BEYOND UVM REGISTERS: SCOREBOARDS AND REGISTER GENERATION

RICH EDELMAN AND YUXIN YOU — MENTOR, A SIEMENS BUSINESS

WHITEPAPER

Mentor®
A Siemens Business

FUNCTIONAL VERIFICATION

www.mentor.com

I. INTRODUCTION

Register verification techniques are a powerful way to create higher level tests and analysis about desired behavior. The barrier to entry for using this technology can be quite high with the existing solutions. This paper provides some background about previous work [3] exploring an alternative solution, and extends that work with some discussion about necessary infrastructure. In particular, it explores what is required for generation of lighter weight UVM Registers, and how tests and scoreboards might be constructed.

The alternative register implementation example

The ID Register is a simple quirky register. It is used on a chip or in a block as the ID of that chip or block. This implementation contains eight 32-bit words, which, taken together are the ID of the chip or block. Each time the register is "read", it returns the next 32-bit word in the list. Once 8 reads have occurred, then it repeats from the beginning of the list. In order to get the ID of the chip or block, this register must be read eight times.

The read() implementation is trivial. There is an array of eight 32-bit words, and an index pointer which is incremented for each read call. The index pointer wraps around at 'size'. These kinds of quirky registers can be hard to implement and hard to debug in other library packages. Using this alternative implementation it is easy to debug and easy to support – it is 24 lines of simple code.

```
 1 interface ID_REGISTER(logic [31:0]in, output logic [31:0]out);
 2   typedef logic [31:0] T;
 3
 4   `include "common_register_code.svh"
 5
 6   // ---------------------------------------
 7   // Special Code for the ID Register functionality
 8   // ---------------------------------------
 9   T values[8];
10   bit [2:0] index;
11   int size;
12
13   initial
14     index = 0;
15
16   function T read();
17     T v;
18     v = values[index];
19     index++;
20     if (index >= size)
21       index = 0;
22     return v;
23   endfunction
24 endinterface
```

Figure 1, below, is the same ID register code, this time with the `include file expanded in place in order to see the contents of the register and the complete infrastructure. It has a reset_value, a value and a dut_value. In addition, it has helper functions: convert2string, dut_check and check_reset. It also has an event that when triggered causes a backdoor load to happen. Finally, it has three threads, one acting as a monitor or change printer, one acting as an input monitor and one ready to do a backdoor load when triggered.

The reset value is set by configuration code and will be used as the value during a 'reset'. The value is a built-in way to hold the register value. It can be used if needed. The dut_value is like the value, but it represents the value in the DUT.

Because this register is 'bound' (Figure 2) into the hardware implementation anytime the DUT register changes, the input monitor is triggered, and the dut_value is updated. Dut_check() is a convenient routine to check the 'dut_value' versus the 'value'. Check_reset() is a convenient routine to check the 'reset_value' versus the 'value'.

```
 1 interface ID_REGISTER(logic [31:0]in, output logic [31:0]out);
 2   typedef logic [31:0] T;
 3
 4   // ---------------------------------------
 5   // Common Register Code.
 6   // ---------------------------------------
 7
 8   T reset_value;
 9   T value;     // Model value.
10   T dut_value; // Mirror of the DUT value.
11
12   // 1 - A name.
13   string name = $sformatf("REG: %m");
14
15   // 2 - A pretty printer (name, value)
16   function string convert2string();
17     return $sformatf("%s: %p [DUT: %p]",
18       name, value, dut_value);
19   endfunction
20
21   // 3 - A BACKDOOR_LOAD trigger
22   event BACKDOOR_LOAD;
23
24   // 4 - A value change printer
25   always @(value)
26     $display("%s", convert2string());
27
28   // 5 - A monitor.
29   always @(in)
30     dut_value = in;
31
32   // 6 - A backdoor load.
33   always @(BACKDOOR_LOAD)
34     out = value;
35
36   // 7 - A checker.
37   function bit dut_check();
38     if ( dut_value != value ) begin
39       $display("@%t: (%m) Mismatch. DUT_VALUE=%p, VALUE=%p",
40         $time, dut_value, value);
41       return 0;
42     end
43     return 1;
44   endfunction
45
46   function void check_reset();
47     if (reset_value != dut_value)
48       $display("@%t: (%m) Reset mismatch. DUT_VALUE=%p, RESET_VALUE=%p",
49         $time, dut_value, reset_value);
50   endfunction
51
52
53   // ---------------------------------------
54   // Special Code for the ID Register functionality
55   // ---------------------------------------
56   T values[8];
57   bit [2:0] index;
58   int size;
59
60   initial
61     index = 0;
62
63   function T read();
64     T v;
65     v = values[index];
66     index++;
67     if (index >= size)
68       index = 0;
69     return v;
70   endfunction
71 endinterface
```

Figure 1 – A complete ID register implementation

II. Previous Work [3]

Concepts

The fundamental concepts and goals for the lightweight registers are

Small memory footprint

No Library Source code

No VPI needed for backdoor

Simple, What-You-See-Is-What-You-Get implementation

Easy to build and understand Quirky registers

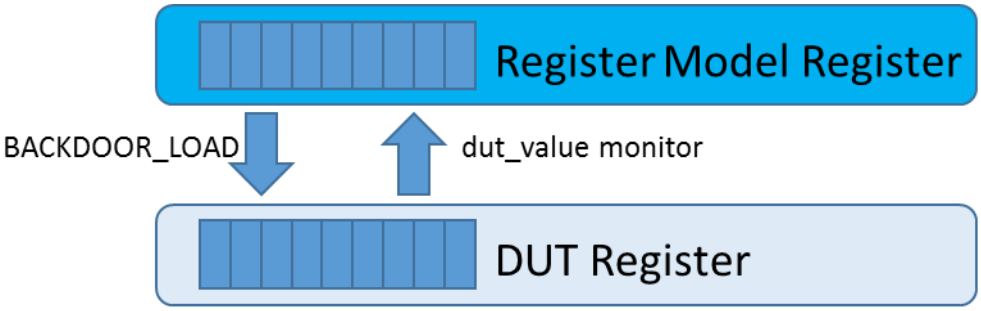


Figure 2 - DUT Register with Register Model Attached

Results

With the previous work [3], the goals have been met. A small, lighter weight register package with no library requirement has been created.

Limitations

There is currently no automatic generation. Since the implementation is based on interfaces, the lack of inheritance, makes managing the large list of registers harder. By adding common techniques, that problem can be minimized. [4]. See Section X Extensions for "Lists Of Registers" Functionality below for additional details.

III. Example Block Level Design

Figure 3 is a simple block – "Block A". This block will be verified "at the block level", likely with a register verification package, writing and reading the registers R1, R2 and R3, from the front-door and the backdoor.

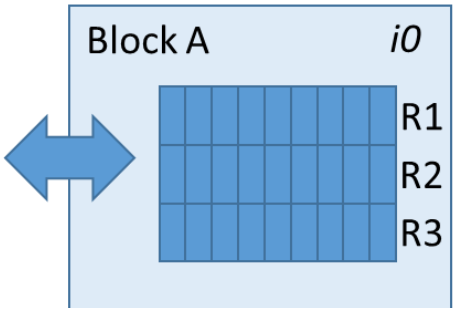


Figure 3 - Block A

Functional coverage will be operating, and results will be analyzed to determine correctness.

Figure 4 below is still a simple block – "Block B", but block B consists of two instances of Block A and a few additional registers. It will also likely be verified with a register verification package, writing and reading the 8 registers. Block B is a higher level block or a sub-system.

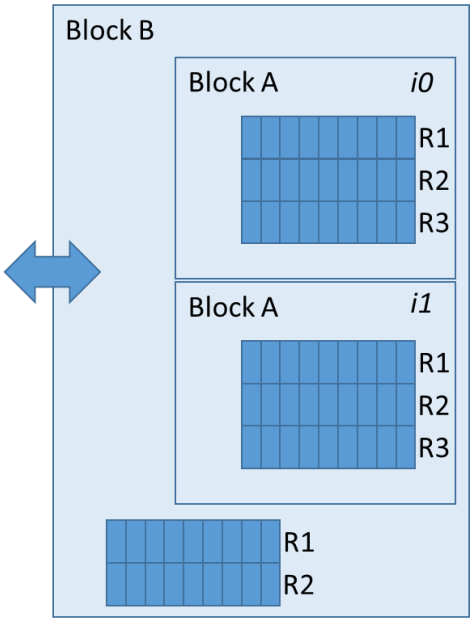


Figure 4 - Block B

Blocks and sub-systems are naturally verified with register library techniques. Writing tests and collecting coverage for register verification of blocks is quite common.

IV. SCOREBOARD

Definition

A scoreboard is used to check the expected results against the measured or actual results. In a register testing environment, a kind of self-checking scoreboard is built into the register model. Most register updates from the DUT are mirrored in a register model. The register model also contains a predictor to predict what the next value is.

At higher levels of abstraction a register scoreboard can be used in unique ways to configure the DUT and then later to evaluate the behavior. This kind of scoreboard is very application specific, but when used can test many parts of the DUT with a single test.

Checkers and Self-checking

A register model contains a description or model of its intended behavior. For example, a CLEAR-ON-READ register (or field) will know how to behave when it is READ.

```
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 5 - A CLEAR-ON-READ register

An ID register defined above (Figure 1) knows how to return the ID, one word at a time.

These functional descriptions are Quirky registers and must be easy to define and easy to debug. These kinds of registers generally have a harder time with automatic testing than simpler registers which just store values.

A simple storage register is not a quirky register, but just a “normal register”. It may have fields and certain Boolean behaviors like READ-ONLY, WRITE-ONLY, etc.

```
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 6 - Simple register with fields "REGA_t"

where REGA_T is defined simply as

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

Figure 7 - REGA_t field definitions

Simple storage register with two fields, a and b. B is read-only.

```
interface REGA_RO_B (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"
   // 'a' is writable. 'b' is NOT.
  T WRITE_MASK = '{a:'1, b:'0};

  function void write(T v);
    value = (v & WRITE_MASK) | value;
  endfunction

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

Figure 8 - Simple register with fields, one READONLY

Configuration of registers

Checking the behavior of a register is important. Its specific functionality must be checked. Another technique for checking takes a less fine-grained approach. It uses registers as configuration and status. The DUT can be configured by loading the registers with various values. Then simulation time can advance, performing some simple or complex function. Then the DUT registers can be read and other results analyzed. This process of configuring and checking is very powerful. The registers can be backdoor loaded, and then simulated and backdoor read, speeding the process even further. Of course, before relying on the backdoor path, it must be verified, along with the front-door path. Using the backdoor path is a simulation speedup trick.

V. BUILT-IN TESTS

Existing built-in UVM register tests

The UVM Register package has built in tests, really UVM sequences. They perform basic tasks: reset check, write ones and zeroes to every bit, write walking ones or walking zeroes. They can write a register from the front-door and read from the backdoor. And they can write from the backdoor and read from the front-door. For registers in multiple maps, they can write with one address map and read with others.

Necessary Tests

There are tests which are necessary. They answer the required questions, including:

Can I read all my bits and fields?

Can I write all my bits and fields?

Does the register "work"? Does any quirky behavior work as expected?

Are there any side-effects such as writing one register and affecting the contents of another?

Hard Tests

Register verification will consist of easy tests (read and write to a "regular register") and hard tests (check functionality in a quirky register). Hard tests may be written by hand, and executed either by hand or automatically. The hard tests are not hard to write, since the register model is quite easy to understand. This property of hard things becoming easy is a property of the new lightweight model. In the UVM Register package this is not the case. The hard things become harder as layers are added. In the proposed light weight model, the functionality of even a hard model is transparent.

What do we need?

We need to run all the easy tests and all the hard tests, and run them just once. After they are run, we have some confidence in the model, and can proceed to layer the blocks together and building a larger system. That larger system won't need the same level of functional tests, but our lightweight model will be able to help with monitoring if we need to debug register model values. The register model is not a "block level only" test infrastructure – it can help at system level too.

VI. Writing Tests Using Register Names

Writing tests using register names is very portable, and powerful. Automated tests can be used with certain registers, but it is always a useful exercise to write the tests yourself. It is simple and easy.

For Quirky registers, it is almost always required to write the test yourself. A specific functionality needs to be tested, one that is not easily automated – thus the requirement for the Quirky register.

In the test below, the basic test is repeated twice, since we need to know if it will work twice. The test loops through N times, reading a part of the ID each time. The sequence 'seq' causes a front-door read using the address looked up from the address map, placing the data read in 'rdata'. Then a simple compare for correctness.

```
repeat(2)
  for (int i = 0; i < bA.id_register.size; i++) begin
    expected_result = bA.id_register.read();
    seq.read(address_map.get_address(bA.id_register.name), rdata);
    if (expected_result != rdata)
      `uvm_info(get_type_name(), $sformatf("%s: Mismatch wrote %x, read %x",
        bA.id_register.name, expected_result, rdata), UVM_MEDIUM)
    else
      `uvm_info(get_type_name(), $sformatf("%s: Match wrote %x, read %x",
        bA.id_register.name, expected_result, rdata), UVM_MEDIUM)
  end
```

Figure 9 - Special Test for an ID Register

This "custom test" of this Quirky register is just 4 lines of code and a compare. Pretty easy and straightforward. Testing "normal" registers is the same, but they are easier to test "generically". Just write all the bits, and reads, etc.

VII. Generation

Address Maps

Address maps are simple lookups. Given a name, find the corresponding addresses. A register is entered by name, along with an address. A register could have more than one address in this map. For example, REGA could be at location 100 and 1100.

A register can be in multiple address maps. For example, REGA can be at location 100 and 1100 in AddressMap 1 and be at location 500 and 600 in AddressMap 2.

There is only one register in the DUT and one register model. But the register can be at many different addresses.

Registers can have different permissions in different maps. For example, on MapA the register is Read/Write, but on MapB, the register is ReadOnly.

Generating an address map is a simple translation from one format to another.

Register Model Functionality

The two kinds of model functionality (Boolean or Special) are generated in two different ways. The Boolean model can be generated, and exists on its own. The Special model can be generated in two ways. First it could simply reference a built-in special model, or it could be generated with special code.

An example of a Boolean model is the REGA and REGA_RO_B models above. (Figure 6 and Figure 8).

An example of a Special model (A Quirky Model) is CLEAR-ON-READ and ID_REGISTER above (Figure 5 and Figure 1).

Because this paper implements register models as interfaces, the opportunity to use inheritance to reuse base functionality is not available. The simplest other technique is just to have the generator put the "base functionality" directly into the generated model. This has certain limitations, but any changes to the "base functionality" can be put into the generated model by simply re-running the generator. This operation takes seconds or minutes. It can be done with each new release of base functionality. Most of the time, the base functionality will not change, since this is the core of the technology library in use.

Register Model Descriptions

In the UVM Register package the library code provides detailed implementation for the model, using code like XpredictX():

```
// XpredictX
function uvm_reg_data_t uvm_reg_field::XpredictX (uvm_reg_data_t cur_val,
                                                  uvm_reg_data_t wr_val,
                                                  uvm_reg_map    map);
   uvm_reg_data_t mask = ('b1 << m_size)-1;

   case (get_access(map))
     "RO":    return cur_val;
     "RW":    return wr_val;
     "RC":    return cur_val;
     "RS":    return cur_val;
     "WC":    return '0;
     "WS":    return mask;
     "WRC":   return wr_val;
     "WRS":   return wr_val;
     "WSRC":  return mask;
     "WCRS":  return '0;
     "W1C":   return cur_val & (~wr_val);
     "W1S":   return cur_val | wr_val;
     "W1T":   return cur_val ^ wr_val;
     "W0C":   return cur_val & wr_val;
     "W0S":   return cur_val | (~wr_val & mask);
     "W0T":   return cur_val ^ (~wr_val & mask);
     "W1SRC": return cur_val | wr_val;
     "W1CRS": return cur_val & (~wr_val);
     "W0SRC": return cur_val | (~wr_val & mask);
     "W0CRS": return cur_val & wr_val;
     "WO":    return wr_val;
     "WOC":   return '0;
     "WOS":   return mask;
     "W1":    return (m_written) ? cur_val : wr_val;
     "WO1":   return (m_written) ? cur_val : wr_val;
     default: return wr_val;
   endcase

   `uvm_fatal("RegModel", "uvm_reg_field::XpredictX(): Internal error");
   return 0;
endfunction: XpredictX
```

Figure 10 - Existing code from UVM 1.1d Register Package

There is quite a large collection of infrastructure to deal with modeling behavior in the UVM Register package. The details of this infrastructure are beyond the scope of this paper. But the real point of the new modeling for UVM Registers is to eliminate this large library of "generic implementation" code. The code path describing updating a model value is hard to follow in the UVM. In this new modeling style, the code path is easy to see directly in the model. It is "what-you-see-is-what-you-get".

The end result is that the register model generation must be able to generate a model in either the Boolean or Special style which completely describes the functionality of this register and its fields. The lightweight model will be generated with the model "intact" in the generated code, not relying on any library functionality.

VIII. COVERAGE – DO I NEED IT?

What is being covered?

For a register at the block level, what coverage is required? It is important to know that each "functional coverage goal" was met. For example, a register may have many values or modes. Functional coverage is a convenient way to know that each mode was operated.

For an address map at the block level, what coverage is required? It is important to know that each "occupied" address was exercised, along with addresses that are NOT occupied. An address map might have functional coverage for the beginning and ending addresses in the map.

Once block level simulation has completed, the functional coverage can be collected to determine if all the register modes have been exercised, and if all the address map functionality has been tested. The combination of the functional coverage saying that functionality DID get exercised and the fact that no functional failures were detected by the scoreboard gives a passing result.

When can functional coverage be stopped?

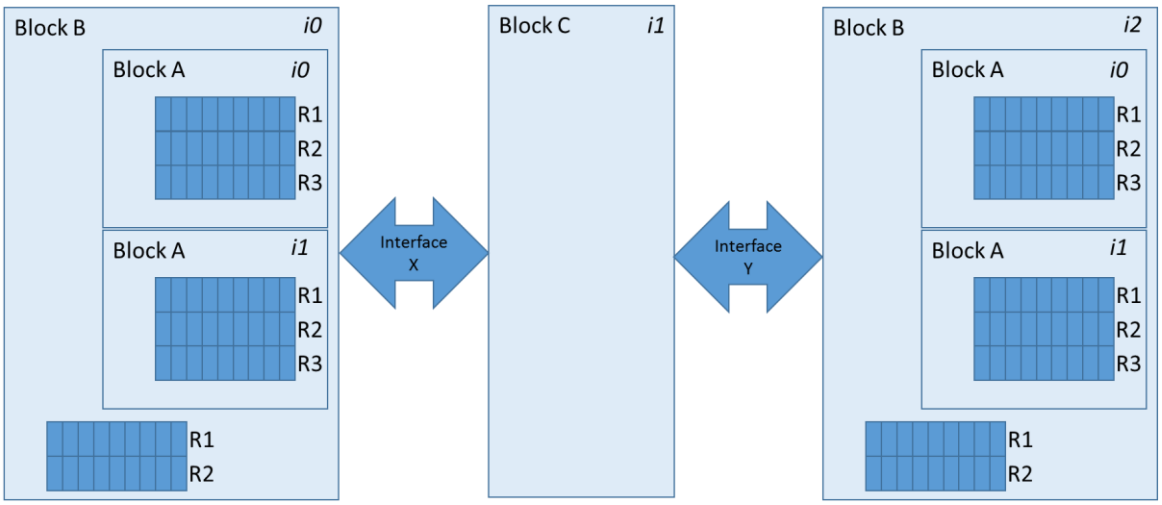
As blocks are reused inside larger blocks, functional coverage can still provide useful results, but at some point as blocks are being reused, the functional coverage results provide no extra useful information. For example, once all the register modes are covered and functional, what would make them fail as they are reused? If nothing would make them fail, then why have the overhead of functional coverage. For small blocks, this overhead is insignificant, but for larger blocks the overhead may matter. Certainly at the system level, the functional coverage of a register holds little value – unless there are conditions that could cause it to fail.

IX. EXAMPLE SYSTEM LEVEL DESIGN

Above, block verification with registers was discussed. But once a system level design is built, register verification loses some of its attraction.

In Figure 11, the lower level, pre-verified block B is instantiated twice, and connected to another block named block C to build a system level design. Block C will operate as a master, reading from one block B and writing to the other block B. These blocks are all RTL, and part of the total system.

At this level, there are no register tests to run, and likely no functional coverage to collect. At this point, the UVM register verification methodology can still be used for mirroring the DUT RTL values into a register model. That model can be printed or otherwise used in debug and analysis.



X. Extensions for "Lists Of Registers" Functionality

For certain functions, like calling reset() on all registers, it is convenient to have a "list of registers". In SystemVerilog, creating a generic list of interfaces isn't permitted. Instead, we can use a class proxy inside each interface, and register that proxy class in a list [4]. Now list processing simply iterates the list of class proxy handles.

The previously discussed ID_REGISTER gets one line added. The `REGISTER_PROXY() line.

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

Figure 12 - Interface with PROXY registration

REGISTER_PROXY builds the proxy class "inside" the interface. It provides a handle to itself using "interface::self()", and it implements the proxy calls to the interface functions. For example backdoor_load() in the proxy calls me.backdoor_load(), which is the interface function.

```
`define REGISTER_PROXY(X)                          \
  virtual X me = interface::self();                \
  class proxy extends registerp_t;                 \
    function new(string name);                     \
      super.new(name);                             \
    endfunction                                    \
                                                   \
    virtual function string convert2string();      \
      return me.convert2string();                  \
    endfunction                                    \
                                                   \
    virtual function void backdoor_load();         \
      me.backdoor_load();                          \
    endfunction                                    \
                                                   \
  endclass                                         \
  proxy p = new(name);                             \
```

Figure 13 - Proxy registration declaration

The proxy class itself is nothing more than a registration by name of itself, and a collection of pure virtual tasks or functions. There are also package scope functions which can iterate the list of registers. (dump() and do_backdoor_load()). Many other functions can be added as needed.

```
package proxy_pkg;
  typedef class registerp_t;
  registerp_t registerp[string];

  virtual class registerp_t;
    function new(string name = "registerp_t");
      registerp[name] = this;
    endfunction

    pure virtual function string convert2string();
    pure virtual function void   backdoor_load();
  endclass

  function void dump();
    foreach (registerp[name])
      $display("Name '%s' registered as %s", name, registerp[name].convert2string());
  endfunction

  function void do_backdoor_load();
    foreach (registerp[name])
      registerp[name].backdoor_load();
  endfunction
endpackage
```

Figure 14 - Proxy class with helper functions

This proxy package and class begin to look like library code, and support code. A small library may be necessary in order to provide additional functionality (iterating a list of registers apply some common function to each – like reset() or backdoor_load()).

XI. CONCLUSION

This paper continues the previous work with further discussion about requirements and missing functionality. Through this discussion it seems possible to achieve a complete working lightweight UVM register package with no requirement for a SystemVerilog library of code, nor a requirement for a VPI/PLI implementation for backdoor access. Additionally, experiments have shown that the model size will be reduced from a class based model which models fields and registers with classes. For a large system level verification environment the smaller footprint will be appreciated.

XII. REFERENCE

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

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

[3] Beyond UVM Registers – Better, Faster, Smarter, DVCON India 2015, Rich Edelman and Bhushan Safi, https://dvcon-india.org/sites/dvcon-india.org/files/archive/2015/proceedings/149_Beyond_UVM_Registers.pdf

[4] Two Kingdoms – "Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches", Dave Rich and Jonathan Bromley, DVCON 2008.

This paper was originally presented at DVCon 2016, San Jose, CA.

Corporate Headquarters
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070-7777
Phone: 503.685.7000
Fax: 503.685.1204

Sales and Product Information
Phone: 800.547.3000
sales_info@mentor.com

Silicon Valley
Mentor Graphics Corporation
46871 Bayside Parkway
Fremont, CA 94538 USA
Phone: 510.354.7400
Fax: 510.354.7467

North American Support Center
Phone: 800.547.4303

Europe
Mentor Graphics
Deutschland GmbH
Arnulfstrasse 201
80634 Munich
Germany
Phone: +49.89.57096.0
Fax: +49.89.57096.400

Pacific Rim
Mentor Graphics (Taiwan)
11F, No. 120, Section 2,
Gongdao 5th Road
HsinChu City 300,
Taiwan, ROC
Phone: 886.3.513.1000
Fax: 886.3.573.4734

Japan
Mentor Graphics Japan Co., Ltd.
Gotenyama Trust Tower
7-35, Kita-Shinagawa 4-chome
Shinagawa-Ku, Tokyo 140-0001
Japan
Phone: +81.3.5488.3033
Fax: +81.3.5488.3004

MSB 09-17 TECH16120-w