Managing Register Side Effects and Exceptions with UVM 1.2 RAL

Steven K. Sherman

AMD

Boxborough, Massachusetts, USA

www.amd.com

ABSTRACT

A verification environment that relies upon UVM 1.2 RAL can handle most general register access situations. However, in actual design environments with thousands of registers, there can be many register side effects and exceptions not well handled by a general approach. This paper discusses methods and approaches used in a RAL environment to address some of these challenges. These include ensuring reproducibility, using register hash tables, using register reset strapping, supporting register enabling, locking, broadcast and block aliasing, using grey and black lists in automated CSR testing and managing mirror check controls.

Table of Contents

Table of Figures

1. Introduction

UVM 1.2 RAL [1] includes support that seems to handle most register accesses well. The UVM User Guide [2] seems also sufficient for most register access though it does not go into much detail about actual register testing. The topic of register side effects and exceptions encountered during register testing was prompted by the SNUG community as a result of previous SNUG RAL presentations [3] and is a motivation for this paper.

During the course of register testing, register values often interact with each other as well as with the design. These interactions can be described with general rules. Such "side effects" may include observable behaviors such as changes to register accessibility, changes to field values of other registers, changes to design behavior and so forth.

There may also be special corner cases or "exceptions" to the general rules. For example, operation of a specific register field might work most of the time but fail when other register fields have certain settings.

Both side effects and exceptions may turn an apparently straightforward task of verifying registers into an extremely difficult and complex challenge. The register side effects and exceptions discussed here are not comprehensive but may be among the more significant. This paper describes general approaches developed and how they have been successfully applied. Though based on actual code, the examples presented here are relatively abstract and simplified to demonstrate basic concepts. This paper approaches side effects and exceptions by dividing these into register data structures, behavior and test.

Not addressed directly in this paper is the use of callbacks. Callbacks could have been chosen to circumvent much of the supported RAL infrastructure and gain greater customized control over register model processing. But, doing so may not leverage some features of the current RAL verification environment and may deny potential advantages of future RAL development. The approaches described here are intended to generally take advantage of RAL features while better enabling an environment to benefit from future UVM RAL improvements.

2. Register Data Structures

Managing side effects and exceptions begins with handling register data structures including use of register object hash tables. This helps address issues with performance and reproducibility when an environment supports thousands of registers.

2.1 Reproducibilty and Associative Arrays

The SystemVerilog specification [4] states that associative arrays are preferred over queues (dynamic arrays) when the size of the collection is unknown. However, a side effect to consider with register access is described by Mantis 4924 [5].

In short, Mantis 4924 asserts that implementation of associative arrays is currently not order-guaranteed in UVM 1.2. Mantis 4924 suggests use of uvm_reg queues instead of associative arrays. However, this may conflict with the stated preference in the SystemVerilog specification for associative arrays. This exception must be considered when managing register structures and functions that use associative arrays and may return lists of items. The order of items in the returned lists may be unpredictable.

For example, UVM provides a comprehensive set of access functions and tasks for the register model. This includes many "get" functions which can return lists of register model objects. But, sorting is needed with use of some register "get" functions that use associative arrays. This

includes `uvm_reg_block::get_registers` which returns a list using this associative array:

```
local int unsigned   regs[uvm_reg];
```

To help ensure reproducibility, a list returned by any call to `get_registers` can be sorted. An example of such a sort is included in 4.1 .

2.2 Register Hash Tables

The register layer model includes generic structures that are usually defined by a model generator. The resulting register model is usually quite complex and includes many details unique to the design. The sheer size and complexity of a register model may require special consideration.

Performance can suffer with frequent calls to "get" functions, forcing a search each time through the register model structures. To improve performance, create and use hash tables holding frequently accessed lists of register model structures, avoiding redundant searches. These tables can include a mix of associative arrays and queues depending on which approach seems more appropriate in actual code usage. In keeping with the previously described reproducibility issue, care should be taken to avoid ordering issues with returned lists from associative arrays.

Performance can suffer even more if searches involve relatively expensive string operations. Hashes can include indexes by string, but this does not avoid performance issues related to string operations. For now, indexing by string is commonly used as it is relatively easy to implement.

This discussion assumes a single, top register model that holds all register instances. In a multi-die configuration, an array of register models may be more appropriate. This would add an additional dimension to the hash tables.

A verification environment may use a few dozen register hash tables similar to these:

- **my_regs** – Includes all registers indexed by interface:
  ```
  local uvm_reg my_regs[interface_e][$];
  ```
  This table is used when accessing registers unique to particular interfaces. As this is a queue there are no reproducibility issues expected.
- **my_regs_addr** – Like my_regs but also includes an index by address:
  ```
  local uvm_reg my_regs_addr[interface_e][uvm_reg_addr][$];
  ```
  This table is used when accessing registers that may have the same address local to different components. As this is a queue there are no reproducibility issues expected.
- **my_regs_name** – Like my_regs, this also includes an index by register name:
  ```
  local uvm_reg my_regs_name[interface_e][string][$];
  ```
  This table is used to detect when there may be more than one register with the same name. As this is a queue there are no reproducibility issues expected.
- **my_regs_comp_name** – Includes all registers indexed by register component and name:
  ```
  local uvm_reg my_regs_comp_name[int][string];
  ```
  This table is used when processing a set of registers in a component that may refer to different registers with the same name. There is no reproducibility issue expected as there should be only one unique register by name per component.
- **my_blocks_name** – Includes all register blocks indexed by interface and by block name:
  ```
  local uvm_reg_block my_blocks[interface_e][string];
  ```
  This table is used when processing a set of register blocks that share the same name across multiple components. There is no reproducibility issue expected as all of the returned

blocks are processed at once.

- **my_blocks_comp** – Like my_blocks_name but also includes an index by component:
  `local uvm_reg_block my_blocks_comp[interface_e][int][string];`

  This table is used when collecting all blocks in a component. All blocks are carefully processed in the returned list to help ensure reproducibility.

- **my_blocks_comp_reg_name** – Like my_blocks_comp but also includes index by register name rather than by block name:
  `local uvm_reg_block my_blocks_reg_name_comp[interface_e][int][string];`

  This table is used when detecting which block contains a specific register. While order may vary in the returned list, the register model is constructed such that there should only be one unique entry in the returned list.

This paper includes code samples featuring `my_regs_name` and `my_regs_comp_name`.

3. Register Behavior

Having addressed some side effects and exceptions in the register data structures, the next step is to address side effects and exceptions associated with register behavior. This section covers register reset strapping, enabling, locking, broadcast and block aliasing. All of these involve behavior that can vary per register.

3.1 Reset Strapping

Approximately two dozen registers (many repeated across various components) may be reset to values determined by the device under test at run time. Each has its own unique reset value which must be matched in the register model. The value usually depends upon features unique to the design variant which are available to the verification environment. The challenge of these exceptions is to implement a consistent approach that can be readily adapted for each strapped register. Each register will have its own unique function to set its reset value with a few support functions available for coding simplicity and convenience.

When the reset signal asserts, a monitor calls a reset analysis port which invokes a general reset for registers. Within that function, the strapped reset function is called. This function calls all register reset strap functions. This includes passing appropriate information to each function to direct to specific register instances as required. An example of a reset strap function is shown in Figure 1.

```
function void reset_strapped_MY_REGISTER(uvm_reg reg_obj);

  automatic uvm_reg _reg_obj = reg_obj;
  bit     status;

  reset_strapped_begin_msg(_reg_obj);

  fork : thread_reset_strapped_MY_REGISTER
    begin
      status = _reg_obj.get_field_by_name("MyField").predict(
                  get_feature($sformatf("DutFieldValue")),
                  .kind(UVM_PREDICT_DIRECT));
      reset_strapped_end_msg(_reg_obj);

    end
  join_none

endfunction : reset_strapped_MY_REGISTER
```

Figure 1: reset_strapped_MY_REGISTER

3.2 Enabling

All access to some registers may be blocked unless enabled by other registers. In order to mimic this behavior, predictors call `check_reg_enable` with each transaction before taking action that can affect the register model as shown in Figure 2.

```
if(!my_ral.check_reg_enable(rg)) begin
  rw.status = UVM_NOT_OK;
  m_pending2.delete(rg);  // delete pending entry
  continue;
end
```

Figure 2: check_reg_enable Usage

The `check_reg_enable` function references two hash tables to determine if access is enabled for a register object as shown in Figure 3.

```
// Check if register is enabled.
function int my_ral::check_reg_enable (uvm_reg reg_obj);
  int        enabled = 1; // default is enabled
  uvm_reg enabling_reg;
  uvm_reg_field enable_field;

  if(reg_obj == null) return 0; // not enabled
  enabling_reg = my_register_enable[reg_obj];
  if(enabling_reg == null) return enabled;
  enable_field = my_register_enable_field[enabling_reg];
  if(enable_field == null) return enabled;
  if(enable_field.get_mirrored_value() != 0) enabled = 1;
  else enabled = 0;

  return enabled;

endfunction : check_reg_enable
```

Figure 3: check_reg_enable

The `my_register_enable` hash table uses the register object as an index to its enabling register. The `my_register_enable_field` hash table uses the enabling register object as an index to its enable field which then determines enable status. The status returned is enabled by default.

An additional consideration is that a separate register field value may control access to all registers within a specific component. This is a relatively simple check and can be made part of the `check_reg_enable` function where the component of the register is determined and access is checked against the separate register field.

3.3 Locking

Some registers may be readable but have write access blocked as controlled by other registers. In order to mimic this behavior, predictors call `check_reg_lock` with each transaction before taking action that can affect the register model as shown in Figure 4.

```
if(rw.kind == UVM_WRITE) begin
  if(my_ral.check_reg_lock(rg)) begin
    rw.status = UVM_NOT_OK;
    m_pending2.delete(rg);  // delete pending entry
    continue;
  end
end
```

Figure 4: check_reg_lock Usage

The `check_reg_lock function` references two hash tables to determine if access is locked for a register object as shown in Figure 5.

```
// Check if register is locked.
function int my_ral::check_reg_lock (uvm_reg reg_obj);
  int locked = 0;
  uvm_reg locking_reg;
  uvm_reg_field lock_field;

  if(reg_obj == null) return locked;
  locking_reg = my_register_lock[reg_obj];
  if(locking_reg == null) return locked;
  lock_field = my_register_lock_field[locking_reg];
  if(lock_field.get_mirrored_value() != 0) locked = 1;
  return locked;

endfunction : check_reg_lock
```

Figure 5: check_reg_lock

The `my_register_lock` hash table uses the register object as an index to its locking register. The `my_register_lock_field` hash table uses the locking register object as an index to its lock field which then determines lock status. The status returned is unlocked by default.

3.4 Broadcast

Some registers can be optionally accessed via broadcast. These are registers in separate components but with matching register name within each component. When not in broadcast mode, these registers are accessed individually. In broadcast mode, when one register is accessed, other registers of matching name in all other components are also accessed.

```
function void my_ral::sync_regs(uvm_reg reg_obj,
                                 bit is_bcast);
  uvm_reg rg[$];
  string reg_name = reg_obj.get_name();

  // Schedule rg updates.
  foreach (intf_access_types[i]) begin // traverse all existing interfaces
    if(is_bcast) begin
      if(my_regs_name[intf_access_types[i]][reg_name].size() > 1) begin
        foreach(my_regs_name[intf_access_types[i]][reg_name][j]) begin
          uvm_reg my_reg = my_regs_name[intf_access_types[i]][reg_name][j];
          // Do not attempt to update reg_obj, already updated.
          if(my_reg != reg_obj) begin
            rg.push_front(my_reg);
          end
        end
      end
    end
  end

  // Process scheduled updates.
  foreach(rg[k]) begin
    uvm_reg_data_t data = rg[k].get_mirrored_value();
    my_ral::predict(rg[k], data); // Update MIRROR and handle callbacks.
  end
endfunction : sync_regs
```

Figure 6: sync_regs

In the case of a write, all registers of matching name in other components are updated in the register model per the write value. Typically, all fields of such registers have matching implementation. In the case of a read, all such register values must be OR'd and checked against the return value.

This requires determining all registers of matching name in other components and updating their mirrored values. This is accomplished within a `sync_regs` function which is called with every detected register write as shown in Figure 6.

This function detects registers to update by name. It avoids a repeated update of the original register as it schedules these for update via a special predict call. The special predict is based on the uvm_reg predict but also manages callbacks and does not return status.

3.5 Block Aliasing

Block aliasing involves separate UVM register blocks with matching registers. When one register is accessed, its match in an aliased block is also accessed. Within aliased blocks, the registers exist separately but are accessed concurrently. In the case of a register write, registers of matching name in aliased blocks must be updated in the register model per the write value. However, each may update with a different value as the actual fields implemented may vary. In the case of a register read in aliased blocks, a calculated OR of registers should be checked against the return value. Block aliasing is considered in the actual code when determining register enables and locks previously discussed as well as when calling `sync_regs`.

This requires determining a corresponding register in an alias block per a given register object. Block aliasing is determined by associating blocks with each other by name. So, the process of handling affected registers involves processing a register object, matching the register's block with an alias block by name, then applying the same process to a corresponding register in that alias block.

```
string alias_1[$] = {"my_rg_blk_1",  "my_rg_blk_2",  "my_rg_blk_3"};
string alias_2[$] = {"my_rg_blk_1a", "my_rg_blk_2a", "my_rg_blk_3a"};
uvm_reg_block reg_block = reg_obj.get_parent();
string reg_name = reg_obj.get_name();
string block_name = reg_block.get_name();

// Process reg_obj.
my_process_reg(reg_obj);

// Do my_process with alias of reg_obj, if it exists.
for(int alias_id = 0; alias_id < alias_1.size(); alias_id += 1) begin

  string target_block_name = get_alias_block_name(block_name, alias_id);
  if(target_block_name.len() != 0) begin
    int component = get_component_from_block_name(target_block_name);
    alias_reg_obj = my_regs_comp_name[component][reg_name];
    if(alias_reg_obj != null) begin
      my_process_reg(alias_reg_obj);
    end
  end
end
```

Figure 7: get_alias_block_name Usage

In the example of Figure 7, a register object is processed with `my_process_reg`. The same process is then applied to a corresponding register in an alias block, if it exists. Two ordered string

lists, `alias_1` and `alias_2`, define the alias blocks by name. The register could belong to one of the blocks in either list or to a block not in either list. To determine a corresponding register in the alias block, a loop steps through the lists and calls `get_alias_block_name` to potentially return the name of an alias block.

The `get_alias_block_name` function (Figure 8) uses string comparisons with the ordered string lists to return an alias block name if it exists. This could be done with hash tables, but is done here with string manipulation for simplicity.

```
// Get alias for named register block, if it exists.
function string my_ral::get_alias_block_name(
              string block_name, int alias_id);
  string alias_block_name = "";

  if(block_name == alias_1[alias_id]) begin
    alias_block_name = alias_2[alias_id];
  end
  else begin
    if(block_name == alias_2[alias_id]) begin
      alias_block_name = alias_1[alias_id];
    end
  end

  return alias_block_name;
endfunction : get_alias_block_name
```

Figure 8: get_alias_block_name

4. Register Test

Having addressed side effects and exceptions in registers in data structures and register behavior, the remaining step is to address issues during test. Register side effects and exceptions are naturally discovered and managed during most tests. Here, a specific test is generated to attempt access to all registers. This also tends to uncover many side effects and exceptions to be managed. Even so, there may yet remain intractable side effects and exceptions requiring more drastic action.

4.1 Automated Register Test

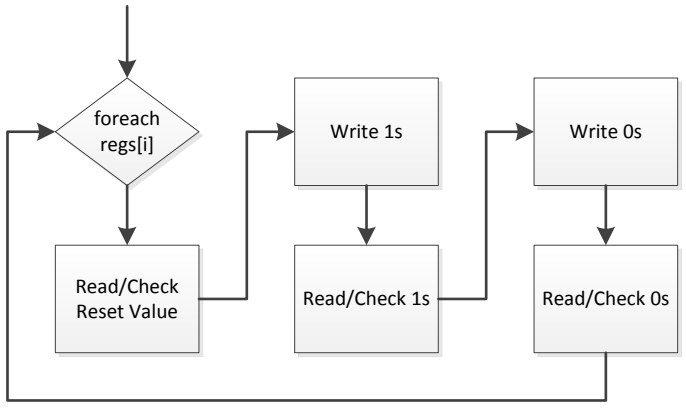

Figure 9: Simple Register Sequence

The majority of tests exercise specific register access and function. To improve register verification

coverage, a sequence is created that attempts access to all registers in the register model. This sequence may start as a simple register sequence. For each register in the register model, it confirms reset value, then toggles and checks all bits per Figure 9.

This sequence relies upon a list of registers built with something like:

```
reg_model.get_registers (regs, 1);
regs.sort() with (item.get_full_name());
```

Note that a sort is included after the `get_registers` call to help ensure reproducibility.

This simple register sequence can be regularly run in a verification environment that incorporates all of the concepts previously discussed. An additional consideration is to run this test as several tests, splitting the registers into segments to keep simulation times reasonable. As this sequence focuses on access rather than function, some features and scoreboards can be disabled. Also, initialization to configuration values can be disabled in order to read and check register reset values. However, configuration values should still be calculated as they may yet be useful as described in 4.1.4 .

This sequence may be one of the later tests developed. Prior to this, other tests can be expected to successfully exercise many registers. This supports reasonable expectations that most registers should function as intended. This sequence can then be developed and run to uncover potential remaining register access issues.

4.1.1 Single Register Add

Consider what happens if a new register is added to the register model with this test. By default, the simple register sequence may be run as shown. But, this may have failures that reveal issues including side effects or exceptions associated with this register. The register may fail due to:

- an unpredictable field
- a simple corner case
- a complex corner case
- any write attempt
- any read attempt

For any of these issues, the sequence can be modified to handle side effects or exceptions. Usually, failure is due to an oversight in the specifications, to undocumented or new features or to ambiguity in specifications requiring more clarification. On occasion, a failure is due to a design issue missed by other testing. For these reasons, failures must be individually root-caused and fully understood before applying any of the modifications discussed next.

These modifications are ordered from the modification with least negative impact on coverage to the modification with most negative impact.

4.1.2 Unpredictable Field

On very rare occasions, if it is impossible to predict what value a specific register field will return during this particular test, then the sequence can disable the check by calling the register field object `set_compare` function with parameter `UVM_NO_CHECK` (discussed in more detail in 4.2.2). This modification has the least negative impact on coverage.

4.1.3 Simple Corner Case

If the register simply fails a particular corner case, trap the register in the code and ensure the corner case is not hit by writing and checking against special values. For example, if an enable bit

should never be set randomly, then ensure that bit is never set by writing with special, hand-coded values that do not set that bit.

There may be less than a dozen registers that have such corner cases in an environment. These special cases can be hand-coded into the test.

4.1.4 Complex Corner Case

If a register has fields that must abide by strict rules that are more complex, then consider limiting a write to use the calculated configuration value. The configuration value usually differs from the reset value and is calculated at run time for the register per constraints. This may enable a write to occur, using a value that should be good per configuration constraints. A read may then proceed normally.

Note that the configuration value might not otherwise be used as it may have been calculated but not enabled for this test. That is, this sequence may have initialization routines turned off that would otherwise have used the configuration value to initialize the register.

There may be about a hundred registers in an environment that have this issue. These registers can be tracked in a write greylist. The sequence can compare the register against the greylist to see if it should write with the calculated configuration value.

4.1.5 Any Write Attempt

If any write to a register results in error with little regard to the value written, then the register may be added to the write blacklist. An environment may have less than a dozen of these registers. These may be special registers that alter fundamental register access and design functions. They should be more carefully tested elsewhere.

4.1.6 Any Read Attempt

Finally, if any read to a register results in error, then the register may be added to the read blacklist. An environment may have about two dozen such registers. This modification has the most negative impact on coverage and is therefore the last resort. These registers may include debug, diagnostic and indirect access registers which abide by special access rules. They should be more carefully tested elsewhere.

4.1.7 Register Sequence with Modifications

Application of these modifications is summarized in Figure 10. The best coverage is likely obtained by taking the paths furthest to the right in the figure, as a rule.

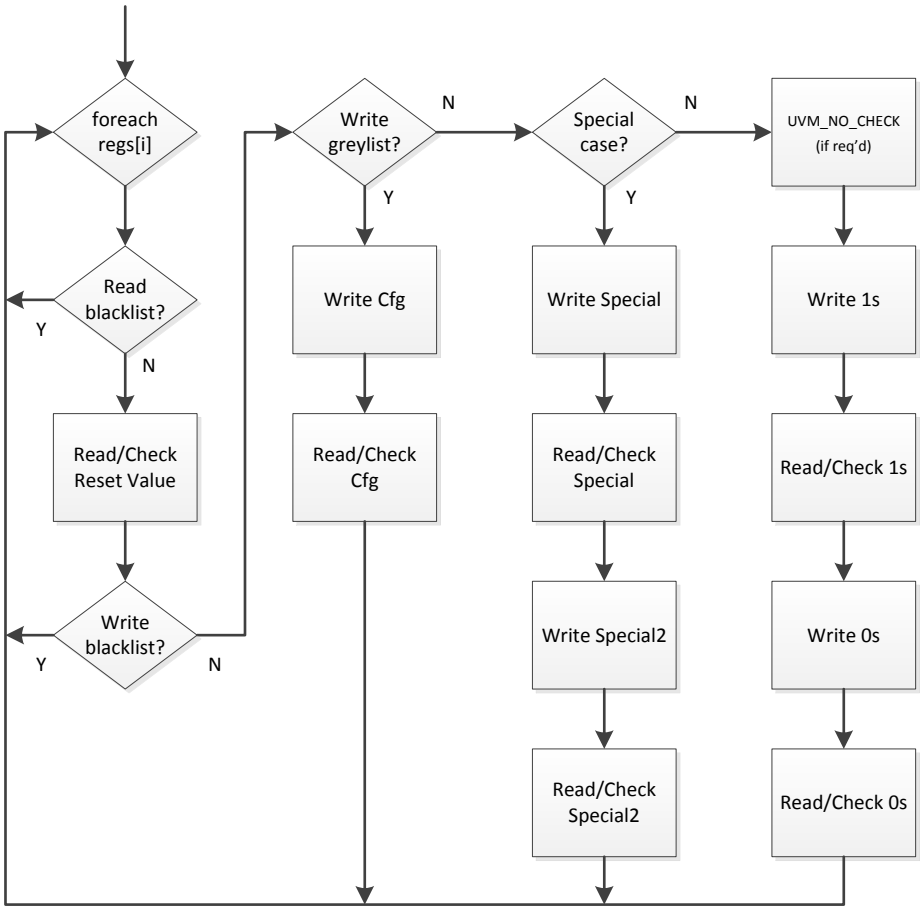

Figure 10: Modified Register Sequence

4.2 Mirror Check Control

In spite of reasonable efforts to address side effects and exceptions of register access, there can be some issues that cannot be addressed with the previously described methods. This can happen when, for example, new functionality is only partially implemented and access must be temporarily allowed with checking against mirrored value suspended. Two approaches to remedy this are discussed here. One is a drastic approach that turns off all checks against mirrored values. The other turns off checking for a particular register and is preferred over turning off all checks.

4.2.1 Command Line: no_mirror_check

This is a drastic approach that should only be used temporarily during debug and development. It involves passing parameter `no_mirror_check` from the command line to the infrastructure. This parameter is tested by a function called by predictors and by the infrastructure before checks are performed on transactions to prevent related checks against mirrored register values.

4.2.2 Function: set_reg_check

This is a safer approach that is limited to a particular register object. This involves a call, typically from a sequence, to `set_reg_check` (Figure 11). This function accepts a register object and can limit check control to all fields or to a particular field.

```
function void set_reg_check(uvm_reg reg_obj, uvm_check_e check = UVM_NO_CHECK,
                            string field_name = "");

  uvm_reg_field fields[$];

  reg_obj.get_fields(fields);

  foreach(fields[i]) begin
    if((field_name == "") ||
       (field_name == fields[i].get_name())) begin
      fields[i].set_compare(check);
    end
  end

endfunction : set_reg_check
```

Figure 11: set_reg_check

By default, `UVM_NO_CHECK` and an empty field name will disable all automatic mirrored checks for the register object. A sequence that only requires the suspension of a check for a specific field can do so via the field_name parameter. This is often much preferred over disabling all field checks for a register. The check can be enabled later by setting the check parameter to `UVM_CHECK`. This is also preferred over keeping checks disabled for the register.

In an environment, `set_reg_check` may be used to disable checks for one field in one register for only one particular sequence. In this instance, the disable may be needed because the field is unpredictable for this particular sequence. For other sequences, the field may be predictable with checking required. At other times, this function can be used by sequence developers to temporarily mask a register side effect or exception to be addressed later.

`set_reg_check` disables checks of registers by field. Even though this function may prevent checks of register fields, it has no effect upon checks of non-field bits. For some register value checkers of all register bits, this can result in mismatches even though field checks are disabled.

5. Conclusions

At any given time, there may be a half-dozen or so projects in process. Each project may require support for a dozen or so testbenches. Each testbench typically has thousands of registers, the vast majority of which may be successfully accessed by the automated register sequence presented. The described approaches with register data structures, behavior and test, have been used to successfully address register side effects and exceptions. This helps with improving performance and maintainability for verification environments with very large numbers of register objects and related structures.

A future enhancement of UVM might provide list sorting upon return from some “get” functions. This might be controlled by an optional parameter for such functions, enabled by default. It might also include some manner of globally disabling the feature should legacy behavior be preferred.

Future enhancements of UVM RAL might also include support for a common set of reproducible register and block hash tables. There might be additions to the set of RAL functions and tasks that might make initialization of register hash tables more straightforward. This is not a topic addressed directly in this paper but does represent a significant amount of actual code. Performance might also be improved if there is some way to save and restore hash tables in separate simulations when the register model is unchanged.

Finally, for performance improvement, future enhancements of UVM RAL should move away from string functions, in general. Instead, use of enumerated types should be encouraged for register and field names instead of strings. These enumerated types could be generated by the register model generator along with the register model. Use of enumerated types instead of strings could greatly accelerate performance of the hash tables cited here where currently indexed by string.

6. References

[1] VIP Technical Committee. Universal Verification Methodology (UVM) 1.2 Class Reference. Accellera, June 2014.

[2] VIP Technical Committee. Universal Verification Methodology (UVM) 1.2 User's Guide. Accellera, October 2015.

[3] Sherman, Steven K. Customization of RAL Adapters and Predictors in UVM 1.2, SNUG Boston, 2015.

[4] IEEE, IEEE Standard for SystemVerilog 1800™-2012, 2013.

[5] Khan, A., foreach loop iteration over assoc array is not order guaranteed, EDA.org Mantis 4924, May 2014 < https://accellera.mantishub.io/view.php?id=4924>