2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Register Testing – Exploring Tests, Register Model Libraries, Sequences and Backdoor Access

Testing efficiency, transparency and simplicity

Rich Edelman, Siemens EDA, Fremont, CA, US (rich.edelman@siemens.com)

***Abstract*—SystemVerilog [1] UVM [2] can present complexities for the novice user. Register testing can be hard to understand and non-transparent. This paper explores the register modeling concepts and maps them into simpler and more transparent implementations, while continuing to provide efficient solutions. This discussion is not to supplant the UVM register models, but rather to teach modeling concepts and practice that can be applied to register testing and many other kinds of testing.**

Keywords—SystemVerilog, UVM, Testbench, register models, register testing, UVM Registers, address maps, UVM sequences, Register Sequences

I. INTRODUCTION

Register testing provides a simple test model. Write a register. Read it. Did it match? It's more complicated than that, but the idea is nicely simple. There are a collection of bits which are accessible using an address. Those bits can be tested by reading and writing them in certain ways.

Assuming the availability of a simple register base class, models with specific behavior can be built. Those models can be incorporated in a test environment.

Simplicity, transparency, and efficiency are the goals that this paper will explore. These goals are perhaps all the same – less code. With less code, each of the goals are achievable. The code becomes readable, predictable and debuggable. This paper will explore how to write simple code that still achieves large objectives. These concepts can be used in any modeling or test situation. The concepts won't be specific to register modeling or testing.

II. FANCY REGISTERS

A simple register model with a small API including, READ, WRITE, PEEK and POKE is assumed.

READ and WRITE implement special behavior. They have implementation specific behavior that are important modeling decision. For example, the CLEAR ON READ register sets its value to 0 when read.

PEEK and POKE are simply bit set and bit get. They move bits without regard to special behavior or access permissions. They set and get the raw bits.

The register models that will be built will use this base class, and implement those routines as needed.

A CLEAR-ON-READ register could be built as

```
class clear_on_read extends register;
  function T read(...);
    T return_value;
    return_value = value;
    value = 0;
    return return_value;
  endfunction
endclass
```

Where, 'T' is a bit vector or struct.

An index register could be built as

```
class index_register extends register;
  function T read(...);
    return value++;
  endfunction
  function write (...);
    // NOP Write. Read-only Register
  endfunction
endclass
```

An index register with a maximum limit could be built as

```
class index_register_max extends index_register;
  int max_value;
  function T read(...);
    if (value >= max_value)
      return value;
    else
      return super.read(...);
  endfunction
endclass
```

The issue of modeling is important. The models are important aspects of the IP of the design. These registers behave in special ways. It might be tempting to put these behaviors into a standard library or in the base class. But really, the behaviors and treatments are very specialized – and would unduly complicate the base class library.

Please see [3] for details on other interesting models.

The models are the models. They are not part of the test or the sequences. They don't automatically start any bus transactions. The register model is simple enough to fit on one printed page. See Appendix I

.

III. SEQUENCES

Sequences are really programs or tests or scenarios. A register sequence might be to write a value using the front door, and then read it and check using the back door. Sequences cause activity. These sequences are general "behavioral" sequences. In addition, sequences can have specific characteristics due to the interface or bus they are connected to. For example, an AXI sequence might have some special characteristics of the AXI protocol. Connecting these two kinds of sequences is key to reusable sequences and to making reuse easy. The connection is still a work in progress. It is implemented as a "master" sequence starting a lower level "worker" sequence. In this case the master sequence is a register sequence, and the worker sequence is a bus sequence. They are running on the sequencer for the bus. The register sequence can be run on any sequencer.

A sequence can expect to issue a 'start_item' and a 'finish_item'. That's the simple way to send a transaction to the driver.

The register sequence calls the special new interface that must be added to the bus sequence, start_reg_item() and finish_reg_item(). See the examples below.

A. Integrating sequences

Given a simple READ/WRITE kind of sequence for a bus protocol or other communication that is managed by a sequence/sequencer/driver triple, a new sequence needs to be created. This new sequence can extend the simple rw_sequence or it can simply be a uvm_sequence parameterized for a bus transaction. In the code below 'bus_transaction' is a bus transaction defined in the sequence/sequencer/driver code.

A register-to-bus translation sequence is created (one per bus protocol or bus transaction type):

```
class REGISTERtoBUS_rw_sequence extends uvm_sequence#(bus_transaction);
  `uvm_object_utils(REGISTERtoBUS_rw_sequence)
```

```
  bit all_done;

  task body();
    `uvm_info(get_type_name(), "...running", UVM_MEDIUM)
    all_done = 0;
    wait (all_done == 1);
  endtask
```

The body(), above, of this sequence simply runs as a "zombie" doing the bidding of others. It gets started by the actual register sequence and ends when it gets signaled by 'all_done'.

```
  task start_reg_item(register_transaction register_tr);
    bus_transaction bus_tr;
    // Translate to BUS
    bus_tr = new("bus_tr");
    bus_tr.rw = register_tr.rw;
    bus_tr.addr = register_tr.addr;
    bus_tr.data = register_tr.data;
    start_item(bus_tr);
    register_tr.bus_tr = bus_tr; // Save it for later.
    // Translate back to register
  endtask
```

The start_reg_item() task above and the finish_reg_item() task below provide a place for any translation between register transactions and bus transactions. There's no new technology here, just arranging bytes or bits and then starting or finishing a bus transaction.

```
  task finish_reg_item(register_transaction register_tr);
    bus_transaction bus_tr;
    // Translate to BUS
    $cast(bus_tr, register_tr.bus_tr);
    bus_tr.rw = register_tr.rw;
    bus_tr.addr = register_tr.addr;
    bus_tr.data = register_tr.data;
    finish_item(bus_tr);
    register_tr.data = bus_tr.data;
    // Translate back to register
  endtask
endclass
```

B. Built-in integration

A register transaction is provided to hold the register name, address and value, as well as READ or WRITE. It is a simple container used as a go-between from the register layer to the bus transaction layer.

```
class register_transaction extends uvm_sequence_item;
  `uvm_object_utils(register_transaction)

  bit [1:0] rw;
  string    register_name;
  bit[31:0] addr;
  bit[31:0] data;

  uvm_sequence_item bus_tr;

  function string convert2string();
    return $sformatf("Register: %s : rw=%0b, addr=%3d, data=%b",
     register_name, rw, addr, data);
  endfunction
endclass
```

A register-to-bus translation sequence is provided – a register sequence:

```
class REGISTER_SEQUENCE#(type SEQUENCE = REGISTERtoBUS_rw_sequence)
       extends uvm_sequence#(uvm_sequence_item);
    `uvm_object_utils(REGISTER_SEQUENCE)

  SEQUENCE seq;
```

The REGISTER_SEQUENCE is parameterized with the class type of the underlying bus sequence. It declares and handle and constructs such a sequence.

```
  task startup();
    seq = new("seq");
    fork
      seq.start(m_sequencer);
    join_none
    #0;
  endtask
```

The startup() task above creates the lower-level bus transaction and starts it. Remember, this is the "zombie" helper sequence that the register sequence will use to start and finish items.

```
  task body();
    register_address_map address_map;
    register_base register;

    register_transaction register_tr;

    string path;
    uvm_hdl_data_t backdoor_value;
    bit [31:0] bv;
    bit [31:0] addr;
    int i;

    startup();

    `uvm_info(get_type_name(), "...running", UVM_MEDIUM)
    i = 0;
    if (address_map == null) begin
      `uvm_info(get_type_name(), "No ADDRESS MAP", UVM_MEDIUM)
      seq.all_done = 1;
      return;
    end

    foreach (address_map.registers[addr]) begin

      register = address_map.registers[addr];

      register_tr = new($sformatf("register_tr%0d", i++));
      register_tr.register_name = register.name;
      register_tr.addr = addr;

      `uvm_info(get_type_name(),
         $sformatf("Issuing Register transaction %p", register_tr), UVM_MEDIUM)

      for (int j = 0; j < 32; j++) begin // Writes 01010101... patterns
        register_tr.data[j++] = 1;

        register_tr.rw = 0;              // WRITE
        seq.start_reg_item(register_tr);
```

```
        seq.finish_reg_item(register_tr);

        register_tr.rw = 1;              // READ
        seq.start_reg_item(register_tr);
        seq.finish_reg_item(register_tr);
      end
    end
    seq.all_done = 1;
  endtask
endclass
```

C. *A Type Specific sequence*

The code below shows three ways to manipulate the register model. The register values can be set directly, or write() can set or poke() can set.

```
typedef struct packed {
    bit [15:0] a;
    bit [15:0] b;
} split_word_T;

register#(split_word_T) reg1_1; // A register handle
split_word_T test_values;       // Some variable to set fields in
```

Fancy struct is defined with a register and a variable for use in tests.

```
register_tr = new($sformatf("register_tr"));
register_tr.register_name = r.name;
register_tr.addr = addr;

// Set the model and DUT with ONE of the BELOW methods
// 1. Set the values
r.value.a = 15;
r.value.b = 7;

// 2. Set a variable (field a and field b). Use poke() to set the model.
test_values.a = 15;
test_values.b = 7;
r.poke(test_values);

// 3. Set a variable (field a and field b). Use write() to set the model.
r.write(test_values);

register_tr.data = b.peek(); // Sample the model to fill the register transaction
```

With the register transaction filled in, it can be executed onto the bus sequence using start_reg_item() and finish_reg_item().

```
seq.start_reg_item(register_tr); // Front-door write
seq.finish_reg_item(register_tr);
```

IV. PUTTING IT TOGETHER IN A TEST

A simple test is built with a DUT having two interfaces (both the same). Those two interfaces share the same address map.

```
class test extends uvm_test;
  `uvm_component_utils(test)
```

```
env e1;
REGISTER_SEQUENCE         register_seq1;

env e2;
REGISTER_SEQUENCE         register_seq2;
```

Above, each interface will have an ‘environment’ and a register sequence.

```
register_model reg_model;

function void build_phase(uvm_phase phase);
  e1 = new("e1", this);
  e2 = new("e2", this);

  if (!uvm_config_db#(virtual simple_bus)::get(this, "*", "busA", e1.bus))
    `uvm_fatal(get_type_name(), "BUS LOOKUP FAILED for e1 (busA)")
  if (!uvm_config_db#(virtual simple_bus)::get(this, "*", "busB", e2.bus))
    `uvm_fatal(get_type_name(), "BUS LOOKUP FAILED for e1 (busB)")

  reg_model = new();
endfunction
```

This is just UVM. Build the environments and get the virtual interfaces (the bus handles for the testbench to wiggle). The code below starts two register sequences in parallel and assigns the address map to use for that sequence.

```
task run_phase(uvm_phase phase);
  phase.raise_objection(this);
  fork
    begin
      register_seq1 = new("register_seq1");
      register_seq1.address_map = reg_model.address_map;
      register_seq1.start(e1.sqr);
    end
    begin
      register_seq2 = new("register_seq2");
      register_seq2.address_map = reg_model.address_map;
      register_seq2.start(e2.sqr);
    end
  join
  phase.drop_objection(this);
endtask
endclass
```

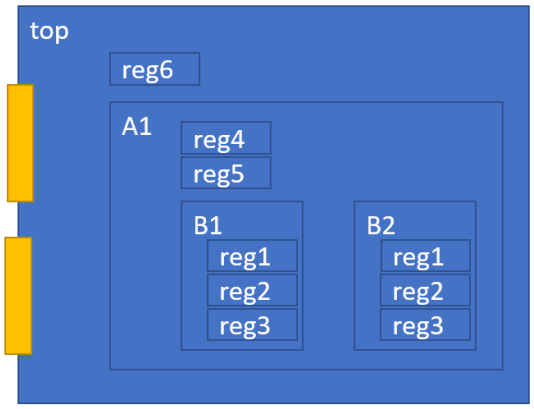

Figure 1- Simple DUT - Block Diagram

V. BUILDING A REGISTER MODEL

This is a trivial register model with uninteresting registers. Only one has fields. Keep it simple helps with this paper, and a more complete example is planned later. See the paper from DVCON Japan to see interesting models.

The block diagram is above. There are two interfaces (yellow) and 9 total registers. The 'register_model' below builds the model representing these registers.

```
typedef register#(bit[31:0]) simple_register;

typedef struct packed {
    bit [15:0] a;
    bit [15:0] b;
  } split_word_T;

class register_model;
  register#(split_word_T) reg1_1;

  simple_register reg1_2;
  simple_register reg1_3;
  simple_register reg4;
  simple_register reg5;
  simple_register reg6;
  register#(split_word_T) reg2_1;
  simple_register reg2_2;
  simple_register reg2_3;
```

The registers are declared above. Then the address map is declared and the constructor builds everything – registers and address map.

```
  register_address_map address_map;

  function new();

    reg1_1 = new(); reg1_1.name = "reg1"; reg1_1.backdoor_name = "top.dutA.A1.B1.reg1";
    reg1_2 = new(); reg1_2.name = "reg2"; reg1_2.backdoor_name = "top.dutA.A1.B1.reg2";
    reg1_3 = new(); reg1_3.name = "reg3"; reg1_3.backdoor_name = "top.dutA.A1.B1.reg3";

    reg4   = new(); reg4.name   = "reg4";   reg4.backdoor_name = "top.dutA.A1.reg4";
    reg5   = new(); reg5.name   = "reg5";   reg5.backdoor_name = "top.dutA.A1.reg5";
    reg6   = new(); reg6.name   = "reg6";   reg6.backdoor_name = "top.dutA.reg6";

    reg2_1 = new(); reg2_1.name = "reg1"; reg2_1.backdoor_name = "top.dutA.A1.B2.reg1";
    reg2_2 = new(); reg2_2.name = "reg2"; reg2_2.backdoor_name = "top.dutA.A1.B2.reg2";
    reg2_3 = new(); reg2_3.name = "reg3"; reg2_3.backdoor_name = "top.dutA.A1.B2.reg3";

    address_map = new();
    address_map.name = "address_map";

    address_map.add(reg1_1,  0);
    address_map.add(reg1_2,  4);
    address_map.add(reg1_3,  8);
    address_map.add(reg4,   12);
    address_map.add(reg5,   16);
    address_map.add(reg1_3, 20);
    address_map.add(reg1_3, 24);
    address_map.add(reg1_3, 28);
    address_map.add(reg6,   32);
```

The registers are constructed, and the backdoor names are filled in, along with the short names. Then the address map is constructed, and the registers are added by handle and address.

```
    foreach (address_map.registers[addr]) begin // Pretty print
       register_base r;
       r = address_map.registers[addr];
      $display("Addr: %0d, register=%s", addr, r.convert2string());
    end
  endfunction
endclass
```

VI. BACKDOOR ACCESS

Most register accesses should be issued across the "regular" interfaces – across the bus interface. These are front door accesses. But the problem with front door accesses is that they require bus signaling and simulation time. To transfer 100 register values across the bus might take hundreds or thousands of clock cycles. And many seconds or minutes of wall clock time.

Backdoor access, on the other hand, could "write" 100 registers with 100 writes to memory using DPI to access to the RTL registers. This operation takes no appreciable wall clock time. Entire banks of RTL register values can be loaded or read in almost no time. This is a very efficient way to test, keeping in mind that this backdoor access isn't part of normal operation. Testing the normal operation of the front door is also important – issuing bus cycles that consume time.

A. *Example*

The code below demonstrates using the raw UVM functions uvm_hdl_read() and uvm_hdl_deposit() to set and get a value from the DUT that has a path name. For example, the line below reads the 'top.dutA.A1.B1.reg1 value.

```
uvm_hdl_read("top.dutA.A1.B1.reg1", backdoor_value)
```

This statement sets the variable 'backdoor_value' to the value of the "register" in the DUT named 'top.dutA.A1.B1.reg1'. It gets the value from the DUT and puts it in a variable in the testbench.

The following code reads from the backdoor, then writes an updated value and reads again – all using the backdoor.

```
// READ
path = register.backdoor_name;
if (!uvm_hdl_read(path, backdoor_value)) begin
  `uvm_fatal(get_type_name(), {"Backdoor READ FAILED on ", path});
end
bv = backdoor_value;
`uvm_info(get_type_name(),
  $sformatf("BACKDOOR READ (%s) = %b", path, bv), UVM_MEDIUM)

// WRITE
bv = bv+2;
path = register.backdoor_name;
if (!uvm_hdl_deposit(path, bv)) begin
  `uvm_fatal(get_type_name(), {"Backdoor DEPOSIT FAILED on ", path});
end

// Re-READ
path = register.backdoor_name;
if (!uvm_hdl_read(path, backdoor_value)) begin
  `uvm_fatal(get_type_name(), {"Backdoor READ FAILED on ", path});
end
bv = backdoor_value;
`uvm_info(get_type_name(),
  $sformatf("BACKDOOR READ (%s) = %b", path, bv), UVM_MEDIUM)
```

Checking values could be included, or updating the register model, or other things. These UVM routines are very easy to use and provide great value for backdoor testers.

B. *Simple read/write/check test*

The code below issues a front-door write and then a backdoor read, and a simple compare.

```
register_tr.rw = 0;                 // Front door write
seq.start_reg_item(register_tr);
seq.finish_reg_item(register_tr);

uvm_hdl_read(path, backdoor_value); // Backdoor read

// Simple self-checking
if (backdoor_value != register_tr.data) begin
  `uvm_info(get_type_name(),
     $sformatf("Error: Mismatch. %b written, %b read from backdoor",
     register_tr.data, backdoor_value), UVM_MEDIUM)
end
```

VII. CONCLUSION

The reader of this paper will come away with trade-offs and background to make their own decisions about creating models for any purpose – including register testing. The examples in this paper are available open source for any use (Apache license).

The register model and tests developed here are not intended as a replacement for the UVM register model, but rather as a mechanism to discuss writing good models, keeping in mind, efficiency, transparency, and simplicity. Keeping these concepts front and center improves productivity and reduces debug time.

The discussion here is not complete. It will be continued, please stay tuned for future updates. Some of the code became complex in order to be explained and understood - it will be wrapped into utility functions. But it is available unwrapped in this paper for examination.

There are some odd coding requirements for extended classes and casting that need to be verified and hopefully improved. These limitations are due to type compatibility rules which appear to be quite restrictive, even when parameterized classes and their base cases are used in compatible ways.

Enjoy your modeling and thinking about clarity and ease of writing and ease of debug.

VIII. REFERENCES

[1] SystemVerilog LRM, “1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language”, https://ieeexplore.ieee.org/document/8299595

[2] UVM 1.1d - https://www.accellera.org/downloads/standards/uvm

[3] *“Register Modeling – Exploring Fields, Registers and Address Maps”*, Rich Edelman, DVCON Japan 2022. Contact author for paper and presentation

[4] *“What Does The Sequence Say? Powering Productivity with Polymorphism”*, Rich Edelman, DVCON US 2022

I. APPENDIX I – REGISTER MODEL

The register library base classes for modeling registers and address maps from DVCON Japan 2022 with some small adjustments.

```
package register_pkg;

  typedef bit [31:0] BV_T;

  typedef bit [31:0] address_t;
  typedef class register_address_map;

  // -------------------------------------------------
  virtual class register_base;
    string name;
    string backdoor_name;
    register_address_map address_maps[address_t];

    virtual function void add(register_address_map am,
                              address_t address);
      address_maps[address] = am;
    endfunction

    virtual function string convert2string();
      return {"No convert2string defined for ", name};
    endfunction

    virtual function void reset();
    endfunction

    pure virtual function BV_T read();
    pure virtual function BV_T peek();
    pure virtual function void poke(BV_T v);
    pure virtual function void write(BV_T v);
  endclass

  // -------------------------------------------------
  class register #(type T) extends register_base;
    T reset_value;
    rand T value;

    typedef bit[31:0] TT;

    virtual function void reset();
      poke(reset_value);
    endfunction

    virtual function BV_T read();
       return value;
    endfunction

    virtual function void write(BV_T v);
       value = v;
    endfunction

    virtual function BV_T peek();
      return value;
    endfunction

    virtual function void poke(BV_T v);
       value = v;
    endfunction
```

```
    virtual function T backdoor_peek();
      // Get the RTL value;
      string examine_string;
      examine_string =
        $sformatf("examine %s",
        backdoor_name);
      $display("INFO: %s", examine_string);
      mti_fli::mti_Command(examine_string);
      return value;
    endfunction

    virtual function void backdoor_poke(T v);
      // Set the RTL value;
      string force_string;
      value = v;
      force_string = $sformatf("force -deposit %s %x",
        backdoor_name, v);
      $display("INFO: %s", force_string);
      mti_fli::mti_Command(force_string);
    endfunction

    virtual function string convert2string();
      string addresses;
      addresses = "";
      foreach (address_maps[addr]) begin
        if (addresses == "")
          addresses = $sformatf("%0d", addr);
        else
          addresses = $sformatf("%s, %0d", addresses, addr);
      end
      if (addresses == "")
        addresses = "<None>";
      return $sformatf("Register '%s' value=%p, backdoor=%s [addr: %s]", name, peek(),
backdoor_name, addresses);
    endfunction
  endclass

  // -----------------------------------------------
  class register_address_map;
    string name;
    register_base registers[address_t];
     register_base registers_by_name[string];

    virtual function void add(register_base r,
                              address_t address);
       registers_by_name[r.name] = r;
      registers[address] = r;
      r.add(this, address);
    endfunction
  endclass
endpackage
```

II. APPENDIX II – SEQUENCES AND TESTS

The main focus of this paper. Sequences.

```
package tb_pkg;

import uvm_pkg::*;
`include "uvm_macros.svh"

import simple_bus_pkg::*;
import register_pkg::*;

typedef register#(bit[31:0]) simple_register;

class register_transaction extends uvm_sequence_item;
  `uvm_object_utils(register_transaction)

  bit [1:0]rw;
  string register_name;
  bit[31:0] addr;
  bit[31:0] data;

  uvm_sequence_item bus_tr;

  function new(string name = "register_transaction");
    super.new(name);
  endfunction

  function string convert2string();
    return $sformatf("Register: %s : rw=%0b, addr=%3d, data=%b",
      register_name, rw, addr, data);
  endfunction
endclass


class REGISTERtoBUS_rw_sequence extends uvm_sequence#(bus_transaction);
  `uvm_object_utils(REGISTERtoBUS_rw_sequence)

  function new(string name = "REGISTERtoBUS_rw_sequence");
    super.new(name);
  endfunction

  bit all_done;

  task body();
    `uvm_info(get_type_name(), "...running", UVM_MEDIUM)
    all_done = 0;
    wait (all_done == 1);
  endtask

  task start_reg_item(register_transaction register_tr);
    bus_transaction bus_tr;
    // Translate to BUS
    bus_tr = new("bus_tr");
    bus_tr.rw = register_tr.rw;
    bus_tr.addr = register_tr.addr;
    bus_tr.data = register_tr.data;
    start_item(bus_tr);
    register_tr.bus_tr = bus_tr; // Save it for later.
    // Translate back to register
  endtask

  task finish_reg_item(register_transaction register_tr);
    bus_transaction bus_tr;
```

```
    // Translate to BUS
    $cast(bus_tr, register_tr.bus_tr);
    bus_tr.rw = register_tr.rw;
    bus_tr.addr = register_tr.addr;
    bus_tr.data = register_tr.data;
    finish_item(bus_tr);
    register_tr.data = bus_tr.data;
    // Translate back to register
  endtask
endclass

  typedef struct {
    bit [packed 15:0] a;
    bit [15:0] b;
  } split_word_T;

  typedef register#(split_word_T) XT;

class REGISTER_SEQUENCE#(type SEQUENCE = REGISTERtoBUS_rw_sequence) extends
                                              uvm_sequence#(uvm_sequence_item);
    `uvm_object_utils(REGISTER_SEQUENCE)

  SEQUENCE seq;

  task startup();
    seq = new("seq");
    fork
      seq.start(m_sequencer);
    join_none
    #0;
  endtask

  function new(string name = "REGISTER_SEQUENCE");
    super.new(name);
  endfunction

  register_address_map address_map;

  task body();
    register_base reg_base;
    typedef bit[31:0] T;
    register #(T) reg_handle;

    register_transaction register_tr;

    string path;
    uvm_hdl_data_t backdoor_value;
    bit [31:0] register_model_value;

    bit [31:0] bv;
    bit [31:0] addr;
    int i;

    startup();
    `uvm_info(get_type_name(), "...running", UVM_MEDIUM)
    i = 0;
    if (address_map == null) begin
      `uvm_info(get_type_name(), "No ADDRESS MAP", UVM_MEDIUM)
      seq.all_done = 1;
      return;
    end

    if (address_map.registers_by_name.exists("reg1")) begin
      register_address_map rm1;
```

```
    register#(split_word_T) r; // Exact type
    register_base b;
    b = address_map.registers_by_name["reg1"];
    $cast(r, b);
    foreach (r.address_maps[addr]) begin
       rm1 = r.address_maps[addr];
       if (rm1 == address_map) begin
         split_word_T test_values;
         register_tr = new($sformatf("register_tr"));
         register_callback(register_tr);
         register_tr.register_name = r.name;
         register_tr.addr = addr; /* $urandom_range(32, 0); */

         r.value.a = 15;
         r.value.b = 7;

         test_values.a = 15;
         test_values.b = 7;
         r.poke(test_values);
         r.write(test_values);

         register_tr.data = b.peek(); // Sample the model

         seq.start_reg_item(register_tr); // Front-door write
         seq.finish_reg_item(register_tr);
       end
    end
  end

  foreach (address_map.registers[addr]) begin
    reg_base = address_map.registers[addr];
    register_tr = new($sformatf("register_tr%0d", i++));
    register_tr.register_name = reg_base.name;
    register_tr.addr = addr; /* $urandom_range(32, 0); */
    register_tr.addr[1:0] = 2'b00; // Divide by 4
    `uvm_info(get_type_name(),
      $sformatf("Issuing Register transaction %p", register_tr), UVM_MEDIUM)

    for (int j = 0; j < 32; j++) begin
      register_tr.data[j++] = 1;
      register_tr.rw = 0;              // WRITE

      reg_base.write(register_tr.data);   // base!

      seq.start_reg_item(register_tr); // Front-door write
      seq.finish_reg_item(register_tr);

      $cast(register_model_value, reg_base.peek()); // Sample the model
      uvm_hdl_read(path, backdoor_value);     // Backdoor read

      if (backdoor_value != register_model_value) begin
        `uvm_info(get_type_name(),
          $sformatf("Error: Mismatch. %b model, %b read from backdoor",
          register_model_value, backdoor_value), UVM_MEDIUM)
      end

      register_tr.rw = 1;              // READ
      seq.start_reg_item(register_tr);
      seq.finish_reg_item(register_tr);
    end

    // READ
    path = reg_base.backdoor_name;
    if (!uvm_hdl_read(path, backdoor_value)) begin
```

```
        `uvm_fatal(get_type_name(), {"Backdoor READ FAILED on ", path});
      end
      bv = backdoor_value;
      `uvm_info(get_type_name(),
        $sformatf("BACKDOOR READ (%s) = %b", path, bv), UVM_MEDIUM)

      // WRITE
      bv = bv+2;
      path = reg_base.backdoor_name;
      if (!uvm_hdl_deposit(path, bv)) begin
        `uvm_fatal(get_type_name(), {"Backdoor DEPOSIT FAILED on ", path});
      end

      // Re-READ
      path = reg_base.backdoor_name;
      if (!uvm_hdl_read(path, backdoor_value)) begin
        `uvm_fatal(get_type_name(), {"Backdoor READ FAILED on ", path});
      end
      bv = backdoor_value;
      `uvm_info(get_type_name(),
        $sformatf("BACKDOOR READ (%s) = %b", path, bv), UVM_MEDIUM)

    end
    seq.all_done = 1;
  endtask
endclass
```

III. APPENDIX III – REGISTER MODEL WITH ADDRESS MAP

A register model with simple registers.

```
class register_model;
  register#(split_word_T) reg1_1;

  simple_register reg1_2;
  simple_register reg1_3;
  simple_register reg4;
  simple_register reg5;
  simple_register reg6;
  register#(split_word_T) reg2_1;
  simple_register reg2_2;
  simple_register reg2_3;

  register_address_map address_map;

  register_base base_array[string];
  register_base b;

  function new();

    reg1_1 = new(); reg1_1.name = "reg1"; reg1_1.backdoor_name = "top.dutA.A1.B1.reg1";
    reg1_2 = new(); reg1_2.name = "reg2"; reg1_2.backdoor_name = "top.dutA.A1.B1.reg2";
    reg1_3 = new(); reg1_3.name = "reg3"; reg1_3.backdoor_name = "top.dutA.A1.B1.reg3";

    reg4   = new(); reg4.name   = "reg4";   reg4.backdoor_name = "top.dutA.A1.reg4";
    reg5   = new(); reg5.name   = "reg5";   reg5.backdoor_name = "top.dutA.A1.reg5";
    reg6   = new(); reg6.name   = "reg6";   reg6.backdoor_name = "top.dutA.reg6";

    reg2_1 = new(); reg2_1.name = "reg1"; reg2_1.backdoor_name = "top.dutA.A1.B2.reg1";
    reg2_2 = new(); reg2_2.name = "reg2"; reg2_2.backdoor_name = "top.dutA.A1.B2.reg2";
    reg2_3 = new(); reg2_3.name = "reg3"; reg2_3.backdoor_name = "top.dutA.A1.B2.reg3";

    address_map = new();
    address_map.name = "address_map";

    address_map.add(reg1_1,  0);
    address_map.add(reg1_2,  4);
    address_map.add(reg1_3,  8);
    address_map.add(reg4,   12);
    address_map.add(reg5,   16);
    address_map.add(reg1_3, 20);
    address_map.add(reg1_3, 24);
    address_map.add(reg1_3, 28);
    address_map.add(reg6,   32);

    foreach (address_map.registers[addr]) begin
       register_base r;
       r = address_map.registers[addr];
      $display("Addr: %0d, register=%s", addr, r.convert2string());
    end
  endfunction
endclass
```

IV. APPENDIX IV – A TEST

A simple test with two interfaces which share an address map running a few experiments in register modelling and behavior testing.

```
class test extends uvm_test;
  `uvm_component_utils(test)

  env e1;
  REGISTERtoBUS_rw_sequence          seq1;
  REGISTER_SEQUENCE         register_seq1;

  env e2;
  REGISTERtoBUS_rw_sequence          seq2;
  REGISTER_SEQUENCE         register_seq2;

  register_model reg_model;

  function new(string name = "test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    e1 = new("e1", this);
    e2 = new("e2", this);

    if (!uvm_config_db#(virtual simple_bus)::get(this, "*", "busA", e1.bus))
      `uvm_fatal(get_type_name(), "BUS LOOKUP FAILED for e1 (busA)")
    if (!uvm_config_db#(virtual simple_bus)::get(this, "*", "busB", e2.bus))
      `uvm_fatal(get_type_name(), "BUS LOOKUP FAILED for e1 (busB)")

    reg_model = new();
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    fork
      begin
        register_seq1 = new("register_seq1");
        seq1 = new("REGISTERtoBUS_seq1");
        register_seq1.seq = seq1;
        register_seq1.address_map = reg_model.address_map;
        register_seq1.start(e1.sqr);
      end
      begin
        register_seq2 = new("register_seq2");
        seq2 = new("REGISTERtoBUS_seq2");
        register_seq2.seq = seq2;
        register_seq2.address_map = reg_model.address_map;
        register_seq2.start(e2.sqr);
      end
    join
    phase.drop_objection(this);
  endtask
endclass

endpackage
```