Featuring:

A Reusable SystemVerilog Testbench...

in Only 300 Lines of Code!

...see page 5

"Each issue will include several in-depth technical articles about the latest advances in verification technology and methodology, as well as some timely tips for making better use of the tools you may currently have."
—Tom Fitzpatrick

verification HORIZONS

A QUARTERLY PUBLICATION OF MENTOR GRAPHICS Q4 '05—VOL. 1, ISSUE 1

Hello and Welcome to the Verification Horizons Newsletter!

By Tom Fitzpatrick, Editor and Verification Technologist

Hello and welcome to the Verification Horizons Newsletter!

We at Mentor are pleased to share some exciting information on the frontiers of functional verification. We hope you will find this quarterly newsletter to be an important source of information as we continue to explore and present solutions in this area. Many of you have heard about the release of our Questa™ Verification Platform back in May. Questa is an evolutionary leap in the life of our successful ModelSim HDL simulator. In keeping with Questa's broader, more ambitious capabilities, we've created this new newsletter to provide concepts, values, methodologies and examples to assist with the understanding of what these advanced functional verification technologies can do and how to apply them most effectively.

We have big plans for this newsletter. Each issue will include several in-depth technical articles about the latest advances in verification technology and methodology, as well as some timely tips for making better use of the tools you may currently have.

In this first issue, the feature article presents a hands-on view of how to build a reusable SystemVerilog testbench. There has been much "buzz" recently about the need to apply

cont. page 2

"Welcome", continued from Page 1

assertions, constrained randomization and functional coverage to the functional verification problem, a need which Questa fills nicely from the tool perspective. In this article, we introduce some techniques for architecting your testbench and show how to apply these capabilities from a language and coding perspective, by focusing on modularity and transaction-level communication interfaces between verification components.

The next article, "Improving the Efficiency of Your ABV Methodology," discusses how to achieve an efficient ABV methodology through assertion automation and the targeted use of assertions. It also describes how CheckerWare assertion macros support automation, maintenance, optimization, and sufficient functional coverage.

The following article provides an overview of constrained random stimulus generation in Questa using SystemVerilog. It covers the topics of random stability for repeatability of results — an important factor in being able to debug designs — and discusses the Questa constraint solver in some detail. It provides a basic understanding of the functionality and operation of the solver and shows how to write constraints that can be solved efficiently by the solver.

In our Standards & Partners section, we will include a discussion of the new SystemVerilog-based Open Verification Library, which Mentor donated to Accellera. In addition to the standardization discussion, we will also show how to use the OVL to get started with Assertion-Based Verification.

In our final article, "Bridging an Untimed High-Level Verification Language with Timed HDL Modeling Environments — The Advantages of Transaction-Based Verification," we expand on the theme of transaction-level communication in the testbench. This article shows how architecting your testbench to use transaction-level communication provides the flexibility to connect components of different abstraction levels, including a transaction-level testbench to an RTL design. We discuss the use of the SystemVerilog Direct Programming Interface (DPI) as a useful inter-language communication mechanism. For example, the SystemVerilog DPI allows a testbench to be written in SystemC and efficiently coupled to a design in HDL, whether the design is executed in a simulator or an emulation session.

We hope you enjoy this newsletter, both in its inaugural and future issues. We will endeavor to use this forum to keep you up to date with all the latest additions to the Questa product family, as well as other verification products across Mentor. Of course, tools by themselves are not enough. Because at Mentor we will continue to focus on supporting standards in all our verification tools, it will always be incumbent on us to provide value to you, our customers, above and beyond the tools themselves. Rather than locking you into a proprietary solution, we will strive to provide this additional value in terms of training, consulting, documentation, and other means. The Verification Horizons Newsletter will be one significant part of this overall effort.

Respectfully submitted,
Tom Fitzpatrick
Verification Technologist
Design Verification and Test

Table of Contents

Verification Horizons is a publication of Mentor Graphics Corporation, all rights reserved.

Editor: Tom Fitzpatrick

Program Manager: Rebecca Granquist

Wilsonville Worldwide Headquarters

8005 SW Boeckman Rd.

Wilsonville, OR 97070-7777

Phone: 503-685-7000

To subscribe visit:
http://www.mentor.com/products/fv/verification_news.cfm

"...it will always be incumbent on us to provide value to you, our customers, above and beyond the tools themselves."

Tom Fitzpatrick
Verification Technologist
Design Verification and Test
and Verification Horizons Editor

A Reusable SystemVerilog Testbench in Only 300 Lines of Code

by David Jones, Xtreme EDA

INTRODUCTION

SystemVerilog offers an exciting new environment in which to construct testbenches. Language features support constrained random generation, object-oriented programming, assertions, coverage, and more. Verification engineers new to this environment may not know where to start or how to use these features. This paper presents a complete testbench for verifying a rock, scissors, paper arbitration module, based on a methodology developed at Mentor Graphics and XtremeEDA, aimed at building effective verification environments with minimal complexity. Due to the simplicity of the device under test (DUT) we can present the complete testbench here, as it totals fewer than 300 lines of code.

The first section of this paper briefly describes the motivations behind the major verification features of SystemVerilog. The paper then proceeds to present a high-level verification environment and describe the components that comprise it.

MODERN VERIFICATION: CONSTRAINED RANDOM COVERAGE-DRIVEN VERIFICATION

A typical test environment developed in Verilog enables directed test cases: each test case sets up a scenario in the DUT, runs a (hopefully) problematic input and verifies that the DUT responds correctly. Directed test cases are good at finding expected bugs, but will not find more complex bugs resulting from the interaction of different features. As devices get more complex, finding these bugs becomes critical to success. SystemVerilog has four main verification features to support advanced randomized testbenches:

- Object-oriented programming features allow the user to represent complex data types and to abstract away low level operations on these types. Typically, SystemVerilog verification proceeds at the transaction level, where the fundamental objects are entire transactions (bus cycles, packets, etc.) rather than signal transitions.

- Constrained randomization selects random values for data transactions. Constraints are used to ensure that the selection of values is both relevant (e.g. Ethernet payload size between 46 and 1500 bytes) and useful.

- Functional coverage tabulates events occurring within the DUT and testbench to allow the verification engineer to determine if important functions inside the DUT have been exercised under all relevant scenarios. For example, a scheduler block may take an exceptional action if all output FIFOs are full. Functional coverage can be used to verify that all output FIFOs are indeed full at some point in the simulation, preparatory to verifying that the scheduler performs correctly in this case.

- Temporal assertions verify low-level aspects of communication protocols. Communication between functional blocks in a design typically must follow a well-defined protocol over time. Such protocols include not only standards such as PCI and Ethernet, but also the timing details of any proprietary internal interfaces.

TYPES OF REUSE

Before examining a methodology for creating reusable testbenches, we should first look at what we are trying to achieve with "reuse". We can identify at least three types of reuse: reuse of verification components, reuse of test cases, and reuse of testbenches. Each type of reuse places conditions on the resulting code.

When one thinks of a reusable testbench, the first thing that comes to mind is reusable verification components, typically associated with a signaling protocol such as PCI or SPI4ϕ2. Indeed, reusable verification components have spawned a whole industry of verification intellectual property (VIP). Verification component reuse requires that the domain of possible transactions (read, write, etc.) is well defined.

In addition to verification components, individual test cases can be reused. For example, a PCI core is verified using an environment consisting of PCI drivers and monitors, as well as a suite of test cases. Although it is common to package the drivers and monitors for reuse, one can also package the test cases to create a complete test suite. Doing this effectively requires that the PCI test engineer define and abstract the protocol operations on the application side of the PCI bus (e.g. a PCI-to-Wishbone bridge would reflect transactions on a Wishbone bus). To use the test suite, customers must implement the application-side protocol in a manner specific to their designs.

Finally, proper architecture of testbenches can enhance reuse possibilities, either related to a single design, or across multiple designs. For a single design, proper architecture at the block level allows the testbench to be used as-is to cover the block within the full-chip testbench. Across multiple designs, proper architecture minimizes re-work to test successive devices.

continued next page

EXAMPLE DUT

This paper will provide the complete testbench for a rock-paper-scissors (RPS) arbiter. The rules for RPS itself are described at http://www.worldrps.com. The example DUT referees a version between two digital logic players. The pin interface for each player is shown below.

```
interface player_if(input clk);
reg          r, s, p;      // Inputs
reg          go;
reg [4:0]    score;  // Range 0-10

   modport player(
      output r, s, p,
      input go, score, clk
   );
   modport arbiter(
      input r, s, p,
      output go, score,
      input clk
   );
   modport monitor(
      input r, s, p,
      input go, score,
      input clk
   );
endinterface
```

There are three inputs that normally must be low. Upon receipt of the go signal (active high for one clock), the player must assert one of the three lines for one clock. The DUT will then determine which wins, and increment the appropriate score. Play proceeds until one score meets a limit, which is configured through a configuration interface:

```
interface limit_if(input clk);
reg          load;
reg [4:0]    limit;

   modport configure(
      input clk,
      output load, limit
   );
   modport dut(
      input clk, load, limit
   );
endinterface
```

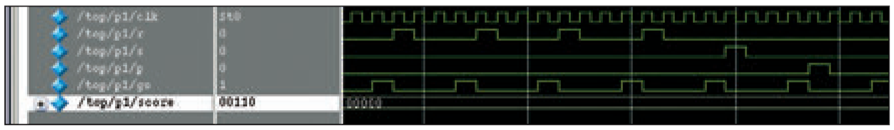


Figure 1: Player protocol

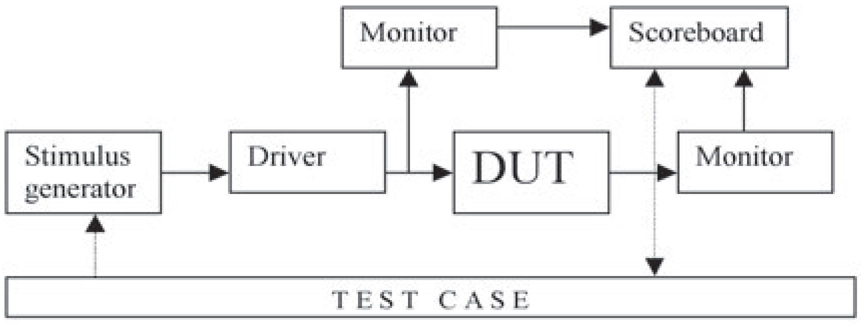


Figure 2: Typical verification environment

The timing of the protocol between a player and an arbiter is shown in Figure 1.

ELEMENTS OF A REUSABLE TEST ENVIRONMENT

The elements of a typical reusable test environment are shown in Figure 2. Drivers, monitors, test cases and scoreboards will be familiar to seasoned Verilog testbench designers. SystemVerilog allows the verification engineer to better model transactions and define the lines of communication between the components. To this end, the elements of a reusable SystemVerilog testbench also include the dynamic data objects, as well as standardized communication channels.

DYNAMIC DATA MODEL

Modern verification is done as much as possible at the transaction level. A transaction is a logical unit of work, such as a burst cycle on a bus, or a packet sent over an interface. Transaction-level modeling concentrates on the interactions of transactions upon the DUT without worrying about the pin-level representation of the transactions. Verilog testbenches cannot model data transactions very well as Verilog's only real data type is the vector of bits. In contrast, SystemVerilog classes can represent complex transactions in an organized manner. Transaction objects are passed among testbench components by reference, improving performance.

Transactions are best modeled using SystemVerilog classes. A transaction object must contain all information required: operation type, address, data, etc. Depending on the transaction it may also include the time at which the transaction was issued.

In addition to defining the data, SystemVerilog's object-oriented features allow the designer to define a functional interface to the objects through a set of public methods (tasks and functions). Manipulation of class instances through the methods is preferred over direct access to the data items. The methods allow decoupling of the data representation and the behavior of the object, so that new fields can be added, or the implementation of certain behaviors modified without impacting the verification components that work with the objects. The following operations are representative of what can be done with methods:

- Make a copy of an object.
- Compare an object with another.
- Create a string representation of the object for use in a text messaging system.
- Pack or unpack the fields of an object into a stream of bytes.

Finally, class definitions allow the specification of random fields and randomization constraints. Constraints permit automatic generation of legal transactions where the specific components of the transaction are randomly generated. Care must be taken when applying constraints. Although some constraints (such as Ethernet frame size) are desired to conform to a protocol specification, it is useful to disable these constraints to test the DUT behavior outside of the specification. Besides constraints applied for correctness, one may also apply constraints to bias stimulus generation towards interesting cases, e.g. generating high-speed streams of small packets to stress a DUT having a minimum per-packet overhead. Often these constraints conflict with one another. To deal with this, either group the constraints into separate constraint blocks and use constraint_mode() to selectively disable them, or create subclasses of a base class for each desired group of constraints. Both approaches are useful in practice.

As an example, here is the definition of the transactions for our RPS example. The definitions are placed in a package (rps_pkg) so that they may be used anywhere in the testbench.

```
package rps_pkg;
  typedef enum { ROCK, SCISSORS, PAPER }   rps_t;

  class rps_c;
    rand rps_t     rps;
    function string toString();
      return rps.name;
    endfunction
  endclass

  class rps_mon_c;
    rps_t       rps;
    bit         ok;
    int         score;

    function string toString();
      reg [63:0]  work;
      string      result;

      $sformat(work, «%0d», score);
      if (ok)
        result = rps.name;
      else
        result = «Invalid»;
      return {«score=»,work,» play=»,result};
    endfunction
  endclass
endpackage
```

Class rps_c models a single move by one player. Its only component is the choice of play (rock, paper or scissors) which we represent by an enumerated type. The rps field is random, enabling generation of random plays.

Class rps_mon_c models a transaction recovered by a monitor. In addition to the play, it conveys an “ok” status (a player may decline to make a move when required) as well as the score.

Both of these classes have a toString() method that returns a string representation of the data values.

STANDARDIZED INTERCONNECTS

Before discussing the static elements of the testbench, it is useful to discuss the techniques used to connect them together. Our methodology uses SystemVerilog interfaces for both pin-level and transaction-level interconnect.

SystemVerilog interfaces encapsulate both signal definitions and task/function definitions inside a construct that can be instantiated much like a module. We use modports to document the various functional aspects inside an interface. Pin-level (physical) interfaces are defined through the signals contained within interfaces, and transaction-level interfaces are defined using tasks and functions. The pin-level interfaces to our DUT have already been described.

For a verification component to be reusable, the functional interfaces through which it generates or accepts transactions must be well defined and standardized. Mentor Graphics has developed SystemVerilog standardized interconnects based on the OSCI SystemC TLM standard transports. Each transport is type-parameterized for the transaction type and optionally the response type.

- The TLM FIFO interface supports unidirectional blocking and non-blocking data transfers. This transport is used where the source does not care about completions (e.g. transmitting ATM cells).
- The TLM request/response channel supports two independent FIFO interfaces, one for requests, and one for responses. This transport is used where a response is required to a request, and requests and responses may overlap in time (e.g. PCI Express.) Each FIFO may block independently.
- The TLM transport channel supports a serialized request-response mechanism. A semaphore is used to ensure that only one request may be outstanding in the channel at any given time. The transmitting component blocks until a response is received.

In addition to the above channels inspired by SystemC, we have developed an analysis port interface. An analysis port is a non-blocking communication channel that can be connected to more than one sink. Each sink component is presented with a transaction using a non-blocking void function call. In contrast to the other transports, the analysis port functions correctly with zero, one, or more than one sink connected.

The above interconnect schemes handle transactional communication. Since transactions can be common to multiple devices or environments, they are good candidates for reuse. However, testbench-specific communication (e.g. between a scoreboard and a test controller) is often necessary. Ad-hoc methods, such as signals (good for boolean indications), events and hierarchical task calls can be used where required.

STATIC COMPONENTS

Referring back to Figure 2, a test environment will have the following types of devices.

Stimulus Generator

A stimulus generator creates the transactions that are sent into the DUT. A directed stimulus generator uses imperative code to create individual transactions. A better approach is to use a random stimulus generator, which randomizes the data properties of a class instance. The stimulus generator usually connects to a blocking transport, such as a TLM FIFO.

Here is the stimulus generator for our RPS example:

```
interface gen(interface.put_if sink);
import rps_pkg::*;

rps_c   item;

   always begin
       item = new;
       assert(item.randomize()) else
           $error("Can't randomize item.");
       sink.put(item);
   end
endinterface
```

The code above creates a new rps_c transaction object, randomizes it, and sends it to the transport, in this case a TLM FIFO. The FIFO must have a finite size such that this generator will eventually block. A more complex generator may support being started/stopped from the test case.

Drivers

Drivers convert transactions into lower-layer transactions or pin activity. The typical driver accepts transactions from a blocking transport such as a FIFO and either creates transactions for a lower-level protocol and passes them on to another transport, or implements the transaction as pin-level activity. Some drivers, such as bus drivers, may also need to obtain a response. These drivers will connect to the request/response or transport channels. Pin-level drivers for synchronous protocols should use non-blocking assignments to avoid race conditions.

Here is the example driver:

```
interface driver(interface.get_if source,
                 interface.player pins);
import rps_pkg::*;
rps_s         xact;

   always @(posedge pins.clk) begin
      if (pins.go && source.try_get(xact)) begin
         pins.r <= (xact.rps == ROCK);
         pins.s <= (xact.rps == SCISSORS);
         pins.p <= (xact.rps == PAPER);
      end else begin
         pins.r <= 0;
         pins.s <= 0;
         pins.p <= 0;
      end
   end
endinterface
```

The driver must conform to the device protocol. It cannot set any of the rock/paper/scissors bits until the arbiter gives us the go signal. At that point, the pins are driven based on the transaction. At all other times the player's pins must be low.

This driver uses try_get() so that it won't block. It is effectively a synchronous circuit in itself. Non-blocking assignments are used to avoid race conditions with the DUT.

Monitors

Monitors convert pin-level activity or lower-layer transactions into higher-layer transactions. In our methodology, monitors connect to analysis ports, which guarantees that the act of issuing a transaction is non-blocking, thereby avoiding the monitor missing subsequent pin-level activity. Pin-level monitors also incorporate assertions to verify the temporal properties of the protocols they are monitoring. Typically, the assertions check that the signals are well-defined (not Z or X), and that each transaction conforms to some legal cycle and follows all of the conditions imposed upon it. Basically, the assertions verify all properties of the protocol independent of the data. Monitors are located on the output path from the DUT as expected, but they are also useful on the input path, to verify that the drivers are working correctly, to provide proper transactions for the scoreboard, and to collect coverage.

Our example monitor code for one player follows:

```
interface play_mon(interface.monitor pins, input rst);
import rps_pkg::*;
rps_mon_c       item = new
rps_mon_c       item_c;
reg             last_go, last2_go;

   // Recover plays one clock after go
   always @(posedge pins.clk) begin
      last_go <= pins.go;
      last2_go <= last_go;
   end



   always @(posedge pins.clk) begin
      item.ok = 1'b1;

      if (last_go) begin
         case ({pins.r,pins.p,pins.s})
         3'b001: item.rps = SCISSORS;
         3'b010: item.rps = PAPER;
         3'b100: item.rps = ROCK;
         default: item.ok = 1'b0;
         endcase
      end

      if (last2_go) begin
         item.score = pins.score;
         item_c = new(item);
         $display("%m: %s", item_c.toString());
         ap.write(item_c);
      end
   end

   analysis_port #(rps_mon_s)  ap();

   // Asertions
   property no_meta;
   @(posedge pins.clk) disable iff (rst)
      $isunknown({pins.go,pins.r,pins.p,pins.s,pins.score})==0;
   endproperty
   assert_no_meta: assert property (no_meta);

   property valid_play;
   @(posedge pins.clk) disable iff (rst)
      pins.go |=> $countones({pins.r,pins.p,pins.s})==1;
   endproperty
   assert_valid_play: assert property (valid_play);

   property in_turn;
   @(posedge pins.clk) disable iff (rst)
      !pins.go |=> {pins.r,pins.p,pins.s}==0;
   endproperty
   assert_in_turn: assert property (in_turn);
endinterface
```

The first part of the monitor recovers transactions from the pin-level activity. We need to sample the player inputs one clock after "go" is sampled high, and we need to sample the updated scores one clock after that.

On the first clock, we decode the pin activity into one of ROCK/PAPER/SCISSORS. Only legal bit patterns are accepted; all others will result in an illegal transaction. On the second clock we sample the score and send the transaction to the analysis port.

The other important part of a monitor is the protocol assertions. We use assertions to verify that:

- There are no Z/X meta-values in the plays or score.
- That exactly one player input is high one clock after "go".
- That no player input is high at any other time.

Scoreboards

A scoreboard is a component that performs complex data checks. A typical scoreboard may include the following components:

- A database of transactions received to date. The format of this database depends on the requirements. For example, a router scoreboard may require that each port maintain an ordered queue of expected packets.
- A behavioral model of the DUT. This model must implement any required data manipulation functions of the DUT. This model is usually simpler than the DUT since it operates at the transactional level rather than the pin level, and need not be synthesizable.
- A collection of data checks. Each data check runs when the database has received sufficient data to do so. Where possible, the data checks should be written to verify the behavior of the DUT without using a behavioral model, as it is likely that a behavioral model will contain the same conceptual errors (although not necessarily implementation errors) as the DUT. For example, a Reed-Solomon encoder should be verified by attempting to decode with a behavioral decoder. If the input has not been corrupted by error injection, then the decoder should be able to confirm zero errors. However, use of checks alone is not always possible; for example, an image processing circuit is often verified against a behavioral model simply because image aesthetics are too difficult to capture in a data check set.

Our example scoreboard connects to the two monitors, one for each player. A transaction is expected at each monitor at the same time. The "database" consists of the expected scores for each player. Due to the simplicity of the DUT, the verification is performed using a behavioral model. The function wins_over() determines who wins given a pair of rps_t items obtained from two transaction objects. We collect two transactions, update the scores and compare against the scores obtained from the transactions. The scoreboard also has logic to determine when the game is over, at which point the test is done.

```
interface scoreboard(interface.ap_get_if ap1,
                     interface.ap_get_if ap2,
                     input int limit,
                     output reg pass, done);
import rps_pkg::*;
int   score1 = 0;
int   score2 = 0;

  analysis_fifo #(rps_mon_s)  fifo1(ap1);
  analysis_fifo #(rps_mon_s)  fifo2(ap2);

  initial begin
    pass = 1;
    done = 0;
  end

rps_mon_s     ev1, ev2;
rps_t         rps1, rps2;

  function bit wins_over(rps_t p1, rps_t p2);
    return (p1 == ROCK && p2 == SCISSORS) ||
      (p1 == SCISSORS && p2 == PAPER) ||
      (p1 == PAPER && p2 == ROCK);
  endfunction
  always begin
    fifo1.get(ev1);
    fifo2.get(ev2);
    rps1 = ev1.rps;
    rps2 = ev2.rps;
    score1 += wins_over(rps1, rps2);
    score2 += wins_over(rps2, rps1);
    cg.sample();

    assert(score1 == ev1.score) else begin
      $error(«Player 1 score incorrect, expected %0d got %0d»,
        score1, ev1.score);
      pass = 0;
    end

    assert(score2 == ev2.score) else begin
      $error(«Player 2 score incorrect, expected %0d got %0d»,
        score2, ev2.score);
      pass = 0;
    end

    assert(score1 <= limit && score2 <= limit) else begin
      $error(«Score is over limit.»);
      pass = 0;
    end
    if (score1 == limit || score2 == limit) done = 1;
    $display;
  end
 endinterface
```

Coverage

Coverage is the component that brings "closure" to the testbench. Coverage is required to ensure that all interesting cases have been tested. This is required in a constrained-random environment because one cannot guarantee that any given random testcase will test all interesting aspects of DUT operation. Instead, a few testcases often end up testing most of the DUT, and specially constrained testcases will be required to test the remaining corner conditions. Candidates for coverage include input and output transactions, state machines inside the DUT, corner cases for FIFOs, etc. Cross coverage (coverage of all combinations of two otherwise independent events) is useful for verifying that functional blocks operate in all modes. Any suspected problem areas within the DUT can also be covered.

The one coverage item that comes to mind in our example is all possibilities of rock/scissors/paper from both players. This is an example of a cross-coverage item since the coverage data is the Cartesian cross-product of more than one data source. We have chosen to integrate coverage into the scoreboard since the data comes from the same transactions upon which the scoreboard operates.

```
  covergroup rps_cover;
    coverpoint    rps1;
    coverpoint    rps2;
    cross         rps1, rps2;
  endgroup

rps_cover     cg = new;
```

The covergroup declaration defines what we are to cover. It is then necessary to actually instantiate the covergroup which is done immediately below. This covergroup is set up with an explicit sampling event, which is executed once the scoreboard has obtained the two transactions to cover.

Test Case

The testcase is where any test-specific configuration is performed. A testcase must configure the DUT as well as any random stimulus generators. The testcase should also manage the test termination conditions.

```
interface test case(input rst, interface.configure cfg,
                    output int limit, input pass, done);

  initial begin
    cfg.load = 0;
    limit = 20;
    @(posedge cfg.clk);
    while (rst) @(posedge cfg.clk);
    @(posedge cfg.clk);
    cfg.limit <= 20;
    cfg.load <= 1;
    @(posedge cfg.clk);
    cfg.load <= 0;
    @(posedge done);
    $display("%s", pass ? "Test PASSED" : "Test FAILED");
  end

endinterface
```

After bringing the DUT out of reset, our example testcase performs DUT configuration: it fixes the score limit at 20. It could also potentially randomize this item. The testcase then waits until the scoreboard claims the test is over, after which it displays the verdict.

There will be a different testcase file for each test scenario. It is often the job of the simulation compile/run script to select a test case to run. Alternatively, all testcases may be compiled into a single environment, such that the test to run can be selected at run time. Techniques for doing this are beyond the scope of this paper.

Top Level

The top-level file instantiates the DUT and all other components. This does not differ in construction from a typical Verilog top-level file, except for possibly the instantiation and use of interfaces. The "gen_fifo" is a standard component from our TLM library.

```
module top;
import rps_pkg::*;

   clkgen #(
      .PERIOD(10),
      .RESET_POLARITY(1'b1)
   )         clks();


wire   clk = clks.clk;
wire   rst = clks.reset;
int    limit;
bit    pass, done;

   player_if   p1(clk), p2(clk);
   limit_if    lif(clk);

   rps_dut     dut(
      .clk, .rst,
      .p1, .p2, .lif
   );
   gen_fifo    #(
      .T(rps_c),
      .BOUND(2)
   )         gen2drv1(), gen2drv2();

   gen        gen1(gen2drv1);
   gen        gen2(gen2drv2);


   driver     drv1(
      .source(gen2drv1),
      .pins(p1)
   );
   driver     drv2(
      .source(gen2drv2),
      .pins(p2)
   );
   play_mon   mon1(
      .pins(p1),
      .rst
   );
   play_mon   mon2(
      .pins(p2),
      .rst
   );
   scoreboard  score(
      .ap1(mon1.ap),
      .ap2(mon2.ap),
      .limit,
      .pass,
      .done
   );
```

```
   test case   tc(
      .rst,
      .limit,
      .cfg(lif),
      .pass,
      .done
   );
endmodule
```

CONCLUSIONS

This paper has presented a basic SystemVerilog testbench using constrained-random, coverage-driven, assertion-based techniques. We used a SystemVerilog version of the SystemC TLM library to manage the interconnect. Although this example looks complex, the test environment itself weighs in at only 300 lines of code, and illustrates the basic concepts and roles of each component. The reader can use this testbench as a template for more complex designs.

REFERENCES

[1] http://www.worldrps.com/

Improving the Efficiency of Your ABV Methodology

by Neil Hand, Mentor Graphics Design Verification and Test Division

ABSTRACT

In the quest to avoid respins and risk while striving for better verification with fewer resources, many companies are moving to assertion-based verification (ABV) methodologies. This article introduces the benefits of ABV, as well as how to achieve a highly efficient ABV methodology while understanding avoidable pitfalls. With a focus on automation, optimization, debugging, and coverage, readers will begin to understand these are essential avenues to achieving higher quality designs faster, through assertions.

INTRODUCTION

Missed market windows and respins. Responsibility for either of these situations strikes fear in even the most stoic of engineering managers. And yet, many companies are still struggling with how to avoid these situations.

A variety of issues can delay or destroy projects. But with functional verification taking more cycles and dollars, and functional bugs remaining the number one cause of respins, insufficient verification is one of the largest culprits. Furthermore, it is increasingly difficult to understand just how much verification is enough. The upshot is that many companies do not perform adequate verification on their designs.

Efforts to rectify this situation with faster simulation runtimes does not cut it. In response, EDA companies are competing to develop the most effective advanced functional verification methodologies, including constrained-random data generation, coverage-driven verification (CDV), testbench automation (TBA), formal verification, and assertion-based verification (ABV). Leading-edge companies that have employed ABV have already realized many benefits, which can be summarized as higher quality verification with fewer resources in a shorter amount of time.

However, as with any new methodology, there is a learning curve and other upfront costs. In the case of ABV, design and verification engineers must learn how to write and use assertions. Engineering managers must also understand how to leverage assertions with other advanced verification technologies and the most effective ways to adopt them. Furthermore, assertions require additional simulation cycles. This problem is exacerbated by the fact that in order for ABV to provide meaningful results, today's incredibly complex, multimillion gate designs require tens, even hundreds of thousands of assertions. Writing all of these assertions by hand takes too much time and is not an effective use of engineering resources.

Any particular ABV approach should and will be judged by how well and how quickly it delivers a highly effective design verification methodology. The standardization of design and verification languages is critical to this effort. Standards enable tool and technology interoperability and allow ABV to be adopted incrementally. However, they are not enough.

Technologies and strategies that boost ABV productivity are needed. The first of these involves the automation of the ABV methodology. This includes the complementary use of assertion libraries and assertion languages. Libraries that increase the automation of the assertions themselves, as well as ABV itself, are integral to immediate and effective adoption of assertions. The second strategy consists of empirically evolved techniques that immediately improve verification productivity and effectiveness through the incremental adoption and targeted application of ABV.

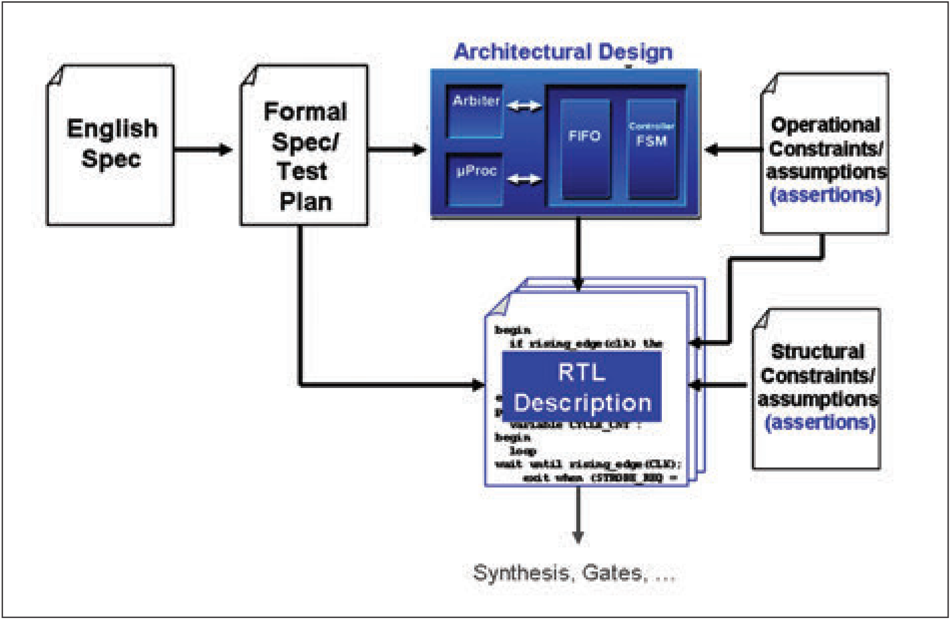


Figure 1. Assertions in an ABV flow

ASSERTIONS IN BRIEF

In order to understand assertions, it is first important to look at problems with traditional verification approaches. As designs grow in complexity, observability and controllability become a problem. Verification needs to be able to control the circuit to known values, propagate test stimulus through the design, and then observe the design's response to the stimulus. Traditional simulation approaches treat the device under test (DUT) as a black box, providing stimulus and observing results without a clear view of what is actually happening inside the design. The larger and more complex the design, the slower this process becomes as more simulation cycles are needed. Additionally, debugging functional failures becomes prohibitively difficult due to limited observability from the design.

Assertions, on the other hand, provide a much clearer view into the design. Assertions specify the intent of the design, and provide direct controllability and observability to/from the source of the problem. In the most basic sense, assertions are like actionable comments. In other words, while a designer is creating the design, he can add comments about how the design is supposed to work. Taking this one step further, by adding assertions, he can specify how the design is intended to work and actually verify its behavior during simulation.

Assertions confirm three distinct conditions. First, they can specify proper operation of the interface. In other words, assertions can verify how a block communicates with other blocks within the design. Second, assertions can verify corner-case assumptions. For example, assertions can check for unusual circumstances that may not easily be discovered during simulation. Third, assertions can be used as coverage points, providing data on how thoroughly various components have been exercised during verification.

Assertion-based verification is the use of assertions in simulation and/or formal verification. ABV compares the implementation of a design against its assertions to verify that a design functions as the designer intended.

GENERAL ABV METHODOLOGY AND THE DESIGN CYCLE

In an ABV methodology that is targeting general verification improvements, the design and verification teams must be committed to using a test plan that utilizes ABV. They understand the value of assertions and add them into the register-transfer level (RTL) code as they develop it — the earlier the better. Design intent and assumptions are captured early during the design phase. Much of this is automated. For some complex blocks, static formal verification is used to confirm the assertions before the code check-in for regression. In addition, the verification team adds protocol monitors to the standard interfaces and sets up the regression environment to run with assertions. The verification team may also add assertions to capture events that are hard to test from a pins-out perspective and that need to be tested based on the test plan. The team aggregates the statistical and coverage information to uncover holes in the regression environment. Finally, the verification team runs dynamic formal verification on multiple complex blocks at the sub-chip or cluster level.

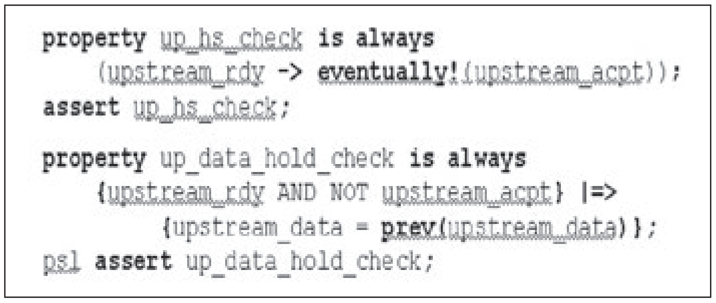
Throughout the flow, engineers should employ as much automation as possible to ensure the highest achievable verification quality with the most efficient use of resources (both man hours and simulation cycles). Additionally, standards should always be used to avoid getting locked into a single vendor flow and to ensure that assertions can be reused.

MOVING TOWARD EFFICIENT ABV

Standard Assertion Languages: A Starting Point

Assertions can be written in any hardware description language (HDL) or assertion language. The Property Specification Language (PSL) and SystemVerilog are two standard assertion languages developed and approved by Accellera. Specially constructed for writing assertions, these languages are more efficient than HDLs for this purpose. They are also standardized to support interoperability. Assertions written in SystemVerilog and PSL can be parameterized, so it is often the case that a small group of assertion experts can write a custom assertion library for use on a specific project. Assertions written in PSL or SystemVerilog can reside in the HDL code within the design or be kept as an associated file used during the verification process. Figure 1 gives an example of PSL assertions that check input handshake protocols.

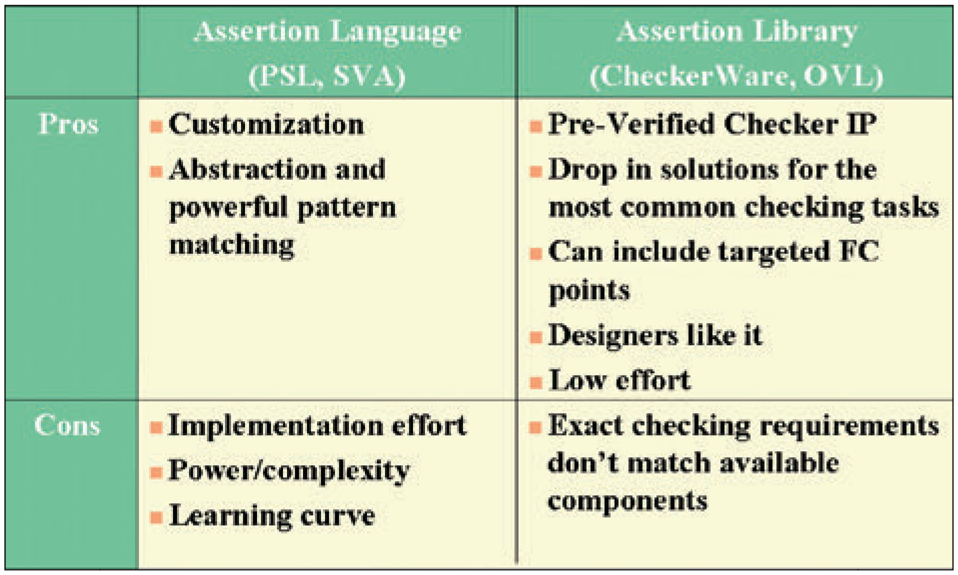
In a typical process, designers generate assertions during the design phase. Verification engineers then run assertions during verification in simulation and/or formal verification.



```
property up_hs_check is always
    (upstream_rdy -> eventually!(upstream_acpt));
assert up_hs_check;

property up_data_hold_check is always
    {upstream_rdy AND NOT upstream_acpt} |=>
        {upstream_data = prev(upstream_data)};
psl assert up_data_hold_check;
```

Figure 2. PSL assertions for verifying input handshake protocols



	Assertion Language (PSL, SVA)	Assertion Library (CheckerWare, OVL)
Pros	▪ Customization ▪ Abstraction and powerful pattern matching	▪ Pre-Verified Checker IP ▪ Drop in solutions for the most common checking tasks ▪ Can include targeted FC points ▪ Designers like it ▪ Low effort
Cons	▪ Implementation effort ▪ Power/complexity ▪ Learning curve	▪ Exact checking requirements don't match available components

Figure 3. Comparison of assertion languages to assertion libraries

Assertion Libraries:
A Step Toward Assertion Automation

While there is a need to generate some assertions manually for custom circuitry and the like, most assertions can actually be generated from assertion libraries. Libraries have certain advantages, particularly for new users, in that they are easy to instantiate in the design, requiring little effort on the designer's part to specify the desired behaviors. However, those behaviors must be contained in the library. Libraries are also high quality, since they are created by experts and have been proven over time by various teams on many projects.

Assertion libraries, such as Accellera's standard assertion library, Open Verification Library (OVL), contain common assertions in a reusable format. Many of the OVL components were donated by Mentor Graphics®. These libraries provide pre-written, pre-verified assertions for common components, specified in either PSL or SVA languages.

To get started using standard OVL libraries, visit the Accellera website and download the free source code here: http://www.accellera.org/activities/OVL_VSVA/OVL_VSVA.

Assertion Macros and Protocol Monitors:
Advanced Automation

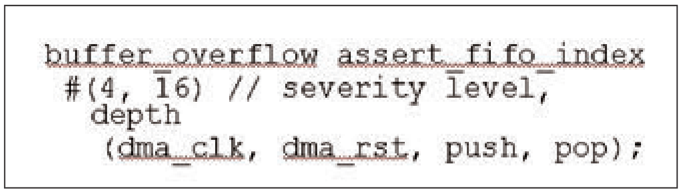
The open, standards-based library from Accellera is a good way to start automating ABV. However, OVL currently offers only 32 components, more is needed to significantly reduce the number of assertions engineers will have to write. Experience has shown that a 10 million-gate design requires over 100,000 assertions to achieve a level of assertion density to fully verify the design.

Fortunately, through a proprietary technology known as Assertion Synthesis, CheckerWare® assertion macros from 0-In provide a much higher degree of automated assertion specification and maintenance. The Assertion Synthesis solution greatly simplifies the specification of assertions by automatically extracting design data, such as clocks, resets, and variable names, from RTL code.

Because they are based on the standard verification languages, CheckerWare assertion macros satisfy the need for standardization. They complement and are totally interoperable with all standard assertion languages, including PSL, SystemVerilog, and OVL. CheckerWare assertion macros automatically generate from 2 to 4000 assertions and coverage points per macro. This means that they cover about 90 percent of the logic you need to verify in your design, vastly reducing the amount of time and effort required to implement ABV.

Assertion macros take the idea of a library one step further. While a library provides pre-defined assertions for common components, assertion macros are high-level compiler directives for the automatic generation of assertions based on supplied arguments and information automatically extracted from the design. In other words, assertion macros automatically generate assertions for common components and hook them up in the design.

Another productivity issue is maintaining assertions. Assertions can suffer from something referred to as "assertion rot." For example, if assertions are written early in the design phase, and the design goes through numerous iterations, many of the assertions may no longer apply. In these cases, assertions basically become dead code that clutter the design and waste valuable simulation cycles. Assertion Synthesis includes a unique design-inferencing capability that allows assertions to automatically adapt to design changes, significantly reducing assertion maintenance as the design evolves.



```
buffer_overflow assert_fifo_index
 #(4, 16) // severity level,
   depth
     (dma_clk, dma_rst, push, pop);
```

Figure 4. OVL assertion for buffer overflow

The CheckerWare library also includes several specialized protocol monitors. A protocol monitor is a pre-packaged, pre-verified set of thorough tests for a standard interface component. During simulation or formal verification, protocol monitors identify the source of any incorrect protocol transactions.

For example, 0-In's protocol monitor for the PCI Express serial interconnect component provides a total of 4000 assertions for functional checking, coverage, and formal constraints. For designs using PCI Express components, this provides a fully automated solution that quickly validates the interpretation and implementation of the interface component within the design. In addition to extensive checking, these monitors also provide detailed coverage data through numerous protocol statistics that are tracked during verification.

Assertion Optimization: Reducing Simulation Time

Efficient ABV methodologies include the means to optimize assertions. Assertions are used in part to reduce verification effort. However, without proper care, assertions have an adverse affect on simulation time. To keep simulation runtimes down, assertions must be optimized.

CheckerWare creates assertions that are synthesizable HDL, which are optimized for performance in all simulators and for all assertion types. As a result, a typical design with 100,000 assertions adds a mere 20 percent simulation overhead, as opposed to the 5X slowdown expected from other approaches. CheckerWare further optimizes ABV by eliminating provable assertions as well as redundant assertions so that precious simulation cycles are not wasted on assertions that do not need to be verified.

Coverage Metrics: Knowing When Verification is "Done"

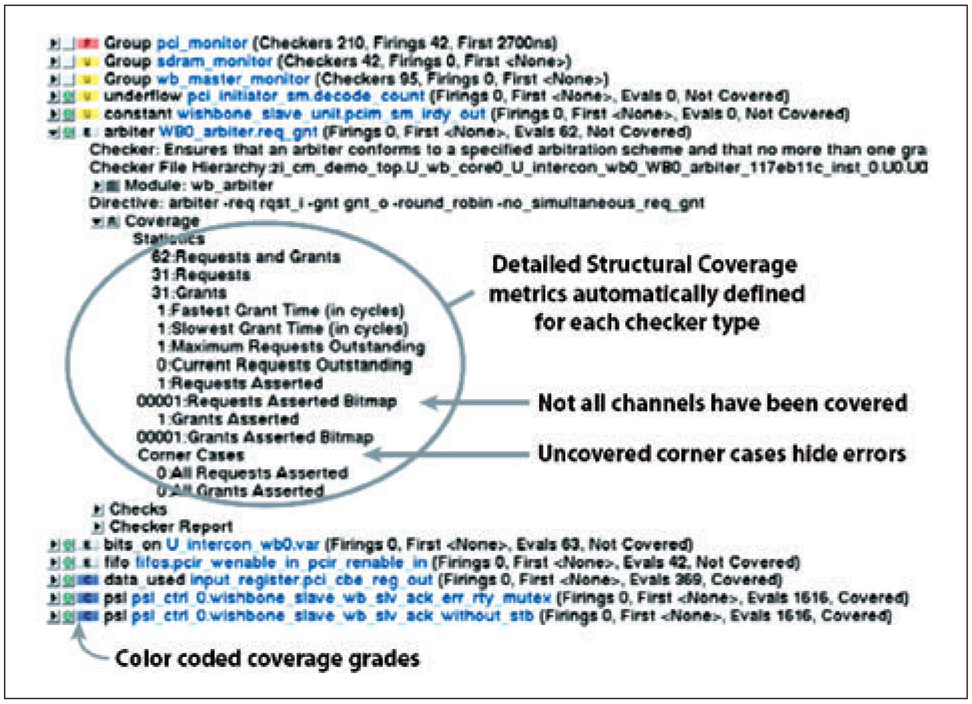
A key goal of verification is doing enough of it to ensure the design functions correctly. But knowing when is "enough" is simply guesswork unless verification coverage is tracked. Metrics and coverage are critical. The CheckerWare library includes specific functional coverage metrics related to the behaviors being checked.

Functional coverage measures verification thoroughness. In coverage-driven verification functional coverage metrics are used to automatically record and analyze information to ascertain whether (and how effectively) a particular test verified a given feature. That information is fed back into the process to target additional verification efforts more effectively. Coverage driven verification requires the specification of coverage points, or specific design behaviors, which must be exercised.

Besides checking for violations of design intent, assertions are a primary method of specifying these coverage points. While this capability is available with manual creation and standard libraries, it is more automated and more comprehensive with CheckerWare. This is partly due to the fact that CheckerWare provides extensive capabilities to track and manage verification coverage provided by assertions. The CheckerWare integrated assertion manager reports on assertion density, structural coverage for RTL components, and transaction coverage for standard interfaces.

Assertion Density: Knowing When You Have Enough

Similar to the coverage questions, the question often asked when using assertions is "how do I know I have enough assertions?" Assertion density strives to answer this question. Assertion density metrics determine the effectiveness of the ABV methodology by ensuring two things: 1) assertions exist for all critical behaviors that must be verified, and 2) all parts of the design are adequately



```
Group pci_monitor (Checkers 210, Firings 42, First 2700ns)
Group sdram_monitor (Checkers 42, Firings 0, First <None>)
Group wb_master_monitor (Checkers 95, Firings 0, First <None>)
underflow pci_initiator_sm.decode_count (Firings 0, First <None>, Evals 0, Not Covered)
constant wishbone_slave_unit.pcim_sm_irdy_out (Firings 0, First <None>, Evals 0, Not Covered)
arbiter WB0_arbiter.req_gnt (Firings 0, First <None>, Evals 62, Not Covered)
   Checker: Ensures that an arbiter conforms to a specified arbitration scheme and that no more than one gra
   Checker File Hierarchy:zi_cm_demo_top.U_wb_core0_U_intercon_wb0_WB0_arbiter_117eb11c_inst_0.U0.U0
    Module: wb_arbiter
   Directive: arbiter -req rqst_i -gnt gnt_o -round_robin -no_simultaneous_req_gnt
    Coverage
      Statistics
        62:Requests and Grants
        31:Requests
        31:Grants
         1:Fastest Grant Time (in cycles)
         1:Slowest Grant Time (in cycles)
         1:Maximum Requests Outstanding
         0:Current Requests Outstanding
         1:Requests Asserted
      00001:Requests Asserted Bitmap
         1:Grants Asserted
      00001:Grants Asserted Bitmap
      Corner Cases
         0:All Requests Asserted
         0:All Grants Asserted
    Checks
    Checker Report
bits_on U_intercon_wb0.var (Firings 0, First <None>, Evals 63, Not Covered)
fifo fifos.pcir_wenable_in_pcir_renable_in (Firings 0, First <None>, Evals 42, Not Covered)
data_used input_register.pci_cbe_reg_out (Firings 0, First <None>, Evals 369, Covered)
psl psl_ctrl_0.wishbone_slave_wb_slv_ack_err_rty_mutex (Firings 0, First <None>, Evals 1616, Covered)
psl psl_ctrl_0.wishbone_slave_wb_slv_ack_without_stb (Firings 0, First <None>, Evals 1616, Covered)
```

Detailed Structural Coverage metrics automatically defined for each checker type
Not all channels have been covered
Uncovered corner cases hide errors
Color coded coverage grades

Figure 5. Detailed coverage report

Successful Use of ABV

National Semiconductor

National is a large semiconductor firm that designed a variety of devices. One group within National designed I/O companion chips for that company's processor line. This chip contained a million gates of logic with an internal bus bridge that connected a new high-bandwidth streaming internal bus to a traditional legacy bus. The protocol between these buses was complex.

The group's challenge was to find bugs early enough in the design process to ensure no schedule impact from bug fixes. They opted to make extensive use of CheckerWare macros and protocol monitors, running them in both block-level and chip-level simulations. They also used 0-In dynamic formal verification to expand upon simulation in search of deep, corner-case bugs.

The result was National found many errors – from simple FIFO de-queuing issues to more complex memory transaction issues. This verification process yielded a chip with no problems at first silicon. The manager of the group was confident that this verification approach found bugs that had a high probability of making it into silicon.

AMD

A certain group within AMD develops complex chipsets for high-performance microprocessors. These chips contained control and data structures with many deep corner cases. The project managers were looking to improve their development process by adding assertions and formal verification. They started with just two chips. One was an I/O hub and the other was a network controller that bridged from a PCI interface to a wireless Ethernet.

With the I/O hub chip, the company targeted their efforts on CDC verification. They used the automated 0-In CDC tool and easily found problematic areas, including a case of reconvergence whereby two independently synchronized signals were combined into the same logic.

In the network controller chip, the company used 200 macros and various monitors from CheckerWare. The resulting assertions enabled engineers to increase the observability of their designs, determine structural coverage metrics, and identify bugs at the source.

The design/verification manager summed up the results nicely by saying, "We increased our tape-out confidence and helped improve our time-to-market, which translated directly to project cost savings."

Sun

Sun is a computer networking company that develops numerous ASIC and microprocessor designs. They were one of the first leading-edge companies to adopt an assertion-based verification methodology. Over the years Sun has expanded its ABV methodology to include CheckerWare macros, protocol monitors, 0-In CDC verification, and static formal verification. As a result, Sun has seen significantly faster times-to-market.

More information on these stories can be found at: http://www.mentor.com/products/fv/success/index.cfm.

covered by one or more assertions. Assertion density points out blind spots in the design due to inadequate assertion coverage, and it gives a heuristic measure of functional complexity versus assertion complexity.

Assertion density can be measured in several ways. First, it can be measured in terms of the number of active HDL constructs—the complexity of the code—relative to the number of assertions. To avoid skewing this measure, the complexity of the assertion is also a factor. For example, a redundant or provable assertion would be considered trivial. Based on the experience of many companies who have adopted ABV, a good rule of thumb is 100 lines of HDL (or 25 statements) per assertion. Another measure of assertion density is minimum sequential distance (MSD), which measures the number of clock cycles required for register values to propagate to assertions. The fewer the clock cycles, the easier it is for simulation or formal verification to evaluate the assertion. By default, CheckerWare considers unobserved logic of greater than 20 clock cycles to be uncovered.

AN EFFICIENT ABV METHODOLOGY

ABV addresses two high-level design challenges. First, as a general verification methodology, ABV adds a new level of verification thoroughness. Second, ABV can be used specifically to target functional bugs that are either missed altogether, or are very difficult to find with traditional verification techniques. The most efficient use of ABV employs both approaches, which are described in the following sections.

ABV AND VERIFICATION HOT SPOTS

Every complex design has a number of verification hot spots. Verification hot spots are those structures within or aspects of the design that are prone to problems and difficult to verify. The 80/20 rule dictates that 80 percent of verification time is spent on small portion of the design (20 percent), which is difficult to verify. The verification difficulty comes from things such as deeply buried, difficult to control logic sequential logic, interactions with state machines or other external agents, and so on. A verification hot spot cannot be completely verified by simulation alone, due to the amount and difficulty of the simulation. Assertions used with formal verification techniques are vital to effective hot spot verification.

These hard-to-verify structures, or corner cases, are called verification hot spots. Using their knowledge of the design, designers introduce assertions that capture the design intent and update their regression environment to support ABV, which specifically targets these hot spots. Designers perform simulation regressions with these assertions on a regular basis. Then, based on the structure of the design, they use static or dynamic formal verification to confirm the assertions exhaustively. Throughout this process they use as much automation as is available.

Through our experience working with customers, we've found that the best way to begin using ABV is by focusing on these verification hot spots. In this way, customers can immediately realize the benefits while adopting it in an incremental fashion by applying scarce verification resources to areas that need them most. Bus arbiters and clock-domain crossings are just two examples of hot spots that 0-In has identified.

A bus arbiter is an example of a common design structure that needs special verification focus. Arbiters allow multiple devices to share buses and are also found in a number of other complex RTL components, such as DMA controllers, schedulers, and traffic filters. CheckerWare supports the automated implementation of both static and dynamic formal verification within an ABV methodology. It includes formal constraints that make the use of formal verification both practical and effective for these hotspots. Other formal solutions require much more effort.

Clock-domain crossings (CDC) are another aspect of today's designs that are highly prone to error. Most designs have more than one clock, some have up to 10. Whenever a signal crosses between domains, special care must be taken to avoid CDC problems. Mentor's 0-In CDC verification solution can be built on top of ABV. In combination with static design checking, assertions can run in simulation and formal verification, so that the solution: 1) identifies CDC signals, 2) checks for the presence of synchronization logic, 3) verifies the correct operation of CDC protocols, and 4) measures whether all phase relationships have been verified. By encapsulating all this verification focused on one hot spot into a succinct and automated package, designers do not need to become CDC experts. All the expertise is captured in a solution driven primarily by assertions.

CONCLUSION

Companies adopt assertions to improve verification quality, detecting bugs locally and more quickly than with traditional simulation approaches. However, not all ABV strategies are equally effective. Automation plays a key role in determining the productivity and quality gains that can be expected from an ABV solution.

Key aspects of an efficient ABV methodology include: 1) using pre-defined, pre-verified assertion macros and monitors for the majority of the work, and standard languages for custom assertions when needed; 2) reducing simulation time through assertion optimization; 3) reducing design time through automated assertion maintenance; and 4) supporting coverage driven verification efforts to understand when enough verification has been done. Likewise, assertions can be used to generally improve verification thoroughness, or they can be used to specifically target verification hot spots. A combination of both is recommended.

By adopting an efficient assertion-based verification methodology — one that heavily utilizes automation — companies can quickly and easily improve their verification quality, while avoiding the addition of time, cost, and risk to the design cycle.

Constrained Random Stimulus Generation in Questa Using SystemVerilog

by Raghu Ardeishar, Mentor Graphics Design Verification and Test Division

INTRODUCTION

In the current verification environment engineers write directed tests to verify the functionality of their design. Once functionality has been verified they add more tests to the suite. Quite often this process is time consuming and many corner cases are missed.

Rather than require the verification engineer to write tests to check each feature individually, constrained-random verification (CRV) effectively allows a single test to check multiple features. With this methodology, each "test" can check many possible scenarios, and the simulator itself chooses a specific scenario for each invocation. This can be an extraordinarily powerful verification methodology, but it is one that is not supported well by either standard Verilog or VHDL.

SystemVerilog has been designed specifically to support this methodology. This article will provide some of the basics for users to start coding efficiently with SystemVerilog.

OVERVIEW OF CRV METH- ODOLOGY DIRECTED TESTING

Directed testing is a term used when the engineer creates the stimulus for every single clock cycle or transaction that might span multiple clock cycles. Each cycle or group of cycles is directed at verifying a particular feature selected by the engineer.

Because of the straightforward nature of directed tests, they are fairly easy to write. Unfortunately, by definition, they only address the explicit scenarios predicted by the verification engineer. As designs get more complex, it becomes harder to write directed tests to cover all of the possible scenarios and corner cases, both because the expected response becomes harder to predict and because the corner cases become harder to hit, if they can be predicted at all. Such tests can be improved somewhat by adding randomization, such as writing random values to a memory in addition to, or instead of, a walking-ones pattern. These tests are still inherently directed . The system functions $random or $dist_uniform in Verilog provide a simple way of filling bits with random numbers. They can also be used to randomize delays or repetition counts.

CONSTRAINED RANDOM TESTING

The idea of feeding totally random stimuli into a design seems inefficient, and it would be if there were no constraints on the random numbers the generators are allowed to produce. The idea behind CRV is that both the data and the transactions generated by the test are chosen at random from a set of valid, or constrained, possibilities.

DIRECTING TESTS FROM CONSTRAINED RANDOM

In a constrained random environment, a directed test is achieved by tightly constraining the choices so that a single scenario is exercised. Thus, a "sanity test" in this environment can be achieved by constraining the test to generate a single write to a specified address followed by a single read from the same address. Once this sanity test is validated, proving that the read/ write interface works properly, removing the constraints allows a full broad-spectrum test to occur in which all of the registers are read/ written in random order, with random data, and in all different modes. When a problem occurs, it is easy to add new constraints to the test in order to focus on the particulars that caused the problem so it can be debugged.

Directed tests are very useful in many circumstances where it may be easier to write a directed test to guarantee that the design reaches a certain state quickly, rather than rely on random behavior to achieve the desired results.

PROCESSOR GENERATED CONSTRAINED RANDOM BUS TRAFFIC

One powerful method for generating stimulus is to program the designs embedded processor to perform this task. Questa offers cycle-accurate models for most ARM embedded cores which support generation of constrained random AMBA bus cycles. The user specifies constraints for the address space, data range, bus cycle types and number of cycles. The cycle-accurate model then randomly generates AMBA bus cycles that fall within these boundaries.

Most AMBA bus slaves are so structured that the value of blasting them with random cycles is probably low, but this feature is useful for generating bus traffic from the processor while other AMBA masters arbitrate for the bus and perform their data transfers. Loading up the AMBA bus with processor driven cycles is more likely to expose any arbitration or bus bandwidth problems than having an AMBA master make transfers on a bus that's dead silent.

Users can ease into CR testing methodology incrementally using inline constraints. Consider the following example in standard verilog. Line 4 creates random values for "data" but there is not much control over what you get. In Verilog 2001 $random provides a mechanism for generating random numbers. The function returns a new 32-bit random number each time it is called.

```
1. initial begin
2. for(i=0;i<32;i++) begin
3.     addr = i;
4.     data = $random();
5.     if(addr < 16)
6.     data[7] = 1'b1;
7.     do_write(addr,data);
8. end
9. for(i=0;i<32;i++) begin
10.     addr = i;
11.     do_read(addr,data);
12.     assert(data == exp[i]);
13. end
14. end
```

Using randomize..with we can enhance the quality of the data we get as follows. You can randomize the "addr" and "data" with the constraint that "addr" is always less than 32 and then create a list of those addresses (line 3 and 5). In SystemVerilog the randomize function is provided which is similar to the $random seen before but with the added benefit that you can guide the random number generation using constraints.

You can then limit your randomization to only those addresses in the list previously created while doing a write (line 9 and 10) thus making sure that you do not read from an address which has not been written to.

```
1. initial begin
2. for (int i = 0; i < 32; i++) begin
3.     randomize( addr, data ) with { addr < 32;}
4.     if (addr<16) data[7] == 1'b1; });
5.     addr_list[addr] = addr;
6.     do_write(addr,data);
7. end
8. for (int i = 0; i < 32; i++) begin
9.     randomize( addr ) with { addr inside {addr_list}; };
10.    do_read(addr,data);
11.    assert(data == exp[addr]);
12. end
13. end
```

RANDOMIZATION WITH OBJECT ORIENTED PROGRAMMING

In SystemVerilog, random variables, random number generators, and constraints are integrated into the object oriented class system. Here are just a few important concepts.

OBJECT ORIENTED PROGRAMMING BASICS

In its simplest form, a class is like a structure, an encapsulation of data. In SystemVerilog, classes are dynamically created, whereas structures are created when they are declared. Please reference figure 1 below.

Classes are dynamic objects in SystemVerilog. The class objects have to constructed using the new() operator. Unlike structs classes can contain member functions and constraints. The member functions act on class data and constraints are used for randomizing the data. Struct data can be randomized too but you don not have the control because you cannot embed constraints to control randomization.

SystemVerilog uses the rand modifier to distinguish the random variables from the non-random variables. In line 2 and 3 in class TBase if the variables "a" and "b" did not have the rand keyword they would not be randomized. A constraint is added as a named list of expressions, declared using the constraint keyword.

The real power of object oriented programming is achieved through the use of inheritance. A new class may be defined as a derivative of a previously-defined base class, from which it inherits everything defined in the base class.

For example if we have a base class:

```
1. class TBase;
2.     rand logic [3:0] a;
3.     rand logic [3:0] b;
4.     constraint c1 { a < 4'b1100; }
5.     constraint c2 { b < 4'b1101; }
6. endclass
```

class Packet_c;
bit [7:0] address; // property
bit [31:0] data;
endclass : Packet_c
	typedef struct {
	    bit [7:0] address; // member
	    bit [31:0] data;
	  } Packet_s;

Packet_c P; // declares a handle to a Packet
P = new(); // Constructs an instance of
a Packet . P is a reference to a Packet
P.data = 1234; // Assign class members
	Packet_s P; //declares an instance of P
	P.data = 1234; // Assign struct members

Figure 1.

This class can be extended by the following class:

```
1. class TDerived extends TBase;
2.     constraint c3 { a > b ; }
3.     constraint c4 { a < b ; }
4.     constraint c5 {  a + b == 4'b1111; }
5. endclass
```

Now when we instantiate the derived class it has all the constraints c1 thru c5. It is true that some of them are contradictory but using another feature in SystemVerilog we can turn the constraints on and off selectively while randomizing the data.

Guideline: Using Inheritance you can add and override constraints as your verification environment grows. This allows for modular testing and reuse.

This is in contrast to adding all your constraints to your base class in which case your base class may have to be rewritten for many tests.

DYNAMICALLY MODIFYING CONSTRAINTS

As defined above if we wanted to randomize an object of type Tderived we would get an error because of conflicting constraints. We can solve the problem using constraint_mode() as follows:

```
1.   TDerived derived = new;
2.   derived.c3.constraint_mode(0);
3.   status = derived.randomize();
4.   derived.c3.constraint_mode(1);
5.   derived.c4.constraint_mode(0);
6.   status = derived.randomize();
```

In line 1 the object is instantiated. In line 2 the constraint c3 is turned off. So only c1, c2, c4 and c5 are valid when line 3 executes. Similarly line 4 and 5 turns on constraint c3 and turns off c4 before further randomizing. Thus we can explore several possibilities using a single class and constraint_mode().

We saw above that constraints can be turned off and on using constraint_mode(). The same can be done with variables using rand_mode().

CONSTRAINT SOLVING

Random numbers are generated with at least one constraint, the size (or number of bits) to be randomized. The size determines the total number of possible values that the random variable may have; i.e., the size of the solution space. A constraint is basically a Boolean expression that is required to be true for the values the solver picks. A constraint typically reduces the size of the solution space.

A constraint expression can be a mixture of random and non-random variables. Non-random variables make the constraint state-dependent, meaning that one can dynamically modify the constraints during the test, based on the values of other variables. If a random variable has no constraints, or appears in constraints with no other random variables, it is called a scalar random variable. Its solution space can be separated and solved independently of other random variables.

The most important concept in CRV is to understand how the solution space is managed based on a set of given constraints. Consider the following.

If we have we have 2 variables X and Y both 3 bits wide:

```
rand bit [2:0] X,Y;
constraint less_than { X < Y;}
```

The solutions are (28 possible):

if Y = 7	X = 6 or 5 or 4 or 3 or 2 or 1 or 0
if Y = 6	X = 5 or 4 or 3 or 2 or 1 or 0
if Y = 5	X = 4 or 3 or 2 or 1 or 0
if Y = 4	X = 3 or 2 or 1 or 0
if Y = 3	X = 2 or 1 or 0
if Y = 2	X = 1 or 0
if Y = 1	X = 0

The points to remember about the constraint {X < Y} are:

- X and Y are solved by the solver at the same time
- constraints are bi-directional i.e, X determines Y and Y determines X in a constraint operator. This is in contrast to conventional programming where X will determine Y.
- Conventional thinking implies the chance of getting Y = 7 is the same as getting Y = 1, But
- Probability of getting (X,Y) = (6,7) or (5,7) etc is the same as (0,1).

From the table above there are 28 possible solutions. Out of the 28 possible solutions there is only 1 solution in which Y is 1 hence

- Probability of Y=1 => 1/28

Similarly in the solution set there are (from the 1st line of the table) 7 possible

solutions for Y = 7 where X ranges from 6 to 0 hence:

- Probability of Y=7 => 7/28

If you wanted an even probability of X and Y without disturbing the solution space then you would add the following constraint along with the previous one:

```
constraint sol_Yb4X {solve Y before X;}
```

- Now the solutions remain the same as before i.e, 28 possible solutions.
- But you are telling the solver to first pick Y and then pick an X from the solution space which satisfies the constraint.
- That creates an even probability of Y from 1 thru 7.

There is another route you can take by randomizing X and Y separately and later constraining X < Y. But that

- Changes the solution space as X and Y are no longer related
- for example solver could pick Y = 0 Causing FAILURE
- But it is a potentially faster solution as the number of interacting variables decrease.

Let's look at a few operators in SystemVerilog which are used in writing constraints.

```
//Implication operator ->
    rand bit s;
    rand bit [2:0] d;
    constraint cons { s -> d == 0;}
```

The above constraint "cons" (line 3) reads if "s" is true then "d" is 0 as if "s" determines "d".But like the constraint we looked at before, the constraint implication operator, ->, is bi-directional. The values of "s" and "d" are determined together.

The important point to remember is that "s" does NOT determine "d". In a conventional "if" statement "if s then d" the value of s determines the value of d. But in this case s and d are chosen together and the solution picked randomly from the solution space.

The 9 possible values in the solution space are :

```
if s = 0  d = 7 or 6 or 5 or 4 or 3 or 2 or 1 or 0
if s = 1 d = 0
```

The (s,d) pairs will be (0,0), (0,1), (0,2),(0,3),(0,4),(0,5), (0,6),(0,7) and (1,0)

The probability of picking s = 1 will be 1 in 9 —> *Not what you thought*. If this were a conventional "if" statement then you would get s=0 with the same probability as s=1.

However if you wanted to keep the pick "s" true with a probability of 50% but not change the solution space then you can advise the solver by adding the following constraint to the above constraint

- constraint cons_plus {solve s before d;}

This additional constraint does NOT alter the solution space. Now the probability of picking "s" 1 is 50% and "s" 0 is 50%.

In the example below, there is an implied constraint in the enum variable on line 7 that the op must be one of READ, WRITE, or NOP.

```
1. typedef bit [7:0] addr_t;
2. typedef enum {READ,WRITE,NOP} kind;
3.
4. class Packet_c;
5. rand addr_t address;
6. rand bit [31:0] data;
7. rand kind op;
8. constraint data_range {
9.              (op == READ) -> data inside
                {[1:100]};
10.             op == WRITE) -> data inside
                {[101:255]};
11.             (op == NOP) -> data inside
                {0};
12.             }
13. endclass : Packet_c
```

If op equals READ, there are 100 possible values for data that satisfy the first implication. If op equals WRITE, there are 155 possible values for data that satisfy the second implication. However, if op equals NOP, there is only one possible value of data that satisfies the third implication. That makes a total of 256 possible solutions. Since only 1 out of 256 possible values for data would satisfy the third implication, op has only a 0.004 chance of having the value NOP.

Similar to the previous example , the randomization process involves calculating a solution space, then randomly picking a single solution. The solution is then written to the random variables as a set. Normally the solver picks each solution with a uniform chance. We can also use "solve before" like the previous example to advice the solver. When modifying the previous example to solve for op before data, READ, WRITE, and NOP will have a uniform chance of being chosen before choosing a value for data.

cont. om page 22

```
1. class Packet_c;
2. rand addr_t address;
3. rand bit [31:0] data;
4. rand kind op;
5. constraint data_range {
6.                  (op == READ) -> data inside
                     {[1:100]};
7.                  (op == WRITE) -> data inside
                     {[101:300]};
8.                  (op == NOP) -> data inside
                     {0};
9.                  }
10. constraint order {solve op before data;}
11. endclass : Packet_c
```

Instead of the solve-before constraint, a distribution constraint adds weighting factors for choosing values, in addition to advising which random variables should have values chosen first. It does this without modifying the solution space, except when a weight is zero. In the example below, the distribution constraint on line 10 defines that op has a 2 in 5 chance (40%) of choosing READ, a 40% chance of choosing WRITE, and a 1 in 5 chance (20%) of choosing NOP.

```
1. class Packet_c;
2. rand addr_t address;
3. rand bit [31:0] data;
4. rand kind op;
5. constraint data_range {
6.                  (op == READ) -> data inside
                    {[1:100]};
7..                 (op == WRITE) -> data inside
                    {[101:300]};
8..                 (op == NOP) -> data inside
                    {0};
9..                 }
10.   constraint op_dist { op dist {READ := 2,
                          WRITE := 2 NOP := };}
11.   endclass : Packet_c
```

Guideline: Use a distribution constraint on only one random variable in a set of interrelated random variables.

It is very difficult to calculate the probability of choosing a value for a random variable when there is what appear to be multiple, conflicting distribution constraints.

Distribution constraints are used after creating the solution space and are not guaranteed to be satisfied.

FUNCTIONS IN CONSTRAINTS

Constraint Expressions which are complicated can be simplified using functions in constraints. For example, if you wanted to compute the number of "ones" in a packed array of bits you could do the following:

```
1. rand bit[3:0] s;
2. rand bit [3:0] d;
3. constraint c1 { if((((s>>3)&1) +
                       ((s>>2)&1) +
                       ((s>>1)&1) +
                       ((s>>0)&1)) > 2) d == 0;}
```

Or you could write a function as shown below:

```
1. function automatic int count_ones ( bit [3:0] w );
2. for( count_ones = 0; w != 0; w = w >> 1 )
3.       count_ones += w & 1'b1;
4. endfunction

5. class foo;
6.      rand bit[3:0] s;
7.      rand bit [3:0] d;
8.      constraint c2 { if(count_ones(s) > 2) d == 0;}
9. endclass
```

A few very important concepts need to be emphasized:

Constraints c2(line8) and c1 (line3) look similar but act differently though seemingly achieving the same goal.

In the first example the constraint c1 is inlined and the random variables "s" and "d" are solved together. There is NO ordering implied.

However in the second example the function call forces a variable ordering. The argument of the function ie, "s" has to be solved before "d" and independent of it (unlike the "solve before" operator). The function call DOES disturb the solution space by dividing it.

EFFICIENCY IN WRITING CONSTRAINTS

Guideline: Keep the total number of related random variables down to the absolute minimum.

The amount of time needed to solve a constraint increases as the total number of interrelated bits of random variables that must be solved for one solution space increases. Try to break up the interdependencies of random variable constraints by randomizing in stages. For example, if you want to generate a sorted list of 100 random variables, you could create an iterative constraint along the likes of Ai< Ai+1.

```
1.  rand bit [9:0] arr[100];
2.  constraint c { foreach(arr[i]) (i < 99) ->
                         (arr[i] < arr[i+1]) ; }
```

However, this would create a single random variable of 1000 bits, most likely unsolvable in any reasonable amount of CPU time. A better solution is to generate 100 independent 10 bit random numbers, then sort them procedurally afterwards.

Guideline: Try not to use multiplication and division using random numbers in constraints.

If you were to think of the solution space, think of its construction like that of a synthesis tool. Like in a synthesis tool, multiplication and divisions using variables, which are declared as rand are very expensive.

```
rand bit [11:0] a, b;
constraint c1 {a * b < 40;}
```

Guideline: Try not to call randomize thousands of times on classes. If you have to, try replacing the class with a struct. Classes have function calls associated with them such pre-randomize, post-randomize, new() etc which makes them inherently slower than an equivalent struct.

Consider the following:

```
1. class dot;
2.     rand bit [15:0] A;
3.     rand bit [15:0] B;
4.     rand bit [15:0] C;
5.     constraint cA { C == B + A ;}
6. endclass
7.
8. class men_line;
9.      rand color colorb;
10.     rand dot a_of_dots[];
11. endclass: men_line
12.
13. class frame;
14.     rand men_line a_of_lines[];
15.     function new(int height,int width) ;
16.     a_of_lines = new[height];
17.     for(int i = 0; i < height; i++)
18.     begin
19.         a_of_lines[i] = new;
20.         a_of_lines[i].a_of_dots = new[width];
21.         for(int j = 0; j < width; j++)
22.             a_of_lines[i].a_of_dots[j] = new;
23.     end
24.   endfunction
25.
26.  endclass : frame
```

We are creating a 2-Dimensional Dynamic array which is going to be randomized. The base class is dot which has 3 elements each 16 bits wide and a constraint. That is instantiated in a class men_line as variable a_of_dots which is a dynamic array. That is further instantiated in dynamic array a_of_lines in class frame.

When we instantiate an object of class frame the constructor is passed arguments to make the 2 dynamic arrays.

16 a_of_lines = new[height]

e.g,

a_of_lines = new[3]

Creates an array a_of_lines with 3 elements but since the element is of type men_line which is a class the entry is only a class handle with a "null" value. hence the statement:

19 a_of_lines[i] = new;

for each element of the array.

Similarly we construct the array a_of_dots :

20 a_of_lines[i].a_of_dots = new[width];

Creates the entire a_of_dots array. But since each element of a_of_dots is a class of type dot the entry is a null class handle and the following extra step is needed for each element of the array.

22 a_of_lines[i].a_of_dots[j] = new;

Now the array elements are no longer null we can randomize the data using the randomize command.

```
frame   = fr;
fr        = new(3,3);
         result  = fr.randomize();
```

If the fr array is small this operation is relatively quick. However if fr is large then you might consider a much quicker approach (if you can give up the constraints in class dot) by replacing class dot by :

```
typedef struct packed {
 bit [15:0] A;
bit [15:0] B;
bit [15:0] C;
} dot;
```

Since dot is no longer a class the new() function can be simplified as:

```
function new(int height,int width) ;
a_of_lines = new[height];
for(int i = 0; i < height; i++)
begin
a_of_lines[i] = new;
a_of_lines[i].a_of_dots = new[width];
end
endfunction
```

cont. om page 24

DEBUGGING OF CONSTRAINTS

Questa offers command line switches to aid the debugging effort during randomization.

Guideline: Always check the return value from randomize.

If the constraints placed on the random variables in a class have no solution, randomize() will return zero. It is critical to check the return value so that any problems can be reported to the user. An immediate assertion can be useful for reporting these problems.

```
assert (P.randomize()) else $error("No solutions for P.randomize");
```

Simulator Command-line Switches

-solvefaildebug

When randomize fails, run vsim with the solvefaildebug switch and questa will display the minimum set of constraints that caused the randomize() call to fail. For example:

```
1. class TFoo;
2.   rand bit [5:0] a, b, c;
3.   constraint c1 { a < b; }
4.   constraint c2 { b < c; }
5.   constraint c3 { a < 23; }
6.   constraint c4 { b > 12; }
7.   constraint c5 { c == 20; }
8. endclass
9. class TBar extends TFoo;
10.   constraint c0 { a == c; }
11. endclass
12. TBar f = new;
13. int status;
14. $display("status = f.randomize();");
15. assert(f.randomize());
```

Questa outputs the following diagnostic error message:

```
# foo.sv(15): randomize() failed due to
conflicts between the following constraints:
#       foo.sv(4): ((f.a)<(f.b))
#       foo.sv(5): ((f.b)<(f.c))
#       foo.sv(12): ((f.a)==(f.c))
```

-solveverbose 2

Gives an idea of the solution space, what random variables are being solved, the size of the variables and the order of the variables.

CONCLUSION

The constrained random verification methodology offers significant advantages over the directed approach. Extra effort in building the infrastructure up front yields huge yields going forwards. This combined with an assertion based testing paradigm is one of the best ways of testing big complex systems. Questa today supports the SystemVerilog testbench features and assertion based methodology and offers a powerful single kernel verification environment.

Bridging an Untimed High-Level Verification Language with Timed HDL Modeling Environments—The Advantages of Transaction-Based Verification

By John Stickley, Mentor Emulation Division

It has become increasingly important to carry over models and testbenches from one level of abstraction to another for the design and verification of system designs. This capability enables reuse with a scalable methodology, allowing engineering teams to leverage the particular strengths of different modeling domains. For example, behavioral testbenches or system-reference models that are developed in a high-level verification language (HVL) [1,2,4] can be reused to verify the design-under-test (DUT) hardware models described in a hardware description language (HDL). But an intermediate modeling level is required to bridge these two levels of abstraction.

This article focuses on a solution that applies inter-language function calls (ILFCs) in order to couple untimed models written in an HVL with timed models written specifically in an HDL. This approach combines the testbench-modeling strengths of HVL with the DUT-modeling strengths of HDL. HVL environments, such as SystemC, are ideal for transaction-oriented manipulations among untimed, communicating threads. HDLs like Verilog and VHDL are more suited for the timed, signal-oriented manipulations that are used to model concurrent hardware at the cycle-accurate, register transfer level (RTL) of abstraction or below.

ILFCs have been standardized in SystemVerilog 3.1. They are referred to therein as the Direct Programming Interface (DPI) [3]. The SystemVerilog DPI allows the creation of two types of functions:

- Imported functions – defined in C, called from HDL
- Exported functions – defined in HDL, called from C

Imported and exported functions provide an ideal mechanism over which to implement untimed, transaction-based synchronizations and data exchanges between models in the HVL domain and transactors in the HDL domain. By carefully crafting DPI function-call interfaces between HDL and C models, function arguments can serve as untimed transactions. Such transactions flow in either the C-to-HDL or HDL-to-C directions between C models and transactors. A variation of the SystemVerilog DPI also was adapted for use with the Verilog 2001 HDL. This implementation was used to prototype the Ethernet-packet-router design described in this article.

Recently, the Transaction Level Modeling Working Group of the Open SystemC Initiative (OSCI-TLMWG) formalized a description of this mixed-abstraction model [8]. It then developed an application-programming-interface (API) bridge between the two levels of abstraction.

The TLMWG defined a three-level modeling paradigm. The top level is defined as the programmer's view (PV). It denotes an untimed level of abstraction for algorithmic, software-oriented modeling. The bottom level is defined as the cycle-callable (CC) level. It denotes the timed, RT level of abstraction and below. In between these two levels is the programmer's view + timing (PV+T) level. Models written at this level create an abstraction bridge between the PV and CC levels. After all, the PV+T models interface with CC models through timing shells and with PV models at the transaction level. Such bridging layers are often referred to as transactors.

Although the TLMWG has developed an API bridge between the two modeling domains, our focus will be on an HDL-transactor, API-less approach. Traditional API-based approaches, such as co-simulation and HVL-based transactors, have fallen short because they are cumbersome, inefficient, difficult to use, and/or don't promote reuse. Conversely, the TLMWG paradigm provides a useful framework and terminology for discussing the methodology put forward in this article.

AN HDL-BASED METHODOLOGY

HVLs offer several benefits: algorithms are easily and quickly prototyped, and architectural exploration is feasible. In addition, system-reference models, which can be used for on-the-fly comparisons to hardware models simulated at lower levels of abstraction, are relatively easy to develop. SystemC [1,6,7] is a good example of an HVL modeling environment. Although it was chosen for the examples in this article, the techniques described herein are general enough to be applied to a class of concurrent, high-performance, software-centric, non-HDL, untimed testbench-modeling environments.

All of these HVL environments provide a means of writing concurrent models, which the TLMWG refers to as communicating processes [8]. This capability allows large collections of inter-communicating models to be simulated simultaneously. SystemC has the advantage of being fundamentally C++. Besides providing concurrency, it comes with good support for a large number of resource libraries that can be beneficial to high-level testbenches. SystemC also provides easy access to system resources like networks, graphical user interfaces (GUIs), and bit-mapped displays.

Using a relatively small set of SystemC constructs, such as static and dynamic threads, inter-process communication mechanisms and directed random testing support can create powerful testbenches. These testbenches are modeled using the untimed level of abstraction. Higher complexity in testbench modeling can be confined to higher abstraction languages. In the early phases of a project, one also can model the entire DUT or parts of it at this level. The engineer thereby creates a reference model that can later be used to verify against the hardware prototype.

Gradually, the DUT can be migrated to hardware modeled in HDL at the timed level of abstraction. It can be coupled to the original testbench using a simple transaction-oriented function-call interface. That interface will create transactors that provide an abstraction bridge between the HVL and HDL models.

Using this technique is easier and more flexible than other HVL-to-HDL interfacing techniques because it is API-less. The interface is fully described in terms of simple-to-use, user-defined functions rather than difficult-to-use, signal-level APIs like PLI and VPI. By supporting this inter-language function-calling mechanism, the whole requirement for a complex, fixed-API is sidestepped. Abstract transactions that originate in the testbench become simple function-call arguments passed to and from the HDL transactor code. That code is capable of transforming them to timed RTL function protocols, which are suitable for direct interaction with the DUT.

The best way to couple abstraction levels is to force a purely transaction-oriented interface directly from the untimed HVL into the HDL domain. In other words, use HDL-based transactors which are fully written in HDL—not HVL. This key advantage satisfies both ease-of-use and reusability objectives. Transactors deal directly with signal activity, so it's more natural and intuitive for a typical HDL user to want to model such activity in an HDL. Additionally, HDL transactors easily scale to any HVL environment that supports the DPI function-call standard and can be fully reused by such an environment.

Untimed concurrent interactions can be elegantly modeled in a testbench using high-level constructs. Examples of such constructs include those offered by SystemC, such as threads, mutexes, semaphores, barriers, queues, and directed random data generation. When these models need to interface to the DUT via an abstraction bridge, they can do so by passing whole transactions to an HDL resident transactor. That transactor can then perform the necessary timed interactions with the DUT.

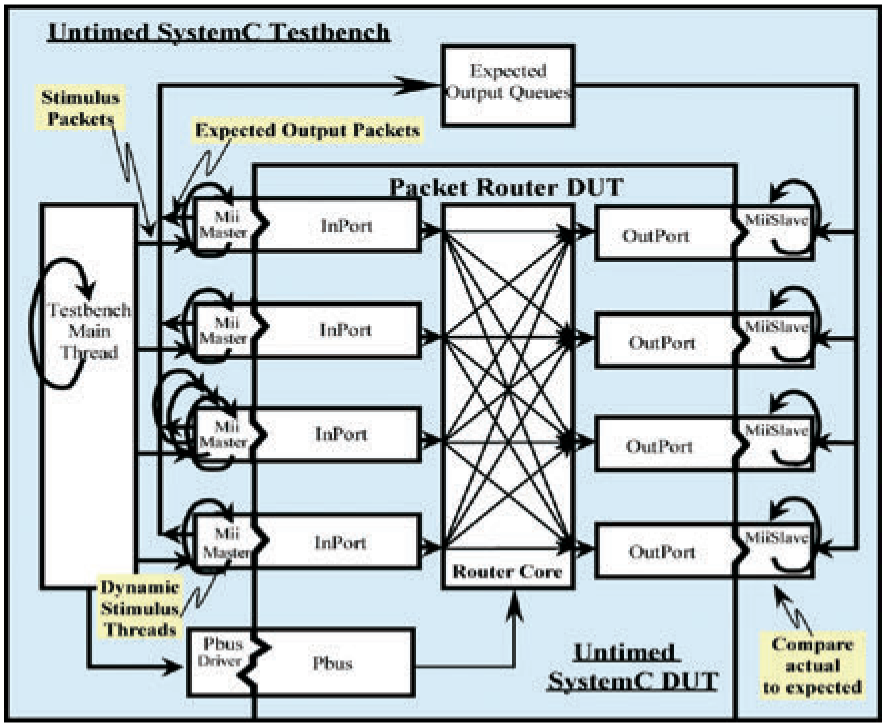


Figure 1: This diagram shows a 4-port Ethernet packet router originally modeled in untimed HVL (SystemC).

Figure 1 shows an example of a system that was initially prototyped in an HVL, such as SystemC. During the architectural-exploration phase, the design, testbench, and DUT are modeled entirely in HVL at the untimed level of abstraction. The design can be viewed as a hierarchical collection of modules (SystemC SC_MODULES, to be specific). Those modules are interconnected via abstract transaction channels (modeled using SystemC classes sc_buffer<>, sc_in<>, and sc_out<>). The transaction channels are represented as straight black arrows. Generally, each arrow depicts the flow direction of the transaction.

Contained among the modules are a number of static and dynamic threads that interact with each other. Those threads are represented in Figure 1 as C-shaped black arrows. The main driver thread first configures the router core for proper operation using a special interface, which is called the Pbus interface. The testbench

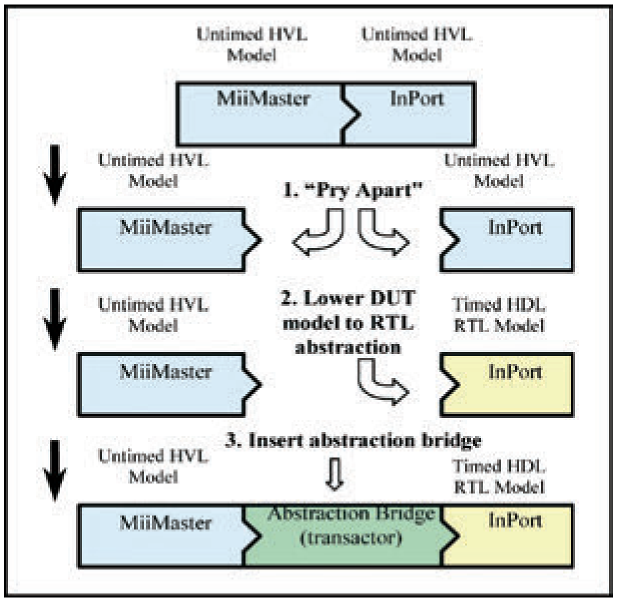


Figure 2: This process forms one approach for bridging levels of abstraction between untimed and timed modules

then enters its main loop. There, it generates a random number of packets. Each packet has a randomly selected source port, destination port, payload length, and payload content. For each packet generated, the testbench driver dynamically spawns stimulus and monitor threads on the selected input and output ports, respectively.

The stimulus threads drive the packets via the MiiMaster module into the InPort interface modules in the DUT. Generated packets also are sent to an expected output queue. Monitor threads in the MiiSlave module monitor the outgoing packets coming from the OutPort interface modules of the DUT. The threads then compare what they receive with what is expected in the output queues.

Each MiiMaster interface is treated as a shared resource. Access to this resource is arbitrated using a mutex lock that utilizes the SystemC sc_mutex class. If multiple threads are spawned that send packets on a given interface, only one can be active at a time. Pending threads will only apply their stimuli when they acquire the requested lock. In Figure 1, this stacking of pending threads is depicted with multiple, C-shaped black arrows on one of the MiiMaster ports.

The spawned threads remain pending until their packets have been successfully sent and received for comparison on the output side of the DUT. After this step, the stimulus and monitor threads die. They are replaced by other pending threads.

The testbench provides a flexible testing harness for the system, which is modeled at a high level of abstraction. All data is exchanged among modules in the form of transactions moving over data channels. Early in the architectural-exploration phase of the design cycle, the DUT also is modeled at the untimed transaction level. A number of architectural trade-offs can be quickly prototyped in this configuration.

At some point, the DUT in Figure 1 must be implemented in hardware. Often times, it will be implemented in HDL at the cycle-accurate, RTL of abstraction. When this implementation happens, the untimed testbench environment should ideally be preserved without alteration. The same testbench can then be reused to test the hardware implementation of the design.

At this point, the testbench and DUT will be of differing abstractions. An abstraction bridge is therefore required at the boundary between the DUT and the testbench. Figure 2 shows how one might go about bridging abstraction between the untimed MiiMaster stimulus module and the now-timed RTL InPort interface module.

The process of bridging abstraction can be summarized as follows:

- "Pry apart" the modules that communicate across the boundary.
- Progressively transform some or all of the DUT from untimed HVL models to timed RTL HDL cycle-callable models.
- Insert an abstraction bridge or transactor that has compatible interfaces to both the untimed testbench and the now-timed DUT.

Figure 3 (below) shows the 4-port Ethernet-packet-router system after the DUT has been

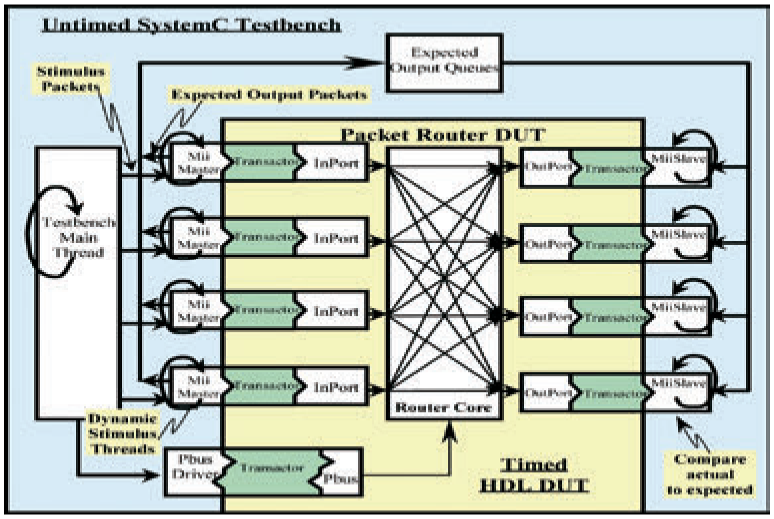


Figure 3,

migrated to hardware. The DUT, which is represented in yellow, is modeled in HDL at the timed, cycle-accurate, RT level of abstraction. The transactor modules, which are shown in green, also are modeled in HDL at the RT level of abstraction. The blue shaded area represents the original, unaltered testbench that was modeled in untimed HVL.

In effect, two disjointed hierarchies make up the simulated system. The untimed testbench remains in its own HVL hierarchy. Independently, the RTL DUT and transactors are combined in their HDL hierarchy. The two hierarchies are loosely coupled by transaction-based abstraction bridges (i.e., transactors).

STIMULUS SOURCE

Figure 5 shows a depiction of what the inside of the MiiMaster stimulus source transactor that is used in the Ethernet packet router design looks like. Using an optimized inter-language function call the SystemC model, MiiMaster makes a call to an exported task (HDL task callable from C) to send it an Ethernet frame header transaction. When the MiiMaster::SendHeader() method is called from the spawned stimulus thread, that call in turn sets the calling scope to the instance of the HDL model containing the exported task, then calls the task itself. Notice that the name of the task, SendPacketHeader() is exactly the same as the HDL name.

This implementation used the new SystemVerilog DPI standard [3] which allows calls to made between the SystemC domain and the HDL domain. The DPI specifies a fixed mapping of C data types to HDL data types of the function arguments. While the call is being made, the SystemC thread is suspended until the call is returned, at which point, execution of the thread continues.

This can happen concurrently with other threads making similar calls on other interfaces or even other instances of the same interface. In fact, in the packet router example, there can be 4 threads actually providing stimulus at any given time.

In the context of this paper it is assumed that transactors are written fully in HDL, not in HVL. This is a key advantage in satisfying both the ease-of-use and the reusability objectives. Since transactors deal directly with signal activity, it is more natural and intuitive for a typical HDL user to want to model such activity, in an HDL. Additionally, HDL transactors are fully reusable by any HVL environment that supports the DPI function call standard.

MONITOR TRANSACTORS

Monitor transactors can be implemented using imported HDL-to-C calls. Figure 6 shows an example of a simple monitor transactor. A thread in a SystemC module (SC_MODULE) called MyMonitor might be waiting for some data to come back from the HDL side via the imported HDL-to-C call, MyFunc() before continuing with its execution. This thread is denoted by the circular arrow in the diagram.

Each instance of my MyMonitor has a private semaphore data member that is implemented as a SystemC sc_event called mySem. This semaphore is used to synchronize the waiting thread with the transaction that was received in MyFunc() when called from the associated instance of the transactor on the HDL side.

To block on the semaphore, that thread calls the mySem.wait()method which is denoted by black line segment bisecting the circular arrow thread symbol. When this happens, the SystemC kernel will suspend that thread until the event occurs, which happens when MyCFunc() is finally called from the HDL transactor and posts to the semaphore.

On the HDL side, inside the transactor MyTransactor, an always block makes a call to the imported C function, MyFunc(). Inside this function on the C side, a post to a semaphore is done by calling the mySem.notify() method.

Although the MyCFunc() function is required to be a standalone C function, it can be declared as a friend of the MyTransactor module. This gives it private access data members inside SC_MODULE(MyTransactor) which include the semaphore itself. It might also include an abstract transaction data structure that can be filled out by imported C function when it is called from HDL. The transaction itself can be considered to input arguments to the function. The SystemVerilog DPI provides a mechanism to associate the call to MyCFunc with a user context pointer that, in this case, can be the SC_MODULE pointer to an instance of MyMonitor. This allows imported HDL callable C functions to be context sensitive.

Using this relatively simple technique provides a powerful mechanism for inter-language process synchronization that can be used in a variety of ways. For example, one could define a SystemC "interrupt service" module. Inside this module could be a thread that does nothing but blocks for interrupts from the HDL side. Once they occur, the HDL side can call an imported C "service routine" service the interrupt, or notify some other pending thread to do it.

SYSTEMVERILOG DPI ADAPTATION

The SystemVerilog 3.1 DPI [3] evolved from several earlier non-standard inter-language function call interfaces, notably, OpenVera's DirectC [2] and SUPERLOG CBlend [7]. The DirectC interface was donated to Accellera by Synopsys and evolved into the current SystemVerilog DPI.

The SystemVerilog DPI allows the creation of two types of functions:

- imported functions – defined in C, called from HDL
- exported functions – defined in HDL, called from C

Imported and exported functions provide an ideal mechanism over which to implemented untimed, transaction-based synchronizations and data exchange between models in the HVL domain and transactors in the HDL domain.

By carefully crafting DPI function call interfaces between HDL models and C models, a user allows function arguments to serve as untimed transactions that flow in either the C-to-HDL direction or the HDL-to-C direction, between C models and transactors.

HDL-BASED TRANSACTORS

The modeling of transactors in HDL has three advantages over co-simulation and HVL-based transactor solutions. First, it is familiar to users and therefore easy to use. Complex transactors can be written in a familiar HDL instead of C or other HVLs that are ill-suited for writing hardware-oriented models with detailed timing behavior.

Secondly, function-call interfacing to those transactors is easily scalable to multiple HVL environments. Because the SystemVerilog DPI standard is defined to be ANSI C, transaction interfaces can be created for other HVLs in addition to SystemC. Examples include Verisity's e language and OpenVera--both of which provide native support for C interfacing. Other transaction-level modeling solutions have been restricted to a single, rigid HVL environment that doesn't promote the reuse of transactors. The problem is that those transactors must be written in that specific HVL. Modeling transactors in HDL and coupling HVL to them with function calls avoids this problem.

Thirdly, transaction-accurate coupling promotes efficiency in high-performance HDL simulation platforms, such as emulators or accelerators. The coupling isn't confined by the low communication bandwidth of conventional, signal-oriented API interfaces. Whole transactions tend to occur far less frequently than the events on the signals that they trigger. These aspects dramatically improve the performance of hardware-accelerated simulations.

John Stickley is a Principal Engineer at Mentor Graphics Emulation Division. His research interests are in system-level modeling and design verification. His most recent work at IKOS Systems and Mentor Graphics has been in the area of high-performance co-modeling techniques used to bridge high-level, multi-threaded C/C++ testbenches to designs under test modeled at the RT level in HDL. He has 18 years of experience in the EDA industry and holds a BSEE degree from Cornell University.

REFERENCES:

1. SystemC - Version 2.0 User's Guide - All contributors - www.systemc.org.

2. OpenVera Language Reference Manual – Version 2.0 – Synopsys, Inc.

3. SystemVerilog 3.1 Draft 6 – Accellera's Extensions to Verilog – Accellera.

4. Usage and Concepts Guide for Specman Elite - Verisity Ltd.

5. Shotgun E - An Eight Step Approach to Experience Random Verification - Peet James, Chris Macionski.(Referenced in full online version of article.)

6. SystemC Verification Standard Specification Version 1.0b – Submission to SystemC Steering Group.

7. Functional Specification for SystemC 2.0 – Version 2.0-Q – All contributors - www.systemc.org.

8. Transaction Level Modeling: Above RTL Design and Methodology – Mark Burton, Adam Donlin.

Figure 3: This figure illustrates the same 4-port Ethernet packet router shown in Figure 1, but after the DUT has been migrated to hardware.

This submission is based on an article that originally appeared in Chip Design magazine, Aug-Sep'05, pages 21-24, and republished with permission by Chip Design magazine. www.chipdesignmag.com

Adopting Assertion-Based Verification with Accellera Standard Open Verification Library

by Kenneth Larsen, Mentor Graphics Design Verification and Test Division & Dennis Brophy, Mentor Graphics

FOREWORD

On August 3rd of 2005, Accellera unanimously approved the new OVL library as a standard, based on the work of the OVL Verilog and System Verilog (OVL-VSVA) technical committee.

In my role as chair of the Accellera OVL-VSVA committee, I had the privilege to work with industry experts from leading EDA vendors as well as end-users of OVL to get their know-how and insight on how we can enable non-expert users to take advantage of advanced verification tools, methodologies and techniques.

In this article I will explain what OVL is and how to use it through a simple example and explanation of what the various control parameters are. I will also outline a methodology on how to begin taking advantage of OVL.

WHAT IS STANDARD OVL?

At the time this article was written, the OVL library was composed of 32 assertion checkers that verify specific properties of a design, as well as capture coverage metrics. By using a single, well-defined interface, OVL provides designers, integrators, and verification engineers an open, vendor-independent interface for functional verification using static and dynamic formal verification and simulation engines. OVL also documents design intent.

OVL assertion checkers are instances of modules whose purpose in the design is to guarantee that some conditions hold true. Assertion checkers are composed of one or more properties, a message, a severity, and coverage.

For example, take a look at OVL 'assert_never'. This assertion checks that the 'test_expr' test expression does not evaluate to TRUE at each rising edge of a clock 'clk'. In cases where the test expression contains unknowns, the checker can flag this situation as well.

```
assert_never #(
 /* severity_level  */    `OVL_ERROR,
 /* property_type   */    `OVL_ASSERT,
 /* msg             */    "Register A < Register B",
 /* coverage_level  */    `OVL_COVER_ALL)
 valid_checker_inst(
  /* clock          */    clk,
 /* reset           */    reset_n,
 /* test_expr       */    regA < regB );
```

The parameters in this simple 'assert_never' example illustrate the control an OVL user has over the individual assertions. For example, it is often the case that you want to guard and verify correct input and output behavior, and by using the 'severity_level', you can report the importance of the check. You could even stop the simulation run, if desired, by setting the 'severity_level' to `OVL_FATAL.

The 'property_type' determines whether to use the assertion checker as an assert property or an assume property. Setting the 'property_type' to OVL_ASSUME tells the verification engines that the OVL checker is a constraint that should not be checked but assumed.

By default OVL checker firings are reported as a "VIOLATION", but by using the 'msg' parameter you can specify that a more meaningful message is reported.

An example of a firing is illustrated below.

```
# OVL_ERROR : ASSERT_NEVER : Register A <
Register B : : severity 1 : time 900 : DUT.valid_
checker_inst.ovl_error_t
```

The capture of cover points is built into OVL checkers. This capability is controlled by the 'coverage_level' parameter. In the 'assert_one_hot' checker, for example, one of the cover points reports if all possible combinations of one-hot values are evaluated, or as illustrated below, if the 'test_expression' actually changed. This provides good insight into how well the test environment exercises the circuit and whether any coverage holes exist.

```
# OVL_COVER_POINT : ASSERT_ONE_HOT :
test_expr_change covered : time 1300 : DUT.
check_fsm_is_onehot.ovl_cover_t
```

All OVL assertion checkers call a number of standard system tasks to check for correct usage of parameter values, error handling, reporting, and so forth. They also allow for customization to facilitate the integration of OVL into your verification environment.

WHAT IS IN STANDARD OVL?

The OVL library includes a number of assertion checker classes, as listed in the text box on the following page. The full list of checkers can be found in the side bar at the end of the article.

OVL Assertion checker Class	Behavior checked
Combinatorial assertions	with combinational logic
Single-cycle assertions	In the current cycle
2-cycle assertions	for transitions from the current cycle to the next
n-cycle assertions	for transitions over a fixed number of cycles
Event-bounded assertions	between two events

Figure 1: Assertion Checkers

GETTING STARTED WITH STANDARD OVL

The number of available checkers in OVL may be overwhelming to new users. A few guidelines will help determine where to add OVL checkers for both HDL simulation and formal verification.

• Add checkers where you make an assumption. For example, where you expect a signal will be available for at least n-clock cycles; and when if one event happens then another will happen. 'assert_always', 'assert_never', 'assert_implication' are examples of checkers that can be used here.

• Add checkers to finite state machines (FSM) to check for illegal transitions, illegal states, and that you do not stay in a state longer than expected. Checkers such as 'assert_no_transition' and 'assert_time' can be used for this. Use the coverage information obtained by the checker to ensure that all states of the FSM have been covered during simulation.

• Add checkers to check for legal ranges. For example, 'assert_range' for addresses to memory structures. Also, checking for underflow and overflow is very valuable. You can use 'assert_underflow' and 'assert_overflow' for counters that are not allowed to wrap around.

• For memory structures, FIFOs are always a good target for adding checkers, because many bugs tend to creep in to them. Another use of checkers is to ensure correct behavior and seek out bugs. 'assert_fifo-index' can be used to ensure that a FIFO pointer should never underflow or overflow

• Add checkers to interfaces to ensure the correct exchange of information. For example, if you send a request, you should receive a grant within a given number of cycles; and that the 'I am done' signal must assert within a given set of cycles after the leader signal was raised. 'assert_handshake', 'assert_unchange', 'assert_win_change' are examples of checkers that can be used.

Looking ahead, we see a movement by the advanced users of assertions and ABV towards completely "hardening" the design interface with checkers monitoring for all illegal behaviors. In this scenario, by definition, all behavior that is not caught by the checkers must be an acceptable behavior. This will provide a number of benefits, such as removing corner-case bugs, reducing "contract break" bugs, and helping verification engineers target their test efforts. If a firing does occur, it means that you have found a bug in a block that interacts with your block, that you or your peer designer have misunderstood the specification about your interface, or that you have a bug in your checker. An article on this subject will be submitted at a later date.

But as always, keep it simple, and don't repeat what you just wrote in your RTL.

USING STANDARD OVL

The OVL library has a single interface and multiple implementations. It is currently provided in Verilog-95 and SystemVerilog. Other language implementations, such as PSL and VHDL, are expected in the near term, but no matter the underlying implementation, the simple use-model will continue to be the same.

To use the OVL checkers, you have to specify which capability to enable, as well as the desired implementation language to use. These settings can often be added to a Verilog command file to simplify the compilation process. Here is a small example of one such file.

```
// ovl.f - OVL Verilog command file
+libext+.v+.vlib+.sv
// Enable OVL checkers and coverage capabilities
+define+OVL_ASSERT_ON
+define+OVL_COVER_ON
-y <Accellera_installation_dir>/std_ovl
+incdir+<Accellera_installation_dir>/<std_ovl>
```

To compile a design with ModelSim using the Verilog implementation of OVL use

```
vlog +define+OVL_VERILOG –f ovl.f <design files>
```

To compile a design with Questa using the SystemVerilog implementation of OVL use

```
vlog –sv +define+OVL_SVA –f ovl.f <design files>
```

To compile a design to be used with 0-In static formal verification, either use the same setup as for ModelSim and Questa or use the '–ovl' and '–ovl_cov' command line options. T

The library will automatically be detected and loaded.

Assertion	Description
assert_always	Ensures that the value of a specified expression is TRUE.
assert_always_on_edge	Ensures that the value of a specified expression is TRUE when a sampling event undergoes a specified transition.
assert_change	Ensures that the value of a specified expression changes within a specified number of cycles after a start event initiates checking.
assert_cycle_sequence	Ensures that if a specified necessary condition occurs, it is followed by a specified sequence of events
assert_decrement	Ensures that the value of a specified expression changes only by the specified decrement value.
assert_delta	Ensures that the value of a specified expression changes only by a value in the specified range.
assert_even_parity	Ensures that the value of a specified expression has even parity
assert_fifo_index	Ensures that a FIFO-type structure never overflows or underflows
assert_frame	Ensures that when a specified start event is TRUE, the a specified expression must not evaluate TRUE before a minimum number of clock cycles and must transition to TRUE no later than a maximum number of clock cycles
assert_handshake	Ensures that specified request and acknowledge signals follow a specified handshake protocol.
assert_implication	Ensures that a specified consequent expression is TRUE if the specified antecedent expression is TRUE.
assert_increment	Ensures that the value of a specified expression changes only by the specified increment value.
assert_never	Ensures that the value of a specified expression is not TRUE.
assert_never_unknown	Ensures that the value of a specified expression contains only 0 and 1 bits when a qualifying expression is TRUE.
assert_next	Ensures that the value of a specified expression is TRUE a specified number of cycles after a start event.
assert_no_overflow	Ensures that the value of a specified expression does not overflow.
assert_no_transition	Ensures that the value of a specified expression does not transition from a start state tot the specified next state.
assert_no_underflow	Ensures that the value of a specified expression does not underflow.
assert_odd_parity	Ensures that the value of a specified expression has odd parity.
assert_one_cold	Ensures that the value of a specified expression is one-cold (or equals an inactive state value, if specified)
assert_one_hot	Ensures that the value of a specified expression is one-hot.
assert_proposition	Ensures that the value of a specified expression is always compositionally TRUE.
assert_quiescent_state	Ensures that the value of a specified state expression equals a corresponding check value if a specified sample event has transitioned to TRUE.
assert_range	Ensures that the value of a specified expression is in a specified range.
assert_time	Ensures that the value of a specified expression remains TRUE for a specified number of cycles after a start state.
assert_transition	Ensures that the value of a specified expression transitions properly from a start state to the specified next state.
assert_transition	Ensures that the value of a specified expression transitions properly from a start state to the specified next state.
assert_unchange	Ensures that the value of a specified expression does not change for a specified number of cycles after a start event initiates checking.
assert_width	Ensures that when value of a specified expression is TRUE, it remains TRUE for a minimum number of clock cycles and transitions from TRUE no later than a maximum number of clock cycles.
assert_win_change	Ensures that the value of a specified expression changes in a specified window between a start even and an end event.
assert_win_unchange	Ensures that the value of a specified expression does not change in a specified window between a start event and an end event.
assert_window	Ensures that the value of a specified expression is TRUE in a specified window between a start event and an end event.
assert_zero_one_hot	Ensures that the value of a specified expression is zero or one-hot.

Figure 2: Assertion Checkers

NOTES

Editor: Tom Fitzpatrick

Program Manager: Rebecca Granquist

Wilsonville Worldwide Headquarters

8005 SW Boeckman Road

Wilsonville, OR 97070-7777

Phone: 503-685-7000

Subscribe: http://www.mentor.com/products/
fv/verification_news.cfm