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

Steven K. Sherman

AMD

September 11, 2014

SNUG Boston

Agenda

Register Access Challenges

Example of Two Interfaces

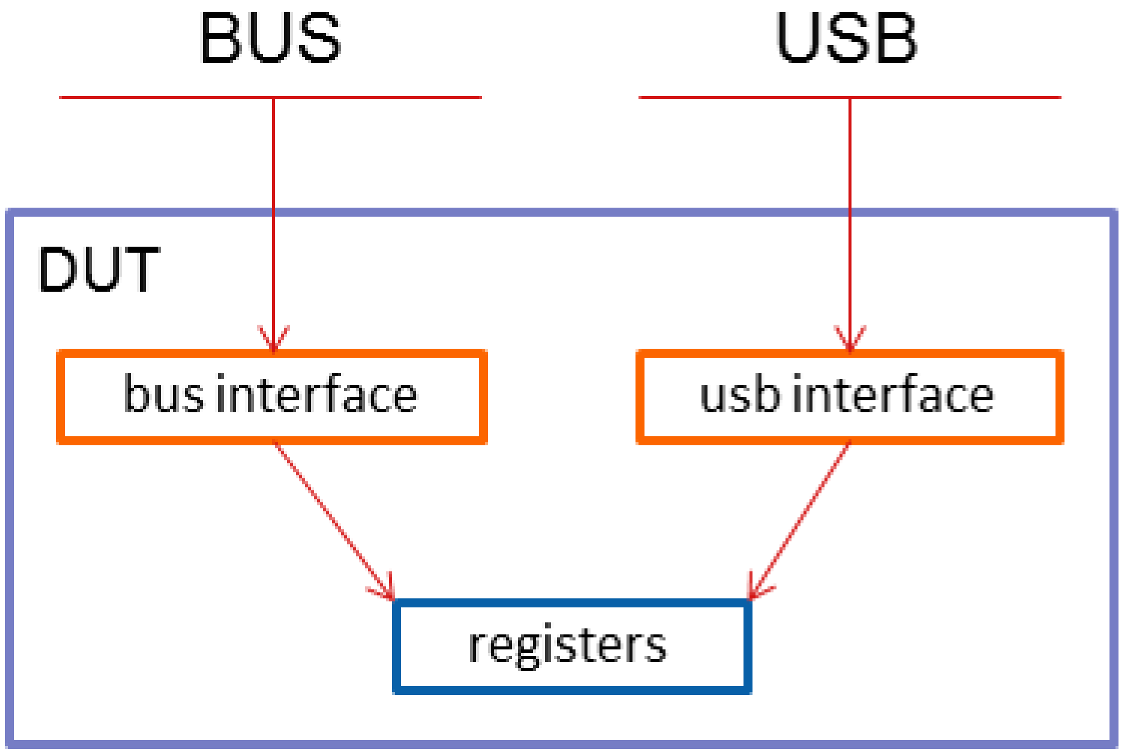


Figure 1

Multi-Interface Challenge

Legacy RAL

- VMM RAL
- “domains” for interfaces
- “block” contains multiple domains and registers
- more than one physical interface supported (good!)

UVM RAL

- UVM 1.1d, UVM 1.2
- “submap” (sort of)
- “block” contains submaps and registers
- only one parent map supported per submap (BAD!)

Multi-Interface Challenge

Policies

- Enhance in predictors (not seqs or drivers) to export passive IP to clients
- Use provided IP pre-processing
- Simplify pre-processing/processing for our clients
- Use provided RAL code generators “as is”

Multi-Mode Challenge

- With a submap you can
  - set access rights and part of register address
  - distinguish between modes:
    - RW for “configure” mode
    - RO for “run” mode
  - flip between modes dynamically (such as “warm” reset)
- If same address with multiple submaps:
  - access may vary by mode (good!)
  - may support many modes (good!)
  - predictor “get_local_map” may return wrong map (BAD!)

NOTE: Same address, multiple submaps generates ugly RAL warning.

Wait! What is RAL?

- RAL provides common functions and features for register access.
- RAL structures fit inside dual hierarchies for:
  - Register instances (organized via blocks)
  - Register access (organized via maps)
- RAL transaction flows include support for:
  - Requests
  - Acks
  - Read/Write Responses

Dual Hierarchies

Register Block Hierarchy

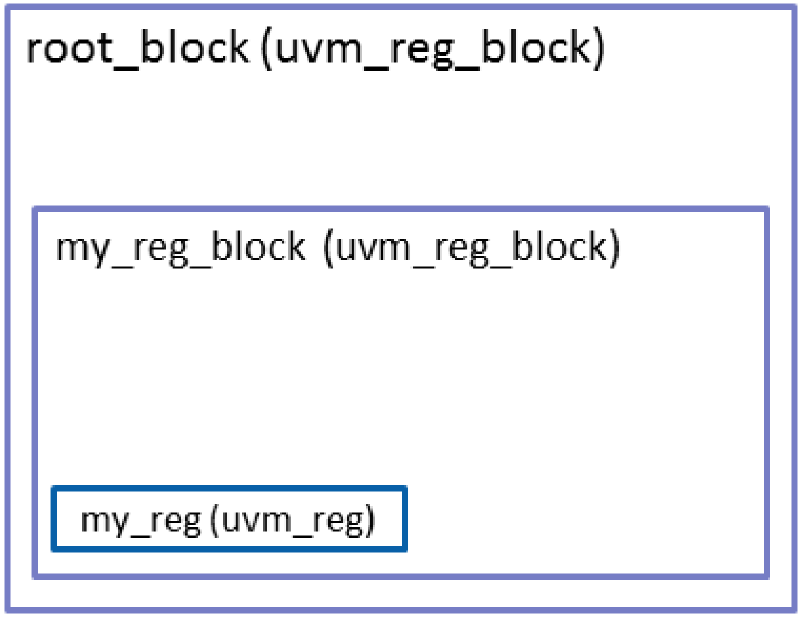


Figure 2

Register Map Hierarchy

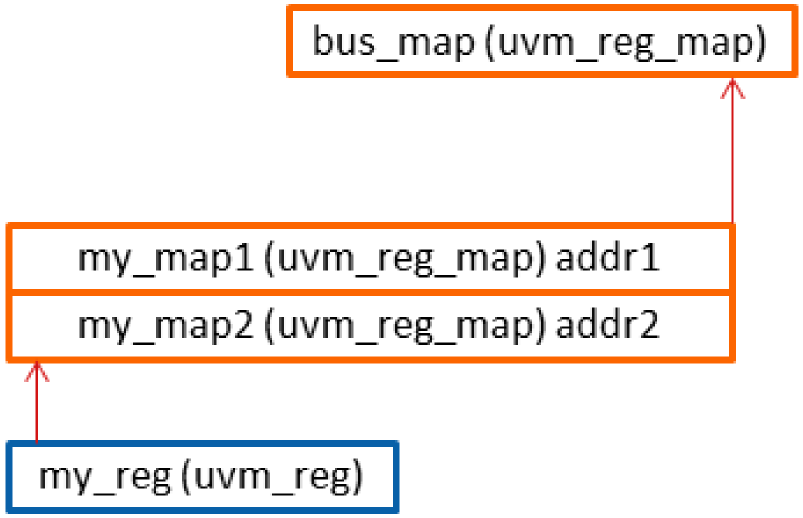
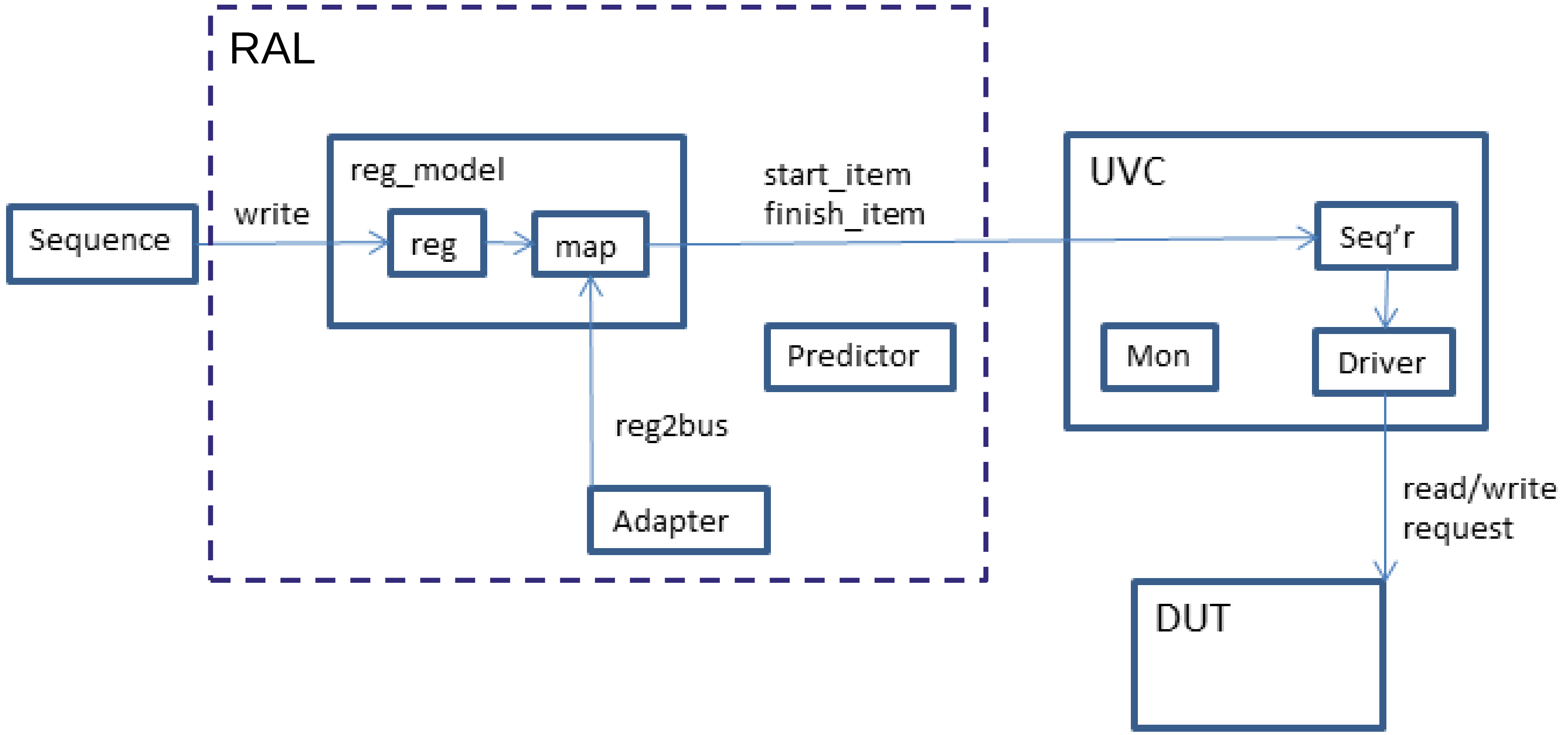


Figure 3

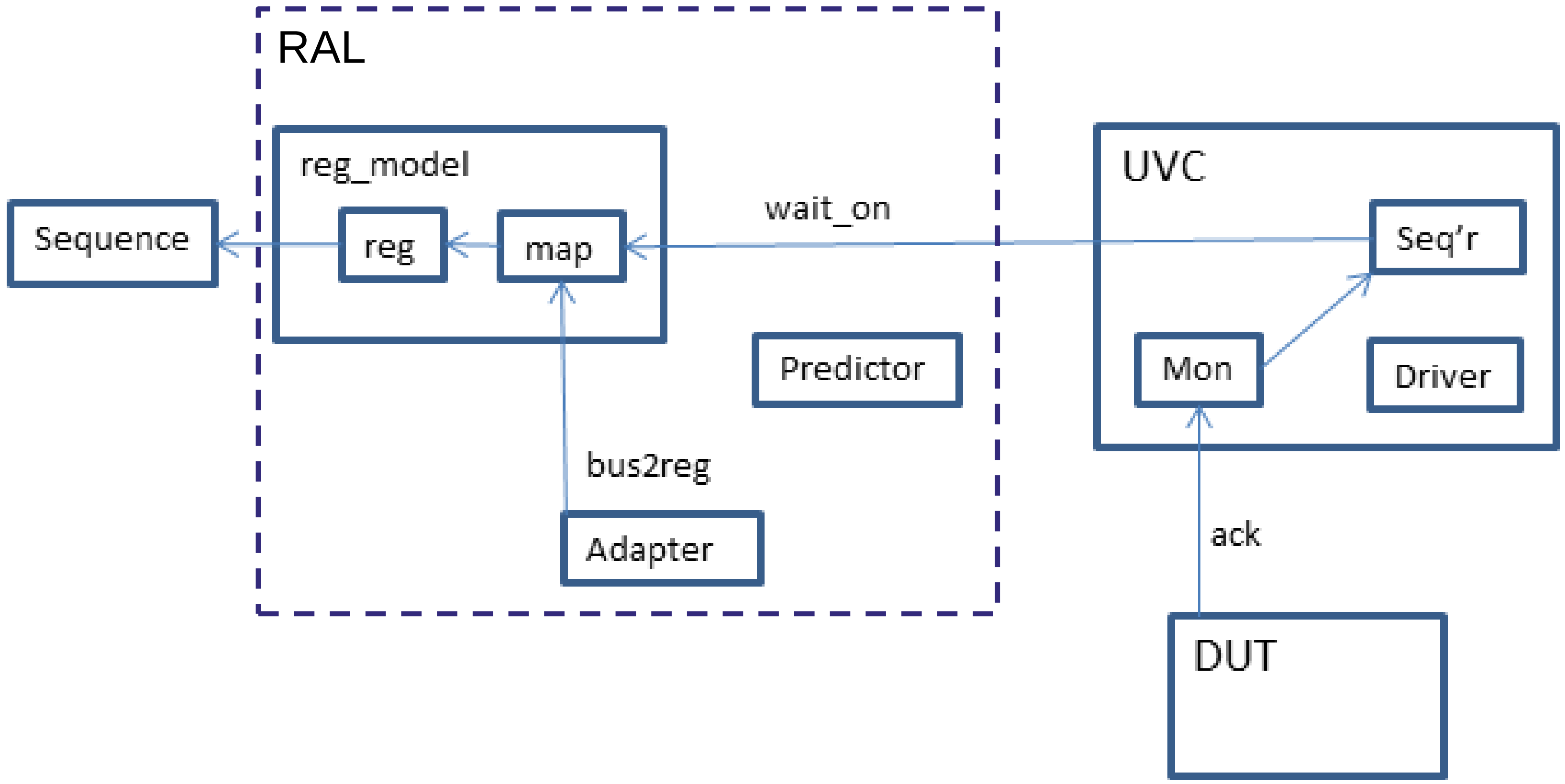
RAL Transaction Flows

Requests



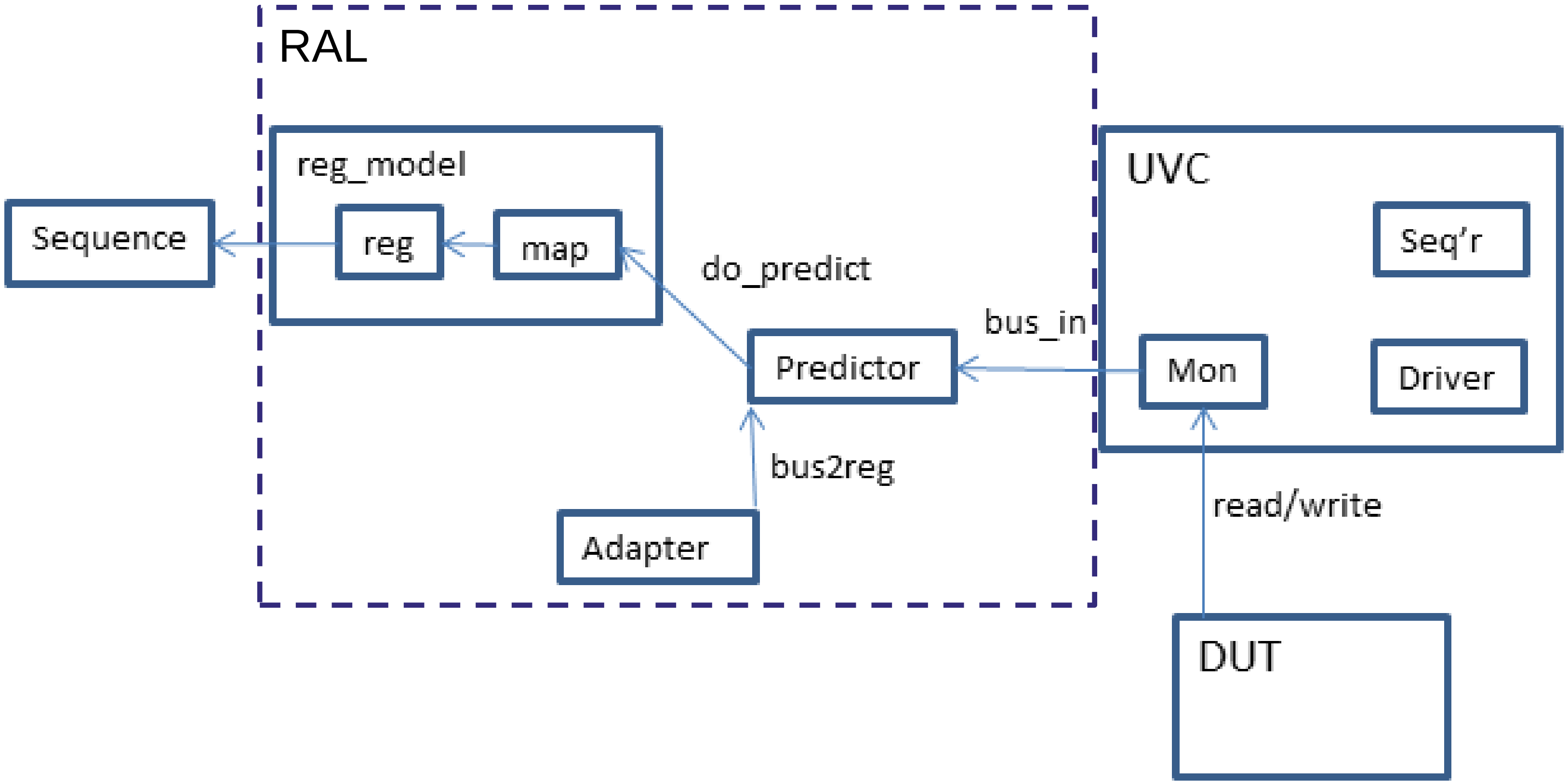
RAL Transaction Flows

Acks

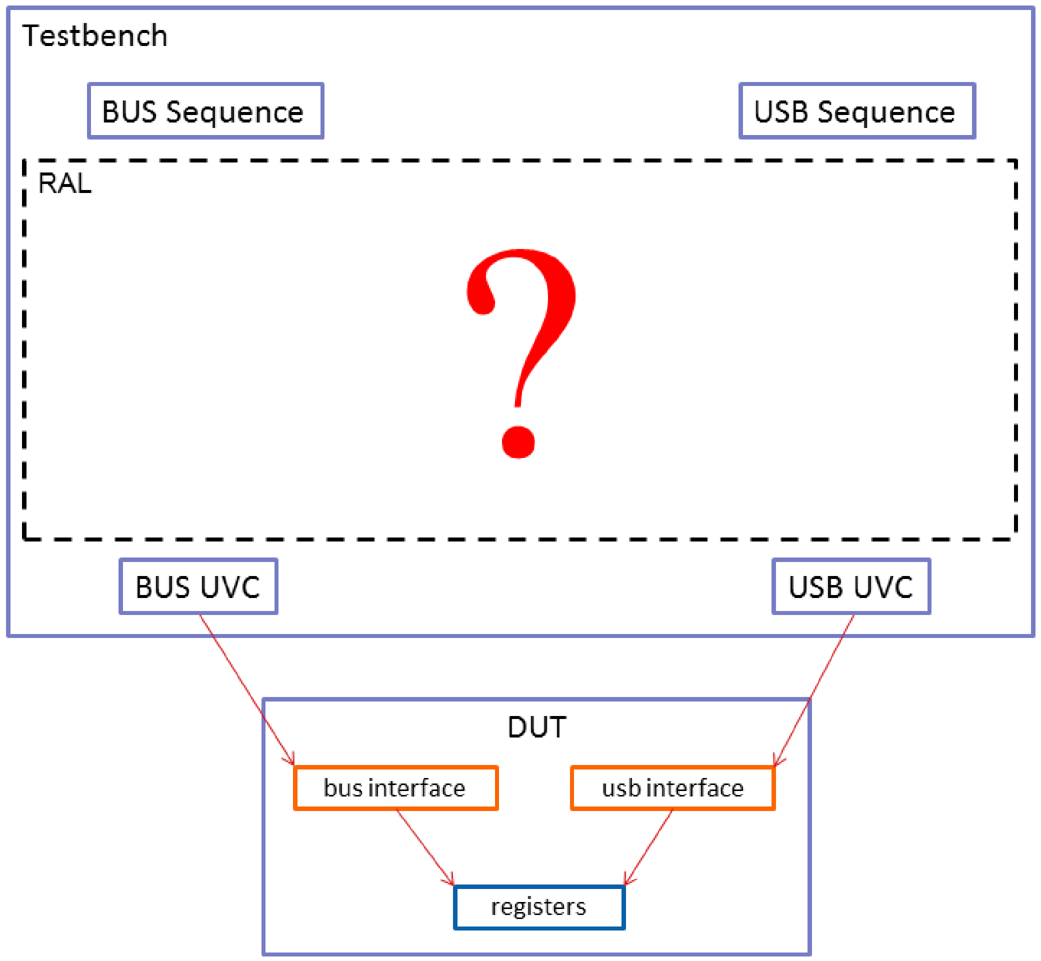


RAL Transaction Flows

Read/Write Responses



The Example Problem



Interfaces and Modes

Single Interface & Mode

- A register with single interface:
  - may only be accessed via one port
  - may be accessed via multiple addresses (with submaps)
  - has only one mode of operation

Single Interface

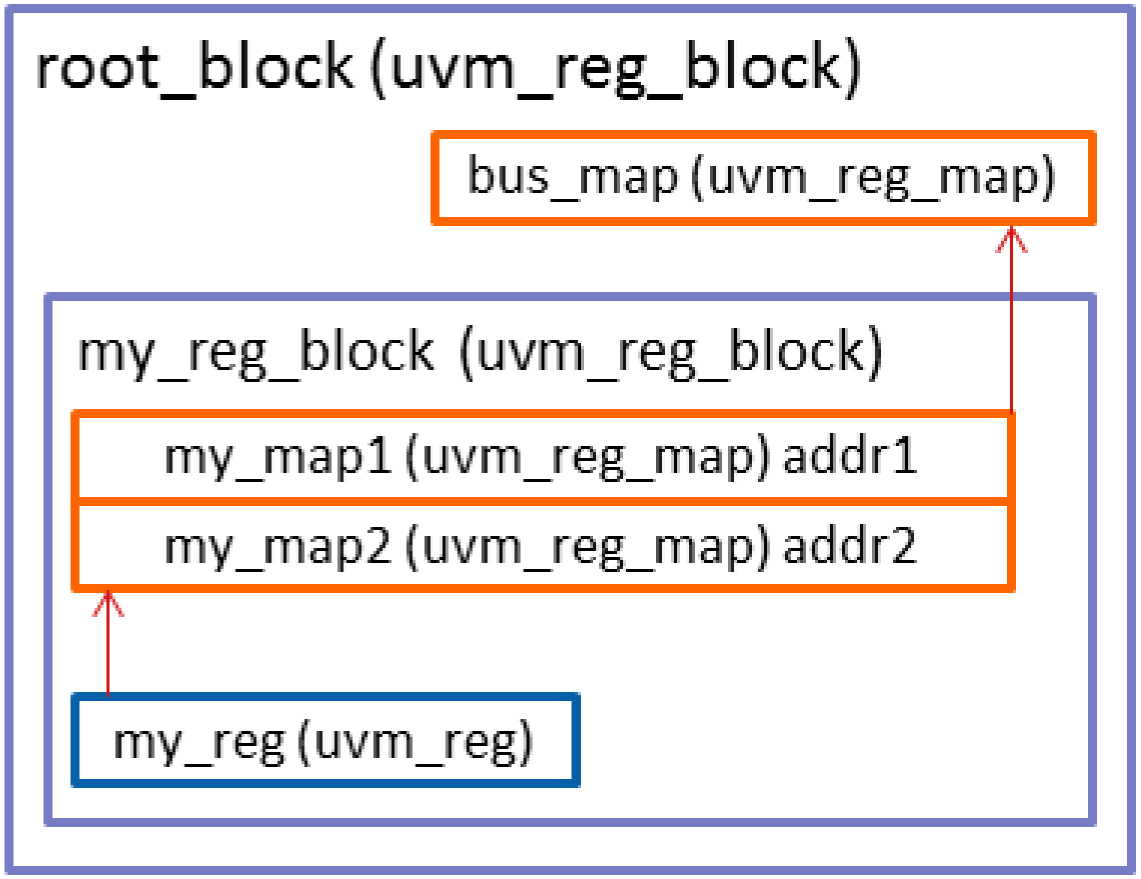

root_block (uvm_reg_block)
bus_map (uvm_reg_map)
my_reg_block (uvm_reg_block)
my_map1 (uvm_reg_map) addr1
my_map2 (uvm_reg_map) addr2
my_reg (uvm_reg)

Figure 4

UVM Predictor

- Normal extension of uvm_reg_predictor.
- Normal “write” -> “bus2reg” -> register object
- “get_local_map” returns the correct, default 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
   …
```

Multi-Interfaces

Illegal!

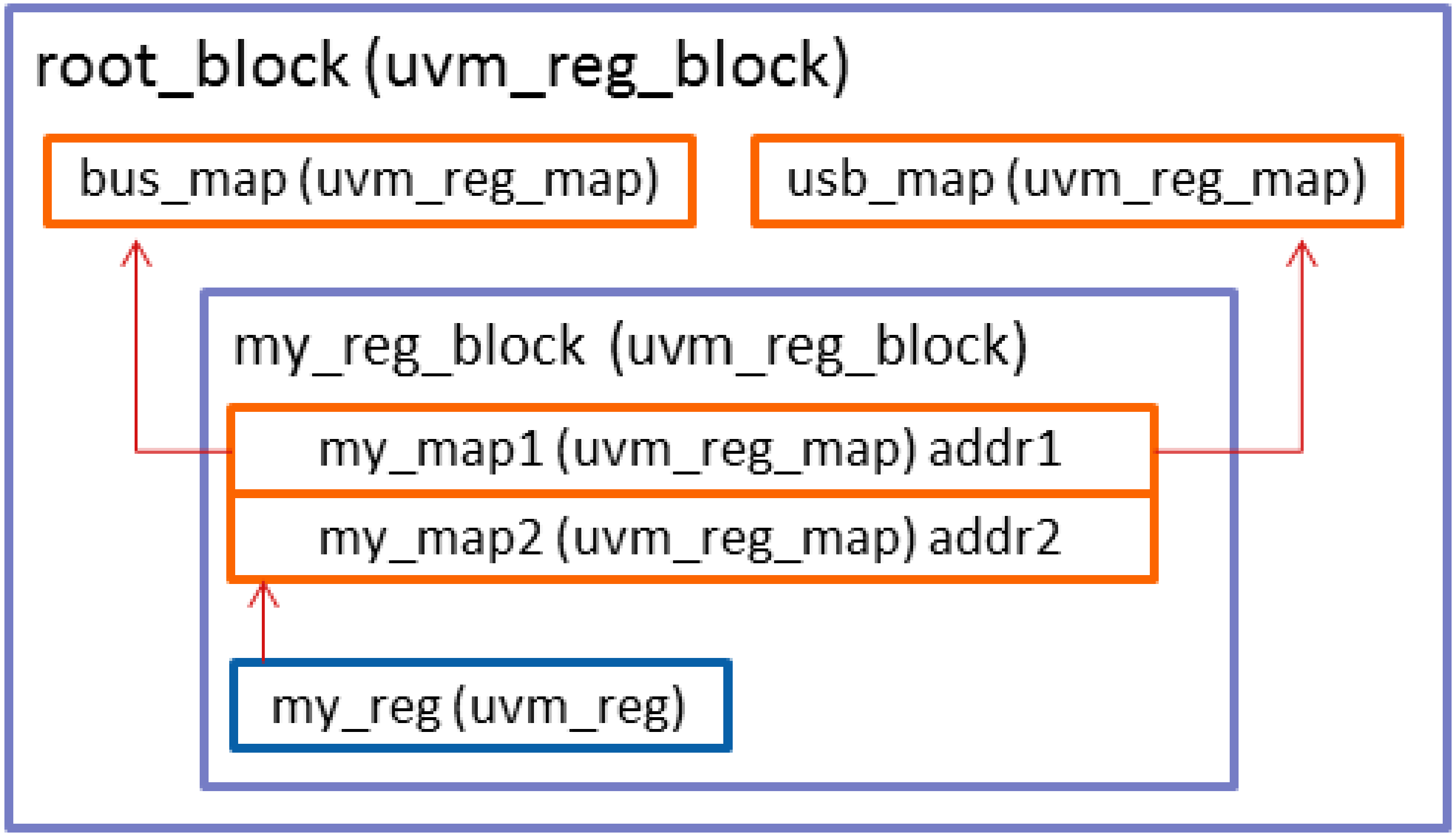

root_block (uvm_reg_block)
bus_map (uvm_reg_map)
usb_map (uvm_reg_map)
my_reg_block (uvm_reg_block)
my_map1 (uvm_reg_map) addr1
my_map2 (uvm_reg_map) addr2
my_reg (uvm_reg)

Figure 5

Multi-Interfaces

Legal!

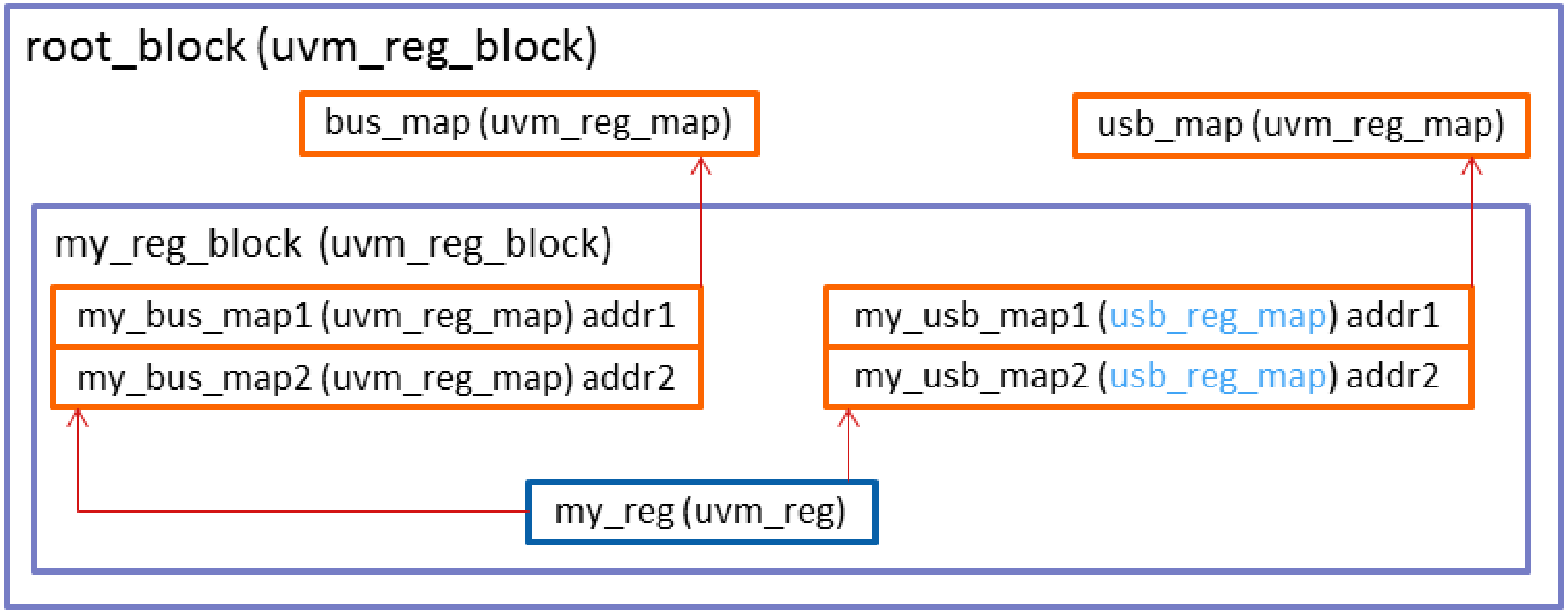

root_block (uvm_reg_block)
bus_map (uvm_reg_map)
usb_map (uvm_reg_map)
my_reg_block (uvm_reg_block)
my_bus_map1 (uvm_reg_map) addr1
my_bus_map2 (uvm_reg_map) addr2
my_usb_map1 (usb_reg_map) addr1
my_usb_map2 (usb_reg_map) addr2
my_reg (uvm_reg)

Figure 6

Multi-Modes

Represents what client expects, but only one interface!

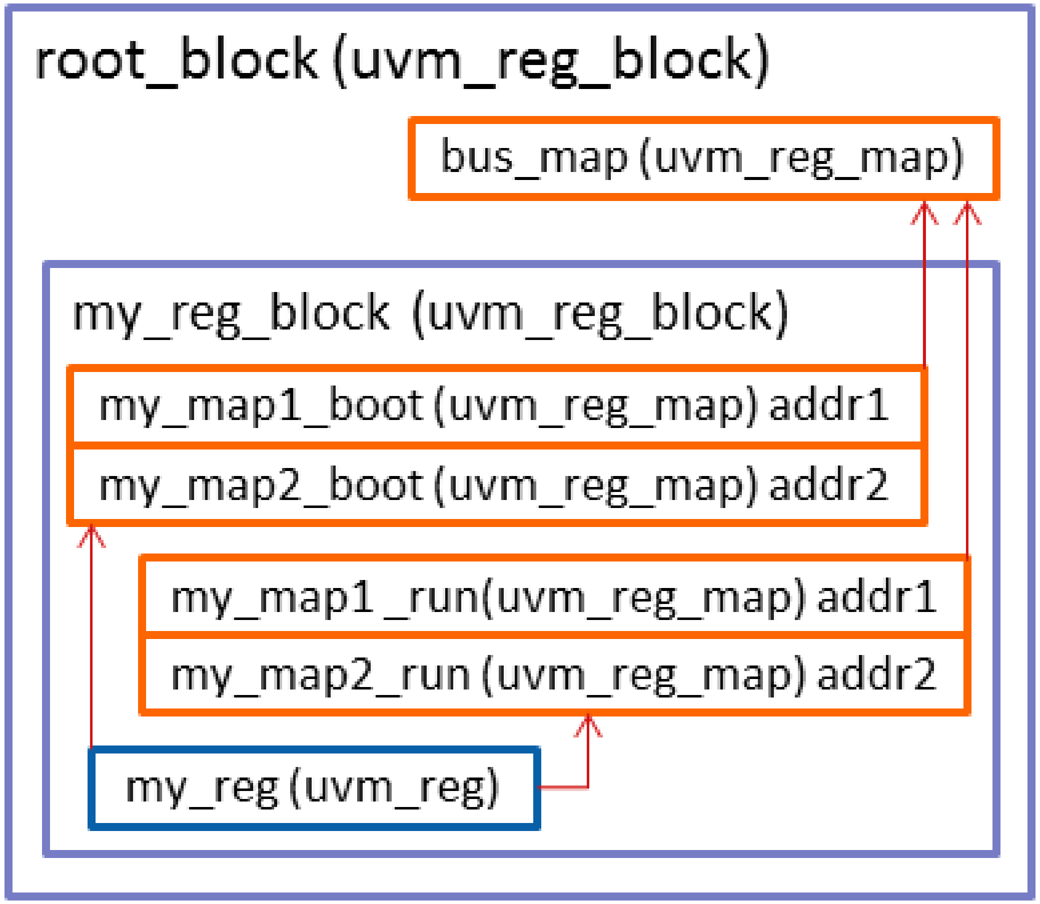

root_block (uvm_reg_block)
bus_map (uvm_reg_map)
my_reg_block (uvm_reg_block)
my_map1_boot (uvm_reg_map) addr1
my_map2_boot (uvm_reg_map) addr2
my_map1 _run(uvm_reg_map) addr1
my_map2_run (uvm_reg_map) addr2
my_reg (uvm_reg)

Figure 7

Multi-Interfaces & Modes

Supports both interfaces, but unacceptable to client!

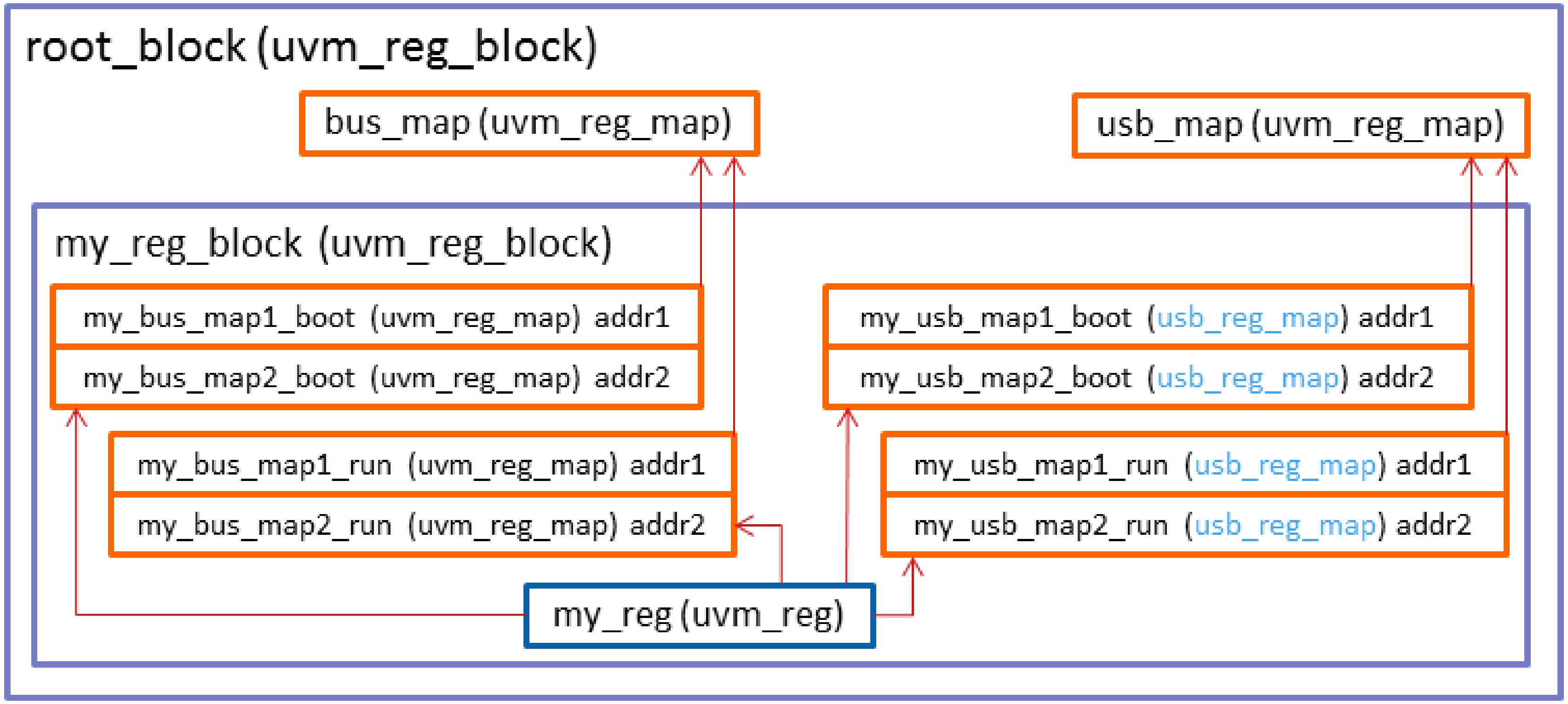
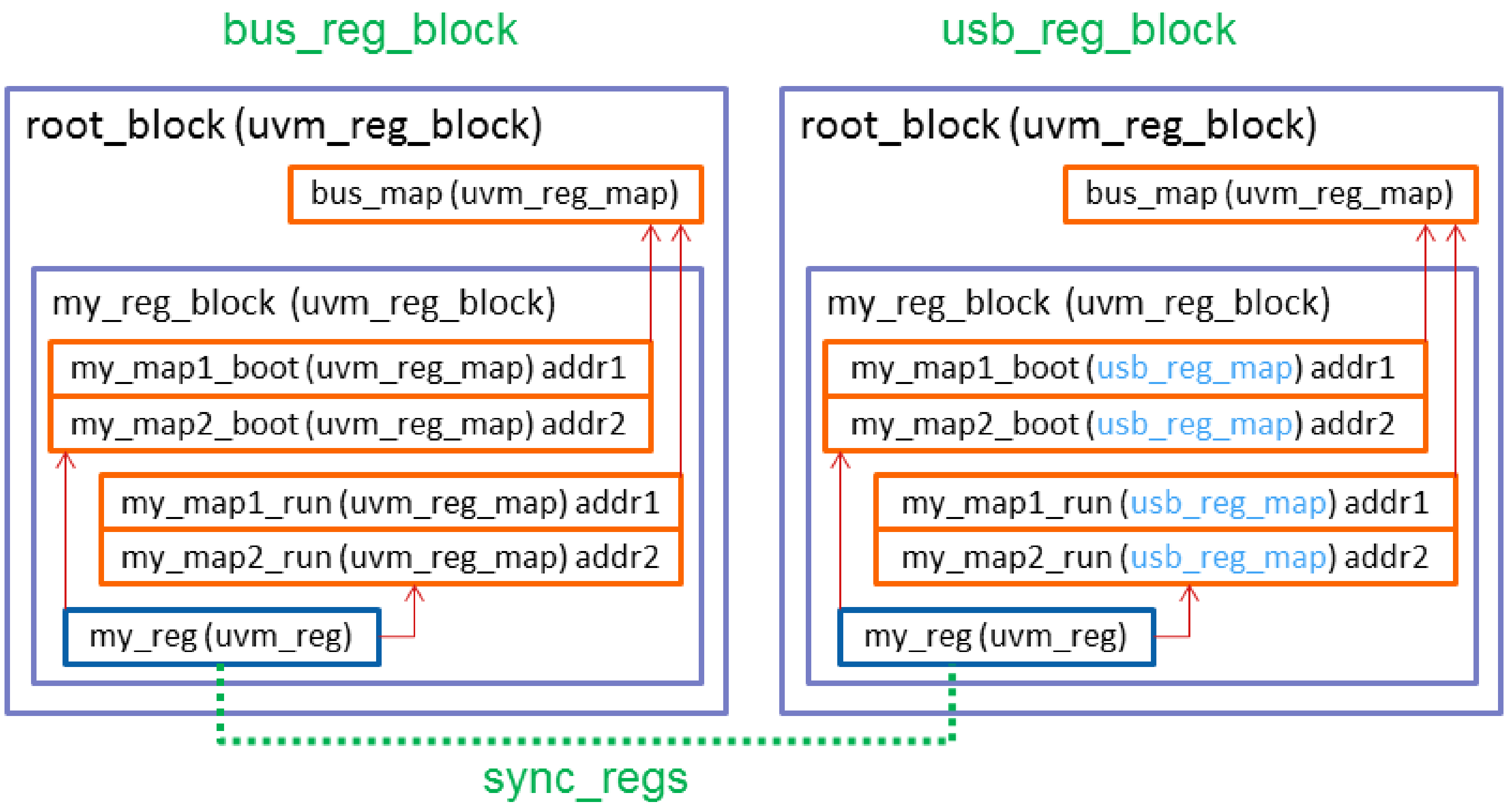

root_block (uvm_reg_block)
bus_map (uvm_reg_map)
usb_map (uvm_reg_map)
my_reg_block (uvm_reg_block)
my_bus_map1_boot (uvm_reg_map) addr1
my_bus_map2_boot (uvm_reg_map) addr2
my_usb_map1_boot (usb_reg_map) addr1
my_usb_map2_boot (usb_reg_map) addr2
my_bus_map1_run (uvm_reg_map) addr1
my_bus_map2_run (uvm_reg_map) addr2
my_usb_map1_run (usb_reg_map) addr1
my_usb_map2_run (usb_reg_map) addr2
my_reg (uvm_reg)

Figure 8

Multi-Interfaces & Modes

Acceptable to client, both interfaces supported!



bus_reg_block
usb_reg_block
root_block (uvm_reg_block)
bus_map (uvm_reg_map)
my_reg_block (uvm_reg_block)
my_map1_boot (uvm_reg_map) addr1
my_map2_boot (uvm_reg_map) addr2
my_map1_run (uvm_reg_map) addr1
my_map2_run (uvm_reg_map) addr2
my_reg (uvm_reg)
root_block (uvm_reg_block)
bus_map (uvm_reg_map)
my_reg_block (uvm_reg_block)
my_map1_boot (usb_reg_map) addr1
my_map2_boot (usb_reg_map) addr2
my_map1_run (usb_reg_map) addr1
my_map2_run (usb_reg_map) addr2
my_reg (uvm_reg)
sync_regs

Figure 9

Distinguishing Multiple Addresses and Modes

Bypass “get_local_map”

Overwrite RAL write function in predictor

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

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
```

Overwriting Submap Data Types

submap overwrite

```
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
```

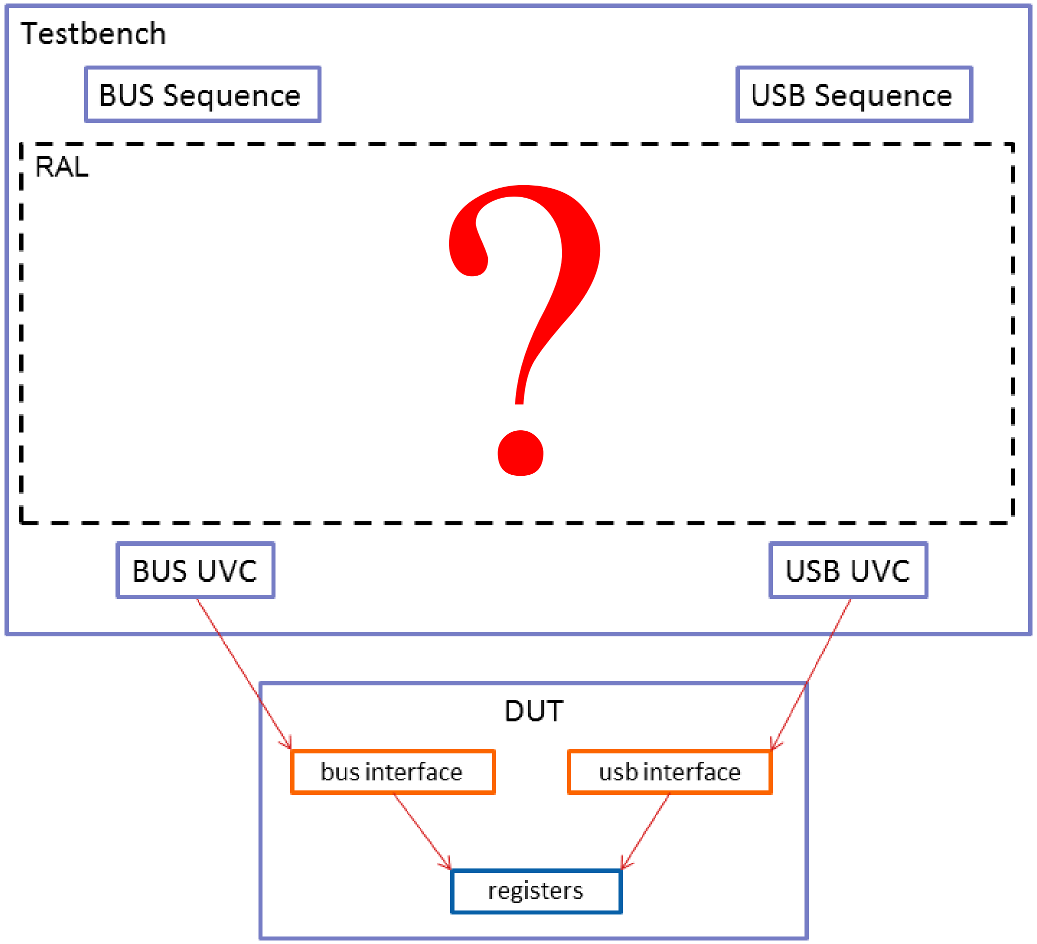
Synchronizing Registers

sync_regs.svh

```
function void my_bus2reg_rsp_predictor::sync_regs(
             uvm_reg  reg_obj, uvm_reg_bus_op rw);
  ...
   if(sync_reg[block_reg_name].size() == 0) begin
    foreach(reg_models[i]) begin
      ...
      // Build sync_reg for this reg_obj,
      ...
    end
  end
  ...
     temp = sync_reg[block_reg_name][i].predict(rw.data);
  ...
```

Conclusions

The Example Solution



Multi-Interfaces

Not trivial!

- Allow for multiple parents of submaps at top level?
  - Ready support for multi-interfaces
  - Hide multiple parents from clients (limit to default map)?

Multi-Modes

- Enhance RAL to distinguish modes?
  - Submaps?
  - Sideband signals?
- Ensuring no adverse impact is not trivial.

Further Development

Consider the corner cases

Might an enhancement:

- break the case of two maps with different offsets to a register?
- introduce conflict with complex block/map hierarchies?
- break functions provided to the user?
- break internal UVM RAL functions?

Options:

- Propose enhancement to the UVM Technical Steering Committee via Mantis
- Implement locally, barring limitations

Thank You