BEYOND UVM REGISTERS — BETTER, FASTER, SMARTER

RICH EDELMAN AND BHUSHAN SAFI, MENTOR GRAPHICS

Mentor Graphics®

FUNCTIONAL VERIFICATION

WHITEPAPER

www.mentor.com

INTRODUCTION

The UVM Register package [2] has many features. These features include reading and writing register values, reading and writing register fields and register blocks. The register model keeps track of the expected value and can directly access the actual modeled register using "back-door access". Using the register model allows a testbench to be written that can check the behavior of registers and address maps. Additionally, the register model can be constrained and randomized to provide stimulus or configurations. It can also be "covered" with functional coverage. These are the main operations of any register package; model the expected and actual values; check address mapping; generate random stimulus; allow direct access; support functional coverage.

The problem in the UVM Register package arises in the intricacies in the register layer source code. For uvm-1.1d, there are 26 files containing over 22,000 lines of source code. That represents 19% of the total number of files in the UVM, and 32% of the total number of lines of source code. Clearly the UVM Register package is a large part of the UVM. In the next release, uvm-1.2, the percentages are reduced by increasing the files and lines in the overall UVM, but the absolute number of files is still 26, and the lines of source code is still over 22,000 for the UVM Register package. In the uvm1.1d User Guide, the description of the UVM Register layer is 26% of the number of pages.

The UVM Register package is a large part of the UVM, by file and line count. There are many detailed, complete tutorials, papers and discussions about UVM Registers, and their various capabilities. These include a detailed description of pros and cons to various quirky register implementation methods [3]; an interesting tutorial on register access routines and the register computing model [4]; various tutorials and UVM register scenarios [5] [6]; and discussions about field modeling [7] [8]. A Google search for "UVM Registers" will yield many high quality results discussing various UVM Register aspects. This community of support is excellent and a welcomed sight for new users. But, one of the reasons that this community has grown so large, and so necessary is exactly the issue with the UVM Register package. It is big. It is complex. It is powerful.

This paper proposes a re-think. A reconsideration of the goals and objectives, along with the implementation decisions. It discusses basic register model requirements and a suggested implementation and conceptual model that completely eliminates the need for a register library class, and the accompanying VPI interface to directly access the hardware.

USING A REGISTER PACKAGE

Any verification team might use registers to answer the following questions. Any register modeling package should make getting these answers straightforward, transparent and easy.

Can I read and write all my registers?

Can I read and write all the addresses for all my registers on all interfaces allowed?

Can I set all bits? Can I clear all bits? Can I read all bits?

Do the registers function properly (including Quirky Registers)?

Can I operate my DUT correctly when I load a set of registers (when I do a configuration) and press "go"?

REGISTER MODEL USE MODELS – WHAT DO I WANT TO DO WITH MY REGISTERS?

A register model value can be read or written – either using a read() or write() routine, or through direct access (register_handle.value = value). A register model usually has a value associated with the "model" or expected value, and another value associated with the current DUT value. These values can be set and checked to verify correct functional behavior.

A "configuration" can be created that consists of one or many registers. These registers taken together represent a "state" or "configuration" which the DUT can be set to. By writing all the registers in the configuration, the DUT is set into the proper state. It can then be set to run, performing some activity, the results are well known.

A register model will provide a way to access the real register value either by the "front-door" or the "back-door". The front-door access is accomplished by some regular bus access on a bus interface. It takes simulation time, and in turn wall clock time. The back-door access is an "out-of-the-box" access, either via VPI, DPI, or other techniques. These back-door accesses are not part of the normal simulation or functional DUT behavior, but provide the testbench a fast way to access the internal DUT values. Back-door access takes no simulation time, and very little wall clock time. Using back-door access allows fast ways to read and write register values.

Registers are often made up of fields. Accessing registers by fields is one of the main considerations for a register model, since a field is a collection of related bits, usually treated as a unit.

REGISTER MODEL DEFINITIONS

A register is a collection of bits. Those bits can be grouped into fields. A register model duplicates this organization, and allows for stimulus generation, coverage and scoreboarding. A register in the DUT exists in only one place. It has a fixed name. For example, there is one such register DUT.BLOCKA.REGX.

Registers can be organized into a collection called a block. The block is a simple container for organizational purposes. Often, registers in a block are at offsets from a base address. Blocks can be handy ways of creating address maps, and handy ways of writing tests. But a block is just a simple container, and implies no other meaning. A block can contain another block with no implied meaning from the hierarchical organization. In the end, the block has a handle to a register. The register has a name – the instance name in the DUT.

An address map is a collection of registers. Each register can have 1 or more addresses. Given a block of registers and an address map, tests can be written.

Register access is simple. Each register read or write is really just a lookup to get an address, then the register read and write is reduced to a bus read and write at that address. Bus level requirements on transfer size or other issues can be handled by the regular bus interface.

A REGISTER MODEL

We begin with the idea that the UVM Register model is large and that modeling a register with a class (and modeling a field with a class) is overkill. A register is naturally modeled with a SystemVerilog packed struct:

```
typedef struct packed {
  logic [2:0] a;
  logic [3:0] b;
} REGA_t;
```

Figure 1. Packed struct register field description

A packed struct model is really just a fancy bit vector. If that fancy bit vector is modeled in a module or interface, it can be "bound" into the actual DUT model, effectively connecting the two without the need for VPI. We will model registers as an interface which has a variable called a 'value'. That interface will be bound into the DUT for easy back-door access. The bound in model creates an easy way to monitor, stimulate and cover the actual register.

```
interface REGA (input  register_types_pkg::REGA_t in,
                output register_types_pkg::REGA_t out);
  typedef register_types_pkg::REGA_t T;
  `include "common_register_code.svh"

  function void write(T v);
    value = v;
  endfunction

  function T read();
    return value;
  endfunction
endinterface
```

Figure 2. Complete register definition

Each register definition shares common code by way of an include statement. This common code defines the reset_value, the modeled value, the monitored dut_value, the name and a variety of other functionality. These are the basic abilities that are common to almost all registers.

```
T reset_value;
T value;     // Model value.
T dut_value; // Mirror of the DUT value.

string name = $sformatf("REG: %m");

function string convert2string();
  return $sformatf("%s: %p [DUT: %p]",
    name, value, dut_value);
endfunction

event BACKDOOR_LOAD;

always @(value)
  $display("%s", convert2string()); // Simple monitor

always @(in)                        // Backdoor monitor
  dut_value = in;

always @(BACKDOOR_LOAD)             // Backdoor load
  out = value;

function bit dut_check();           // Built-in checker
  if ( dut_value != value ) begin
    $display("@%t: (%m) Mismatch. DUT_VALUE=%p, VALUE=%p",
      $time, dut_value, value);
    return 0;
  end
  return 1;
endfunction

function void check_reset();          // Built-in reset value checker
  if ( reset_value != dut_value )
    $display("@%t: (%m) Reset mismatch. DUT_VALUE=%p, RESET_VALUE=%p",
      $time, dut_value, reset_value);
endfunction
```

Figure 3. Shared register code – common_register_code.svh

The register model operates as in the diagram in Figure 4. The model is bound into the DUT. The SystemVerilog packed struct allows direct access to read or write fields. Coverage can be built on the fields. The fields can be randomized. Many register accesses can be direct – register_handle.value.

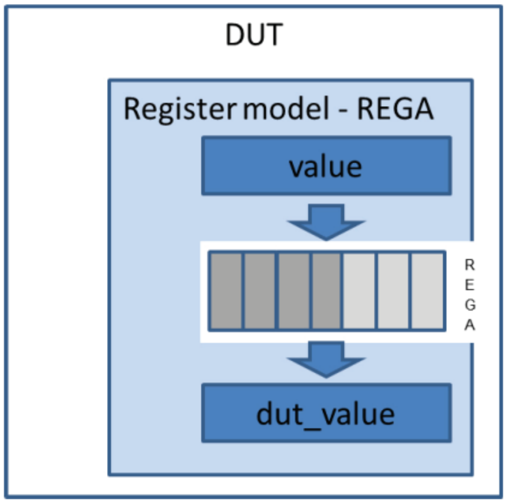

Figure 4. A register model "bound" into the DUT

For Quirky registers read() and write() APIs can be used to implement the special behaviors. As the complexity of the Quirky Register grows, the read() or write() routines may become more complex, but this complexity should be shielded from the end user by API, and shielded from the register modeler by the register model generation script.

In SystemVerilog the "%p" notation can be used to automatically pretty print the register value. An easy implementation of the necessary convert2string() functionality is listed in Figure 3.

WRITING A TEST

Writing a test is relatively the same in this new register model as in the UVM Register model. A vif handle represents the register, and the value is accessed either directly or by using an access routine like read() or write(). A helper bus interface sequence is used to issue the bus transfer after retrieving the register address from the address map.

In Figure 5, the model value is written to the register, then a read is issued for the same register. The written and read values are checked.

```
wdata = bA.rega.value;
seq.write(address_map.get_address(bA.rega.name), wdata);
seq.read (address_map.get_address(bA.rega.name), rdata);
if (wdata != rdata) // Self checking
  `uvm_info(get_type_name(),
    $sformatf("%s: Mismatch wrote %x, read %x",
      bA.rega.name, wdata, rdata), UVM_MEDIUM)
```

Figure 5. Direct register value access – using the ".value"

In Figure 6, a value is written to the register model for the broadcast control register. Then a bus interface write is issued for the same value to the register. Finally, the built-in dut_check() routine is used on each of the broadcast register targets. The dut_value (monitored) will be compared with the modeled value in each of the target registers.

```
wdata = 1;
for (int i = 0; i < 5; i++) begin
  bA.broadcast.write(wdata);
  seq.write(address_map.get_address(bA.broadcast.name), wdata);
  bA.b1a.dut_check();
  bA.b1b.dut_check();
  bA.b2.dut_check();
  wdata++;
end
```

Figure 6. Using the API – read() and write() access

The way that a register model is bound into the actual DUT allows a test to set registers or start processing and then easily wait for a value change. For special processing, a simple event can be waited for using SystemVerilog. In Figure 7, any value change in rega's monitored dut_value will be captured in the coverage collector.

```
forever begin
  @(bA.rega.dut_value);
  cg_REGA.sample(bA.rega.dut_value);
end
```

Figure 7. Wait for a register change, then perform some action

QUIRKY REGISTERS

Quirky Registers are registers that have "special" behaviors. Behaviors are well documented in the UVM Register package. These are the "regular" behaviors which are easily described by a Boolean equation. For example a READ-ONLY register or a WRITE-ONLY field. These so called "Boolean equation" Quirky registers are relatively straightforward to implement.

More complicated quirky registers contain WRITE-ONCE fields or CLEAR-ON-READ fields. A clear-on- read register is defined in Figure 8.

```
interface CLEAR_ON_READ (input logic[31:0] in,
                         output logic[31:0] out);
  typedef logic[31:0] T;
  `include "common_register_code.svh"

  function void write(T v);
    value = v;
  endfunction

  function T read();
    T tmp_value;
    tmp_value = value;
    value = 0;
    return tmp_value;
  endfunction
endinterface
```

Figure 8. Quirky register – clear-on-read register

Even more complicated Quirky registers such as ID Registers, Broadcast Registers, Coherent Registers and others are easily modeled with the new register model. It is not the ability of the the model that it so important, it is the simplicity of the implementation. Models strive to be "what-you-see-is-what-you-get" models; easy to write, easy to debug and easy to understand.

The broadcast control register contains handles to three targets. When the broadcast control write() routine is called, it calls the write() routine of the three targets in turn. The implementation is simple and clear, as shown in Figure 9.

```
interface BROADCAST_REGISTER(input  logic [31:0] in,
                             output logic [31:0] out);
  typedef logic [31:0] T;
  `include "common_register_code.svh"

  virtual REG32 b1a;
  virtual REG32 b1b;
  virtual REG32 b2;

  function void write(T v);
    value = v;
    b1a.write(v);
    b1b.write(v);
    b2.write(v);
  endfunction
endinterface
```

Figure 9. Quirky register – broadcast register

In order to implement Quirky Registers, access functions are used. Direct access is not recommended, since the quirky behavior may not be honored. A simple set of access routines is defined write(T v), T read(), reset() and backdoor_load(). These simple routines are easy to implement, and are complete in the register model, without a supporting library.

In Figure 10 a coherent register is implemented such that when the read() routine is called, the three target snapshot registers have their snapshot() routine called.

```
interface COHERENT_REGISTER(input  logic [31:0] in,
                            output logic [31:0] out);
  typedef logic [31:0] T;
  `include "common_register_code.svh"

  virtual SNAPSHOT_REGISTER c1a;
  virtual SNAPSHOT_REGISTER c1b;
  virtual SNAPSHOT_REGISTER c2;

  function T read();
    c1a.snapshot();
    c1b.snapshot();
    c2.snapshot();
  endfunction
endinterface
```

Figure 10. Quirky register – coherent register

In Figure 11 a snapshot register implementation is shown. The simple snapshot() routine simply saves the current value into a safe place – into the snapshot_value. This snapshot_value field can be used later along with other snapshot values to provide a consistent view of the current state of the snapshot registers. (They each have their snapshots taken at the same time, so that are consistent for the snapshot time). The snapshots provide a coherent view.

```
interface SNAPSHOT_REGISTER (input  logic [31:0] in,
                             output logic [31:0] out);
  typedef logic[31:0] T;
  `include "common_register_code.svh"
  T snapshot_value;

  function void snapshot();
    snapshot_value = value;
  endfunction

  function void write(T v);
    value = v;
  endfunction

  function T read();
    return value;
  endfunction
endinterface
```

Figure 11. Quirky register – snapshot register (works with coherent register)

ADDRESS MAPS

A register on the DUT is usually addressable with at least one address. These addresses change depending on many things. Furthermore a register can be accessed at a variety of addresses on a variety of interfaces. This complexity is easily modeled with the register model. A SystemVerilog class is created which contains a simple associative array. Each element of the array is indexed by a full path to the register. For example "top.block1.block2.b4.r6". At each element is a list of addresses, representing the addresses that a register can be accessed with. It is a simple list which can be directly or randomly accessed. All addresses can be exercised in any way practical. An address map is a simple lookup table. Given a register name, return the list of addresses it occupies. Pick randomly (or otherwise) from the list.

```
class blockA_address_map extends ADDRESS_MAP;

  virtual REGA rega;
  virtual REGB regb;

  virtual REG32 b1a;
  virtual REG32 b1b;
  virtual REG32 b2;

  virtual BROADCAST_REGISTER broadcast;

  virtual ID_REGISTER id_register;

  function void load_maps(int base);
    address_map[id_register.name].push_front(base+32'h00);

    address_map[rega.name].push_front(base+32'h04);
    address_map[rega.name].push_front(base+32'h84);

    address_map[regb.name].push_front(base+32'h08);
    address_map[regb.name].push_front(base+32'h88);

    address_map[broadcast.name].push_front(base+32'h10);
    address_map[b1a.name       ].push_front(base+32'h14);
    address_map[b1b.name       ].push_front(base+32'h18);
    address_map[b2.name        ].push_front(base+32'h1c);
  endfunction
```

Figure 12. Defining an address map

```
        address_mapA1 = new();
        address_mapA1.name = "address_mapA1";
        uvm_config_db#(ADDRESS_MAP)::
          set(null, "", "address_mapA1", address_mapA1);
        address_mapA1.rega = bA1.rega;
        address_mapA1.regb = bA1.regb;
        address_mapA1.id_register = bA1.id_register;
        address_mapA1.broadcast = bA1.broadcast;
        address_mapA1.b1a = bA1.b1a;
        address_mapA1.b1b = bA1.b1b;
        address_mapA1.b2 = bA1.b2;
        address_mapA1.load_maps(0);
        $display("DEBUG: Address Map = %s",
          address_mapA1.convert2string());
```

Figure 13. Filling the address map register handles

CONSTRAINTS

A constraint can be constructed between fields or between registers. It is very easy to define interesting constraints without any additional complexity. A simple constraint for legal values in fields is listed below.

```
    rand register_types_pkg::REGA_t rega_value;

    constraint rega_constraint {
      rega_value.a inside {[0:1]};
      rega_value.b inside {[0:7]};
    }
```

Figure 14. Simple field constraints – in one register

A slightly more interesting constraint, but a very important constraint is constraining two fields from two different registers. In the constraint in Figure 15, the field named 'a' in rega and regb must be the same.

```
    constraint rega_regb_constraint {
      rega_value.a == regb_value.a;
    }
```

Figure 15. Constraining fields between registers

FUNCTIONAL COVERAGE

Just as constraints are easy to add, functional coverage is easy to add. Each register field is easy to access. Each register is easy to access. Each address map entry is easy to access. In Figure 16, a covergroup is written to cover field 'a', 'b' and the cross between 'a' and 'b'.

```
    covergroup cg_REGA_t with function sample(REGA_t value);
      cp_a: coverpoint value.a;
      cp_b: coverpoint value.b;
      x_ab: cross cp_a, cp_b;
    endgroup
```

Figure 16. Functional coverage for registers and fields

In Figure 17, code snippets provide the covergroups and their construction, along with a simple routine to print the coverage as simulation progresses.

```
cg_REGA_t cg_REGA;
cg_REGB_t cg_REGB;

rand register_types_pkg::REGA_t rega_value;
rand register_types_pkg::REGB_t regb_value;
function new(string name = "register_sequence");
  super.new(name);
  cg_REGA = new();
  cg_REGB = new();
endfunction

function void print_coverage_report();
  $display("COVERAGE: %0d%% (%0d%%, %0d%%, %0d%%) ",
    cg_REGA.get_inst_coverage(), cg_REGA.cp_a.get_inst_coverage(),
      cg_REGA.cp_b.get_inst_coverage(), cg_REGA.x_ab.get_inst_coverage());
  ...
endfunction
```

Figure 17. Constructing the covergroups

In Figure 18 an example is shown of starting threads to monitor any register change. When a change occurs, the register covergroup sample() routine is called. Many different reasons exist that might cause a register sample function to get called. This is the simplest – any value change causes a sample to be collected.

```
task start_coverage_threads();
  fork
    forever begin
      @(bA.rega.dut_value);
      cg_REGA.sample(bA.rega.dut_value);
      print_coverage_report();
    end
    forever begin
      @(bA.regb.dut_value);
      cg_REGB.sample(bA.regb.dut_value);
      print_coverage_report();
    end
  join_none
endtask
```

Figure 18. Starting sample threads for covergroups

OTHER QUIRKY REGISTERS

Quirky registers are beasts. They can be hard to write, and hard to verify. Given a certain definition they require writing code in order to capture their behavior. They may need visibility into the address map. They may need visibility to other registers in their register blocks or some other arbitrary block. They can be complicated. Describing them must be simple.

ID REGISTER

An ID register is unique per "functional block". When this register is read(), each successive read() returns the next value from a list of values. After the last value in the list is returned, the next value read is the first. This ID is used to identify blocks in a system uniquely. This register is never written.

COHERENT REGISTER

A Coherent register is a register with causes other registers to take a snapshot of themselves for later use. For example, when the coherent "control" register is READ, the other "dependent" register values are snapshotted. This way a consistent view of the register space can be saved. This is important for registers that change values quickly, or when registers change values in response to other registers being read.

BROADCAST REGISTER

A broadcast register is simple. When you write to the broadcast control register, then that value is in turn written to each of the "dependent" registers. This functionality is useful to have a register value propagate to a list of registers quickly.

BUSES AND REGISTERS

In order to have a register read or write occur, an underlying bus read or write must occur. In our UVM testbench, this means that some kind of supporting bus sequences must be running. Our test starts the bus sequences.

```
fork
  seq1.start(e1.sqr);
  seq2.start(e2.sqr);
join_none
```

Figure 19. Starting bus sequences

Then the test starts the register sequences. The register sequences have a proxy handle to the underlying supporting bus sequence. Each supporting bus sequence has a read() and write() routine which creates a bus transaction and interacts with the driver.

```
register_seq1.seq = seq1;
register_seq2.seq = seq2;
fork
  register_seq1.start(e1.sqr);
  register_seq2.start(e2.sqr);
join
```

Figure 20. Starting register sequences

The bus interface sequence and the read() and write() tasks that the register sequence will call. The simple interface between registers and bus protocol is just address and data (see Figure 21).

How do we get to the simple register address? By doing an address lookup in the address map. This code is what-you-see-is-what-you-get. First an address for the register is retrieved using 'get_address()' That result is passed in to the bus interface write() routine, along with the value to write. This causes a write to occur on the bus.

```
class write_read_sequence extends uvm_sequence#(bus_transaction);
  `uvm_object_utils(write_read_sequence)
  ...

  bus_transaction t;

  task read(bit[31:0]addr, output bit[31:0]data);
    if (t == null)
      t = bus_transaction::type_id::create("bus_transaction");
    t.rw = 1; // Read
    t.addr = addr;
    t.data = 0; // Clear old data.
    start_item(t);
    finish_item(t);
    data = t.data;
  endtask

  task write(bit[31:0]addr, bit[31:0]data);
    if (t == null)
      t = bus_transaction::type_id::create("bus_transaction");
    t.rw = 0; // Write
    t.addr = addr;
    t.data = data;
    start_item(t);
    finish_item(t);
  endtask

  bit m_done = 0; // Never set to 1 in this code.

  ...

  task body();
    wait(m_done == 1); // Keep alive.
  endtask
endclass
```

Figure 21. Bus interface – register helper sequence

In the next step, an address is looked up, and a read is performed. Wdata and rdata now contain the values written and read – available for checking or other uses.

```
    seq.write(address_map.get_address(bA.rega.name), wdata);
    seq.read (address_map.get_address(bA.rega.name), rdata);
```

Figure 22. Writing and reading a register

WRITING TESTS

A register test can be simple – write to the register and read back the value. Or a register test can be complicated – write to the register and read back the value from a different address. Writing and reading register values are simple tests – block level tests. They can be repeated as blocks are consumed into higher level blocks, but that test is more like an address mapping test, instead of a register functionality test.

Checking a clear-on-read register consists of clearing it, then predicting the expected value (this register is a clock counter). Then we issue a read() and check the value. This kind of register can be hard to test, but the simple register model is allowing us to have easy access to all the pieces we need (the clk, the address map and the register value)

```
int count = 0;
// Clear the register.
seq.read(address_map.get_address(bA.clear_on_read.name), rdata);
repeat(10) begin
  bA.clear_on_read.write(count++); // Calculate expected value. Clock counter.
  bfm.wait_for_posedge_clk();
end
count += 2; // account for transfer delay.
seq.read(address_map.get_address(bA.clear_on_read.name), rdata);
if (rdata != count)
  `uvm_info(get_type_name(), $sformatf("%s: Mismatch. Expected %x, read %x",
      bA.clear_on_read.name, count, rdata), UVM_MEDIUM)
```

Figure 23. Testing the clear-on-read register

The UVM Register package defines a simple set of built-in tests; walking-ones and walking-zeroes; reset values; address map access; and shared address map access. These simple tests are valuable tests, but in a normal register map, there are many special registers (Quirky Registers) which do not do well when having these kinds of tests read or write them, so they must be turned off. Instead of trying to apply simple, general tests to all registers we should be writing tests which test all registers according to their capabilities in a simple but complete way, and avoid turning any registers off for such tests.

Write to the broadcast register, read from the broadcast targets; the CHECK macro is just short-hand for comparing 'wdata' and 'rdata'. The front-door data written and read, respectively.

```
wdata = 1;
for (int i = 0; i < 5; i++) begin
  bA.broadcast.write(wdata);
  seq.write(address_map.get_address(bA.broadcast.name), wdata);

  seq.read (address_map.get_address(bA.b1a.name), rdata);
  `CHECK(bA.b1a.name, wdata, rdata);
  seq.read (address_map.get_address(bA.b1b.name), rdata);
  `CHECK(bA.b1b.name, wdata, rdata);
  seq.read (address_map.get_address(bA.b2.name), rdata);
  `CHECK(bA.b2.name, wdata, rdata);
  wdata++;
end
```

Figure 24. Testing the broadcast register – front door

As we develop tests, as we think about their details, we know what registers are being tested. We know what is legal and what is not legal. A test can be written which is random, yet legal according to register permissions. Register access testing is important. We want to test that each legal address for a register actually reaches the register. This kind of test can be accomplished by writing from the front-door, and reading from the back-door. Then writing from the back-door and reading from the front-door. The new register modeling philosophy make back-door reading and back-door writing almost free of complexity.

```
function void backdoor_write();
  /* Copy randomized values into the registers */
  bA.rega.value = rega_value;
  bA.regb.value = regb_value;
  -> bA.rega.BACKDOOR_LOAD;
  -> bA.regb.BACKDOOR_LOAD;
endfunction
```

Figure 25. Writing values using the back door

To check using the back door, write to the broadcast register model, then issue a front-door write to the register, finally do a simple call to the built-in function called dut_check(); this is a back-door check. This will compare the automatically monitored dut_value to be compared with the value field.

```
wdata = 1;
for (int i = 0; i < 5; i++) begin
  bA.broadcast.write(wdata);
  seq.write(address_map.get_address(bA.broadcast.name), wdata);
  bA.b1a.dut_check();
  bA.b1b.dut_check();
  bA.b2.dut_check();
  wdata++;
end
```

Figure 26. Reading from the back door

SCOREBOARDING

Scoreboarding is about making sure things are functioning correctly. A scoreboard might check to see if the register is functioning correctly. Many times the correct functionality of a register is already "pre-checked", meaning that at a block level register behavior was checked as correct. When this block is re-used at a higher level, there is nothing that might change that behavior, so modeling the expected value and checking it against the actual value is not required. In this case many levels of efficiency can be achieved. Even though block checking may have given approval for turning off checking at higher levels, it might still be turned on for a nervous verification lead.

Besides register functionality, register addresses need to be checked. Did the correct register get addressed? Address map checking is usually hard, since address maps are large, but one check is to simply iterate all addresses that are allowed for a register, and see if a read returns a proper value. Address map checking can more economically be performed by checking all registers, checking the start and end of the address map and a few addresses in between.

Finally, a scoreboard may be interested in some "end-to-end" functionality, for example, setting a configuration – a collection of registers, and then running until that configuration has completed. For example, an encryption algorithm can be setup with register settings, and then 100,000 clocks can be run, and the end of which a new encryption value can be checked.

The simple register model allows for such scoreboard models and checks to be expressed easily and quickly. Their details go beyond the scope of this paper.

SUMMARY

The UVM Register package is popular, and powerful, but complex.

The new model introduced in this paper is simple and fast.

It provides much of the features of the UVM Register package with almost none of the complexity or overhead. The new model lacks certain features, which have useful if debatable value; trade-offs between the models are currently unexplored. As users adopt the UVM Register package they will ask themselves, "Is this new complexity warranted by the value added?" We have not seen this question answered yet.

We hope this paper will cause some discussion about the features needed for register tests versus the performance and complexity of the underlying model. We believe our model is almost trivially simple, has no required SystemVerilog support, nor required VPI support, and is transparent in its implementation. It is truly "what-you-see-is-what-you-get". And you can see it all in a short section of code.

This paper was originally presented at DVCon India 2015.

REFERENCES

[1] SystemVerlog LRM, http://standards.ieee.org/getieee/1800/download/1800-2012.pdf

[2] SystemVerilog UVM 1.1d, http://accellera.org/images/downloads/standards/uvm/uvm-1.1d.tar.gz

[3] Mark Litterick, "Advanced UVM Register Modeling", DVCON 2014, http://www.verilab.com/files/litterick_register_final_1.pdf

[4] Keisuke Shimizu, "UVM Tutorial for Candy Loves" Register Access Methods and Register Abstraction" http://cluelogic.com/2013/02/uvm-tutorial-for-candy-lovers-register-access-methods and http://cluelogic.com/2012/10/uvm-tutorial- for-candy-lovers-register-abstraction

[5] Tom Fitzpatrick, "Introduction to UVM Registers", http://www.mentor.com/products/fv/multimedia/intro-to-uvm-registers

[6] Various authors, "Registers", https://verificationacademy.com/cookbook/registers

[7] Tudor Timi, "Theory vs. Practice - Reserved Fields in UVM RAL", http://blog.verificationgentleman.com/2014/08/theory-vs-practice- reserved-fields-in.html

[8] Tudor Timi, "Custom Field Access Policies in UVM RAL", http://blog.verificationgentleman.com/2014/04/custom-field-access- policies-in-uvm-ral.html