfunction

  ...
endclass

// Integration-level sequence
class integration_level_sequence extends uvm_sequence;
  // Integration-level register block
  Integration_level_block all_registers;

  task body();
    ...
    // write to alias address
    all_registers.ip_0.regA.write(..., .map(all_registers.alias_map));
    ...
  endtask
endclass
```

Figure 8 Pseudo Code for Replica_map and Aliasing Usage

Replica_map is added at the integration level. If there are more alias address ranges, simply define one replica_map per address range. No change is made to the IP-level register model. This is important when the register model comes from third party.

Replica_map maintains the UVM register usage model, and satisfies requirements listed in section I.B.

VI. Duplicated Registers in Another Address Space

There could be more than one address space at the integration level, and the design may specify to access the same IP configuration register from all address spaces. Figure 1, "(c) duplication" shows this topology. Standard implementation of the UVM register model is to create one set of register maps for every integration level address space at the IP level, and add registers to each set of maps. If the IP-level register model does not have enough sets of register maps for all integration-level address spaces, it is not straight forward to add at the integration level.

If no additional set of register maps is created, the topology is another form of multiple parents for the UVM register model. As discussed before, the UVM register model does not support it. Replica_map can enable this usage.

In Figure 9, there are two integration-level address spaces, represented by Integration-Level Register Map A and B. The IP only implements one address space, represented by IP Register Map. Replica_map is a local map of IP registers, and also a child map of Integration-Level Register Map B. Such replica_map duplicates the IP registers to another address space.

When an access request to IP registers is specified to use a replica_map, the request will be passed to the replica_map. Replica_map calculates the offset of the register in Integration-Level Register Map B from its main_map, which is actually IP Register Map. Effectively, IP registers are duplicated to another address space.

Replica_map is added at the integration level. If there are more address spaces, simply create one for each. No change is required to the IP register model. This is important when the IP register model is difficult to change.

The usage model of replica_map when duplicating registers to another address space follows the original UVM register model. All requirements listed in I.B are satisfied.

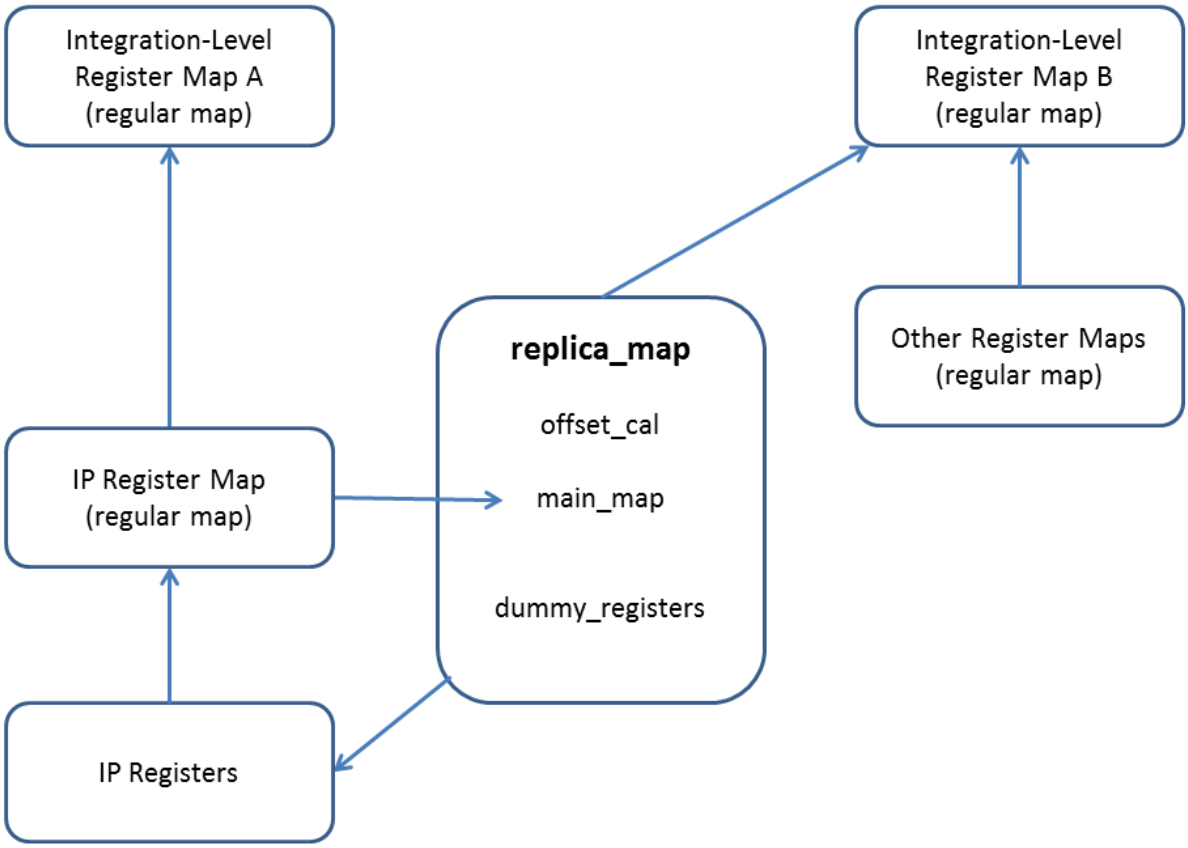

Figure 9 Duplicating Registers Using Replica_map

Figure 10 shows the pseudo code for replica_map and its duplicating usage.

```
// Replica_map enables a register to be accessed from another address.
// Following pseudo code demonstrates duplicating usage.
//
// Since replica_map extends uvm_reg_map, it can be used as a uvm_reg_map.

class replica_map extends uvm_reg_map;
  // the map to be aliased
  uvm_reg_map main_map;

  // calculates address in target map
  function uvm_reg_addr_t offset_cal();
```

```
    task do_write;
      parent_map_write(.addr(offset_cal()), .data(data[0]));
    endtask

    ...

  endclass

  // IP level register model
  class ip_register_block extends uvm_reg_block;
    // IP level registers
    ip_register regA;

    // IP level register map
    uvm_reg_map ip_map;

    ...
  endclass

  // Integration-level register model
  class integration_level_block extends uvm_reg_block;
    // Instantiate IP level block
    ip_register_block ip_0;

    // Integration-level address map A
    uvm_reg_map internal_map;

    // Integration-level address map B
    uvm_reg_map top_map;

    // replica_map map registers to another base address
    replica_map dup_map;

    ...

    // Build function of register block
    function void build();
      ...
      // data_index_map of proxy_map is top
      alias_map.main_map = ip_0.ip_map;

      // internal_map has child map
      internal_map.add_submap(ip_0.ip_map);

      // top_map has child map
      top_map.add_submap(dup_map);

      ...
    endfunction

    ...
  endclass

  // Integration-level sequence
  class integration_level_sequence extends uvm_sequence;
    // Integration-level register block
    Integration_level_block all_registers;

    task body();
      ...
      // write to address in top_map
      all_registers.ip_0.regA.write(..., .map(all_registers.dup_map));
      ...
    endtask
  endclass
```

Figure 10 Pseudo Code for Replica_map and Duplicating Usage

VII. SUMMARY

The Integration-level UVM register model should use IP-level register models as sub-register-blocks. User defined register maps, namely proxy_map and replica_map, can provide modelling capability of 3 common types of register address space composition – alias, duplication, and indirection. Difficulties from UVM register model are overcome. The same programming interface is kept for test sequences and testbench components. No change to the UVM package is required. The IP-level register model does not need to be changed either. All reuse requirements from section I.B are satisfied. This described structure can be repeated recursively at higher level testbenches.

Using proxy_map and replica_map, register address space composition of Figure 1 can be solved as shown in Figure 11. Proxy_map is used for indirection. Replica_map is used for alias and duplication. This example was implemented to demonstrate usage of proxy_map and replica_map. Testing of them should be carried out with real project scenarios. Since they are part of a testbench, they could be checked by Synopsys Certitude as well.

All example code was tested with Synopsys VCS 2014.12 and UVM 1.2.

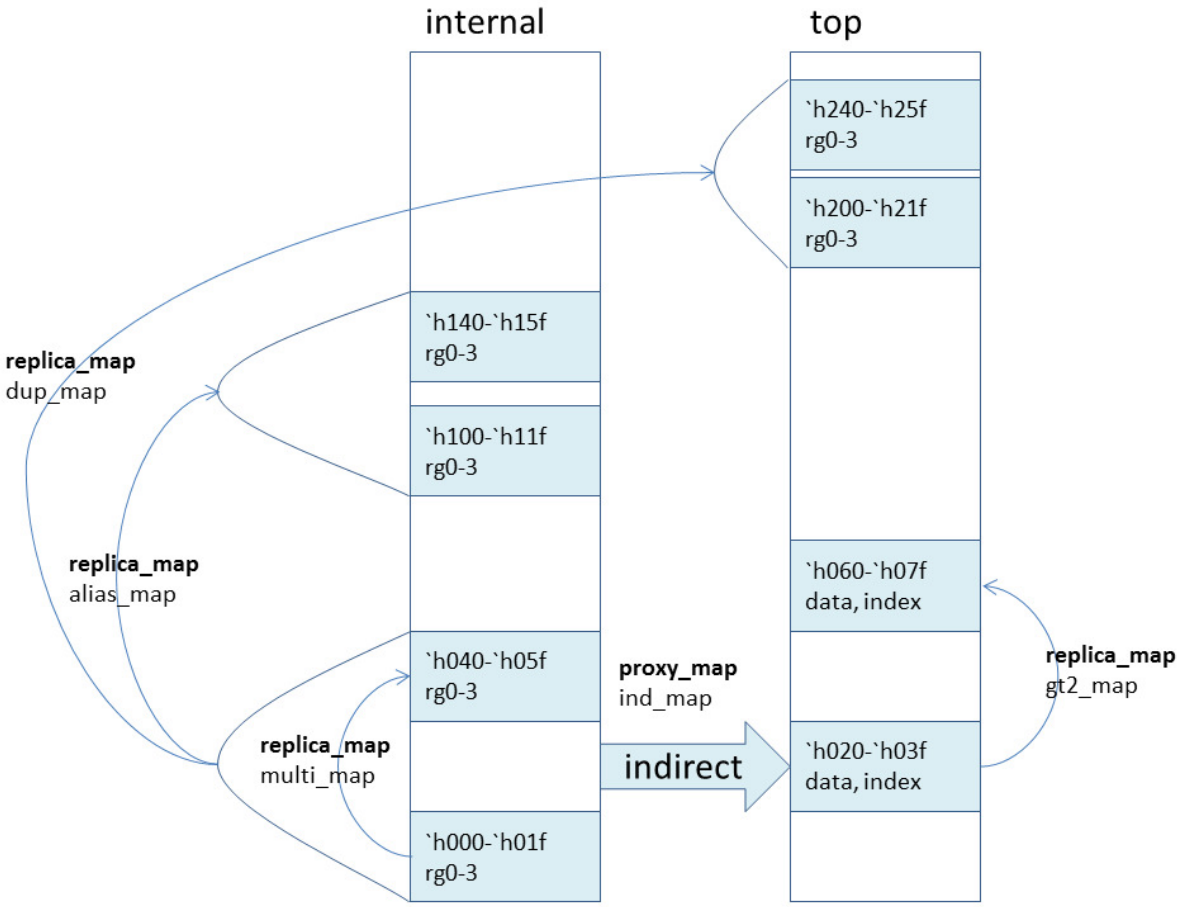

Figure 11 Integration-Level Address Space Compositions Using Proxy_map and Replica_map

ACKNOWLEDGMENT

The author would like to thank Karl Whiting, Juan Zhou, Hori Hu, Theta Yang, and David Chen for their help in understanding integration-level testbench requirements. Gord Caruk and other industry experts reviewed this paper, it is much appreciated.

REFERENCES

[1] Accellera, UVM User Guide, v1.1, www.uvmworld.org

[2] Accellera, UVM Reference Guide, v1.1d, www.uvmworld.org