SystemVerilog Interface Cookbook

Paul Egan
Rockwell Automation
Milwaukee, WI
pbegan@ra.rockwell.com

Kathleen Otten
Rockwell Automation
Milwaukee, WI
kkotten@ra.rockwell.com

***Abstract*— The interface is perhaps the most versatile part of the SystemVerilog language when it comes to verification. The interface is where static meets dynamic, abstract meets concrete, the rubber meets the road, the glue that holds a verification environment together…**

The interface is the main communication mechanism between the static Device Under Test (DUT) and the dynamic testbench world. Since the introduction of the SystemVerilog language in 2005, there have been several papers written on interfaces and testbench-DUT connections [3-11], but no comprehensive reference that shows the many ways to use an interface.

This paper gives an overview of where to apply the different testbench-DUT connection methods for a typical System on Chip (SOC) design.

Keywords—interface; abstract; concrete; register layer; backdoor access

I. INTRODUCTION

The most common method to connect a testbench to a DUT is the SystemVerilog virtual interface. This approach is well-defined and proven, and in many situations the best way to connect to the DUT. In large and complex SOC's containing one or more blocks of reused IP, non-standard communications protocols, and application specific IP, how does a user connect all of the legacy Verification IP (VIP) and UVM compliant VIP in a manner that allows creating a reusable UVM testbench? What if the user has a large library of VHDL Bus Functional Models (BFMs)? Do they have to rewrite all of these in SystemVerilog? What if the user has Verilog or SystemVerilog BFM's? Can these be integrated into a UVM testbench? What if the design requires code running on a processor? How does the user synchronize the testbench with the processor?

Based on our professional experience, we believe the testbench should be completely independent of the DUT, and the DUT treated as a blackbox. As such, we are adamantly opposed to the usage of SystemVerilog hierarchical references from the dynamic testbench world back to the static DUT world. The testbench should be architected such that it doesn't know or care about the DUT hierarchy. This will make the testbench more easily reused. When the testbench does require access to an instance inside the DUT, for example, backdoor register read/write, we show how to use the SystemVerilog bind construct and/or the abstract-concrete class to connect the DUT to the testbench. This keeps the "hierarchical" reference where it belongs in the static DUT world.

In all of the examples shown, the overriding theme is the test environment is architected as though all of the VIP is UVM compliant. This allows users to migrate legacy VIP to Universal Verification Components (UVCs) as time permits without having to change the test environment, sequences, and tests. The BFMs will be integrated into an environment and look just like a UVC. The abstract base class/concrete derived class connection method is included here since it looks similar to an interface, and in some cases is the best way to connect the DUT to the testbench.

Fig. 1 is a high-level block diagram of a typical SOC – processor, peripherals, and custom logic. The examples that follow refer to the UART block of our SOC.

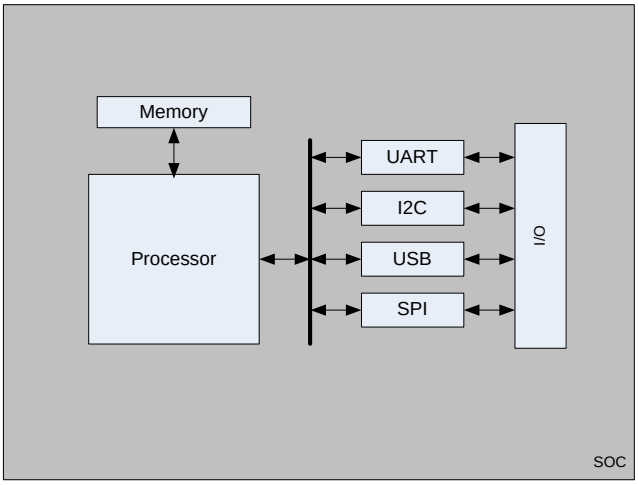

Fig. 1. Typical SOC Block Diagram

II. VHDL BUS FUNCTIONAL MODEL

Contrary to popular belief, VHDL is not "dead," nor is it the new Latin [10]. VHDL is still widely used for FPGA development. In many cases, users may have a large library of existing VHDL BFM's and a desire to migrate to a UVM test environment, but may not know how to integrate the BFM into a UVM testbench. Some of these users are under the assumption that all of their legacy VHDL models must be converted to a SystemVerilog UVC. Here, we demonstrate that conversion to a UVC is not obligatory. Unlike the Verilog BFM, for which there are multiple ways of integrating into a

UVM test environment, there is only one way to connect a VHDL BFM in a UVM testbench.

There are a few problems to consider when using a VHDL BFM, the most important being there is no Language Reference Manual (LRM) for VHDL-SystemVerilog simulation. This means each simulator vendor has its own specific rules on interoperability (restrictions on VHDL port types, generics, and data types). Next, it is not possible to call a VHDL procedure from SystemVerilog; or use a cross module reference (XMR) into a VHDL entity from SystemVerilog (note: there is also no support in the UVM base class library for register model backdoor access to VHDL since this is vendor dependent).

The interoperability and procedure calling problems can be solved by adding two layers of code to the BFM. The first layer is a VHDL wrapper that serves two purposes: to decompose ports of record type into individual signals; and to call the BFM procedures. The second layer is to connect the VHDL BFM wrapper to a SystemVerilog virtual interface. To ensure the greatest probability of interoperability success between different simulators, the ports on the VHDL BFM wrapper will use std_logic, std_logic_vector, integer, and real data types (Note: strings are typically supported as well).

In the example shown in Fig. 2, the DUT is a simple UART in a typical UVM testbench. The test environment contains our shell UART UVC agent (and possibly a scoreboard and other agents). The static testbench contains the DUT, a clock and reset generator, the wrapped BFM, and some virtual interfaces.

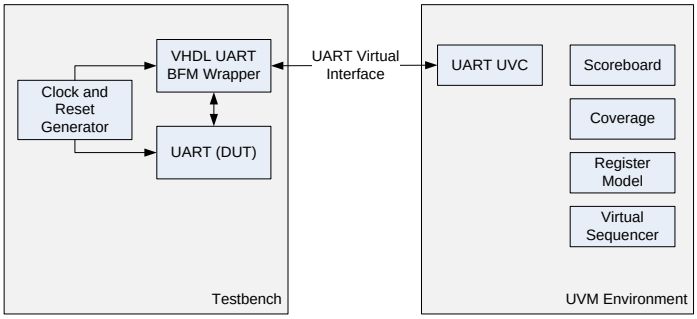

Fig. 2. Testbench and Environment with Legacy VHDL BFM

The legacy BFM shown in Fig. 3 includes a record in its port map. It pops transactions out of a queue and calls the uart_read, uart_write, and uart_reset tasks which are defined in the package shown in Fig. 4. This package also defines the record used in the port map.

```
entity uart_bfm is
  port(
    clk_in     : in   std_logic;             -- BFM clock input
    rst_in     : in   std_logic;             -- BFM reset input
    datout     : in   std_logic_vector (7 downto 0);   -- data from uart
    interrupt  : in   std_logic;             -- interrupt(1)
    sout       : in   std_logic;             -- serial output
    clk        : out  std_logic;             -- 10 mhz clock
    reg_rw     : out  reg_rw_trans_t;    -- Outputs for register R/W transactions
    rst        : out  std_logic;             -- reset(0)
    sin        : out  std_logic              -- serial input
  );
end uart_bfm ;

architecture beh of uart_bfm is begin
  main: process begin
    case trans_queue(q_index_out).opcode is
    when reset =>
      uart_reset (    trans_q => trans_queue(q_index_out),
                      rst_n   => rst);
      q_index_out := q_index_out + 1;

    when write =>
      uart_write ( trans_q      => trans_queue(q_index_out),
                   wr_data      => reg_rw.datin,
                   wr_addr      => reg_rw.addr,
                   rw_n         => reg_rw.nrw,
                   chip_select => reg_rw.cs);
      q_index_out := q_index_out + 1;

    when read =>
      uart_read  ( trans_q      => trans_queue(q_index_out),
                   rd_addr      => reg_rw.addr,
                   rd_data      => datout,
                   rw_n         => reg_rw.nrw,
                   chip_select => reg_rw.cs);
       q_index_out := q_index_out + 1;
    end case;
  end process main;
end beh;
```

Fig. 3. Legacy VHDL BFM Source Code

```
package uart_bfm_pkg is
   -- define records and enumerated types
   type reg_rw_trans_t is
      record
         addr  : std_logic_vector (2 downto 0); -- 3-bit address
         cs    : std_logic;                     -- chip select
         datin : std_logic_vector (7 downto 0); -- data to uart
         nrw   : std_logic;                     -- r(0), w(1)
      end record;
   type opcode_e is (reset, write, read, none);

   -- Declare procedures
   procedure write ( write_data  : in std_logic_vector(7 downto 0);
                     write_addr  : in std_logic_vector(2 downto 0));
   procedure read  ( read_addr   : in std_logic_vector(2 downto 0));
   procedure reset ( num_clks    : in natural );
   procedure uart_reset ( variable trans_q : inout trans_t;
                          signal   rst_n   : out   std_logic);
   procedure uart_write ( variable trans_q : inout trans_t;
                          signal   wr_data: out std_logic_vector(7 downto 0);
                          signal   wr_addr: out std_logic_vector(2 downto 0);
                          signal   rw_n    : out std_logic;
                          signal   chip_select : out std_logic);
   procedure uart_read ( variable trans_q : inout trans_t;
                         signal   rd_data : in  std_logic_vector(7 downto 0);
                         signal   rd_addr: out  std_logic_vector (2 downto 0);
                         signal   rw_n    : out  std_logic;
                         signal   chip_select : out std_logic);
end uart_bfm_pkg;

package body uart_bfm_pkg is
   procedure reset ( num_clks : in natural ) is
      variable reset_trans : trans_t :=
         ( opcode    => reset,
           address   => (others => '0'),
           write_data => (others => '0'),
           read_data  => (others => '0'),
           num_clks => num_clks);
      begin
         trans_queue(q_index_in) := reset_trans;
         q_index_in := q_index_in + 1;
      end reset;

   procedure uart_reset ( variable trans_q : inout trans_t;
                          signal   rst_n   : out   std_logic) is
   begin
      report "In uart_reset";
      rst_n <= '0' after CLK_PRD, '1' after trans_q.num_clks*CLK_PRD;
      wait for trans_q.num_clks*CLK_PRD;
   end uart_reset;

   -- Define remaining procedures
   end uart_bfm_pkg;
```

Fig. 4. Package Accompanying the Legacy VHDL BFM

The VHDL BFM wrapper shown in Fig. 5 decomposes the BFM's ports of record type into individual signals. Based upon signals driven by the virtual interface, it calls tasks which insert transactions into the queue utilized by the uart_bfm beh architecture.

```
entity uart_bfm_wrapper is
   port(
         -- Inputs for BFM
         clk_in       : std_logic;
         rst_in       : std_logic;
         -- other inputs for BFM

         -- Inputs from UVC
         write_data   : in std_logic_vector (7 downto 0);
         rw_addr      : in std_logic_vector (2 downto 0);
         r_wn         : in std_logic;
         num_clks     : natural;
         reset_start  : in std_logic;
         rw_start     : in std_logic;

         -- Outputs for DUT
         addr         : out std_logic_vector (2 downto 0);
         clk          : out std_logic;
         -- other outputs for DUT...

         -- Outputs for UVC
         read_data    : out std_logic_vector (7 downto 0);
         reset_done   : out std_logic;
         rw_done      : out std_logic
   );
end uart_bfm_wrapper;
architecture beh of uart_bfm_wrapper is
   -- Declare components and any internal signals...
   begin
      uart_bfm_inst : uart_bfm
         port map (
            -- Decompose records
            reg_rw.addr   => addr,
            reg_rw.cs     => cs_internal,
            reg_rw.datin  => datin,
            reg_rw.nrw    => nrw,
            -- Connect other BFM I/O...
      );
      main: process begin
         if (reset_start = '1') then
            reset_done <= '0';
           reset(num_clks => num_clks);
            wait until rising_edge(rst_internal);
            reset_done <= '1';
         elsif (rw_start = '1') then
            rw_done <= '0';
            if (r_wn = '1') then
               read(rw_addr);
            else
               write(write_data, rw_addr);
            end if;
            wait until rising_edge(cs_internal);
            rw_done <= '1';
         end if;
      end process main;
end beh;
```

Fig. 5. VHDL BFM Wrapper Source Code

Fig. 6 shows the SystemVerilog interface that will connect the dynamic verification environment to the VHDL BFM wrapper.

```
interface uart_uvc_if (input clk, input reset);
   // Inputs to BFM Wrapper
   logic  [7:0]   write_data;
   logic  [2:0]   rw_addr;
   logic          r_wn;
   logic  [2:0]   num_clks;
   logic          reset_start;
   logic          rw_start;

   // Outputs from BFM Wrapper
   logic  [7:0]   read_data;
   logic          reset_done;
   logic          rw_done;
endinterface: uart_uvc_if
```

Fig. 6. Second Layer: SystemVerilog Interface

The testbench in Fig. 7 instantiates the DUT, the VHDL BFM wrapper, and the SystemVerilog interface:

```
module uart_tb ();

   uart_uvc_if uart_if(.clk (clk_in), .reset (rst_in));

   uart dut(
      .addr      (addr),
      // Make remaining port connections...
   );

   uart_bfm_wrapper bfm_wrapper
   (
      .clk_in     (clk_in),
      .rst_in     (rst_in),
      // Inputs from DUT  //////////////////////
      .datout     (datout),
      .interrupt  (interrupt),
      .sout       (sout),
      // Inputs from UVC  //////////////////////
      .write_data  (uart_if.write_data),
      .rw_addr     (uart_if.rw_addr),
      .r_wn        (uart_if.r_wn),
      .num_clks    (uart_if.num_clks),
      .start_reset (uart_if.start_reset),
      .start_rw    (uart_if.start_rw),
      // Outputs for DUT  //////////////////////
      .addr  (addr),
      .clk   (clk),
      .cs    (cs),
      .datin (datin),
      .nrw   (nrw),
      .rst   (rst),
      .sin   (sin),
      // Outputs for UVC  //////////////////////
      .read_data     (uart_if.read_data),
      .reset_done    (uart_if.reset_done),
      .rw_done       (uart_if.rw_done)
   );

   // Add the virtual interface to the uvm_config_db
   initial begin
      uvm_config_db #(virtual uart_uvc_if)::set(null, "uvm_test_top", "vif_uart", uart_if);
      run_test();
   end

endmodule: uart_tb
```

Fig. 7. Testbench Source Code

The main task of the UART agent in the environment is to call procedures in the BFM. The procedures, in turn, wiggle the pins on the DUT. The start and done signals of the uart_uvc_if cause the procedures to be called. When a sequence is started on the UART agent with either a read or write sequence item, the driver sets the rw_start signal, which is connected through the wrapper layers to the BFM. This initiates a procedure call (either read or write) from the VHDL wrapper to the BFM. When the procedure completes, the wrapper sets the rw_done signal which tells the uart_driver to call item_done. See the driver source code in Fig. 8.

```
virtual task drive_non_blocking();
   forever begin
      @(negedge vif.clk)
      if (!vif.reset) begin
         seq_item_port.try_next_item(req_txn);
         if (req_txn == null) begin
            // Send Idle pattern
         end
         else begin
            drive_dut();
            // Calls item_done at the rising edge of vif.rw_done
            seq_item_port.item_done();
         end
      end
   end
endtask: drive_non_blocking

virtual task drive_dut();

   vif.write_data <= req_txn.write_data;
   vif.rw_addr    <= req_txn.rw_addr;
   vif.num_clks   <= req_txn.num_clks;

   if (req_txn.trans_type == RESET) begin
      vif.start_reset <= 1'b1;
      vif.start_rw    <= 1'b0;
      @(posedge vif.reset_done);
      vif.start_reset <= 1'b0;
   end else begin
      vif.start_reset <= 1'b0;
      vif.start_rw    <= 1'b1;
      if (req_txn.trans_type == READ) begin
         vif.r_wn <= 1'b1;
      end else begin
         vif.r_wn <= 1'b0;
      end
      @(posedge vif.rw_done);
      req_txn.read_data = vif.read_data;
      vif.start_rw     <= 1'b0;
   end
endtask: drive_dut
```

Fig. 8. UART Driver Source Code

From the perspective of the test environment, everything looks and operates just like a plain old vanilla UVM test environment. The tests and test environment can be used without modification; and at some later time when a project schedule allows, the UART UVC could be migrated to a standard UVC with minimal changes in the uart_driver code.

III. VERILOG BUS FUNCTIONAL MODEL

For a Verilog BFM, there are multiple options available to integrate the BFM into a UVM testbench. The method chosen depends on a few factors:

- Can the BFM code be modified?
- Does the BFM require parameters?
- Does the BFM use a parameterized interface?

These factors will determine which method is best suited to integrate a specific BFM into the UVM testbench. All of the following solutions allow using multiple BFMs in the testbench without having to hardcode instance identifiers in the test environment.

A. BFM Wrapper

If the existing Verilog BFM cannot be modified or is encrypted Verilog, one way to handle this is creating a wrapper around the BFM, just as was done for the preceding VHDL example.

B. Abstract and Concrete Class

It is also possible to use the Verilog BFM as is through the use of abstract and concrete classes [11]. In this case, the static testbench and dynamic environment take on slightly different forms, with the testbench defining a concrete class, which derives from an abstract class defined in the environment:

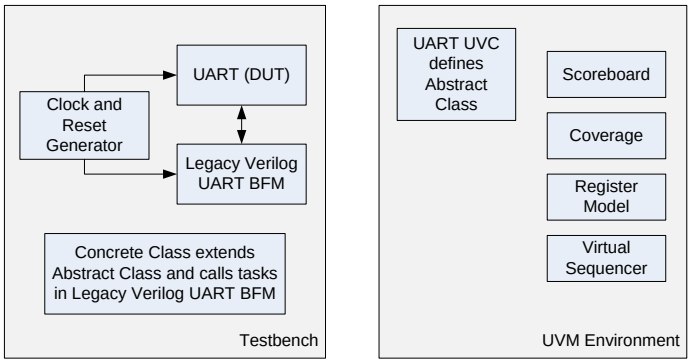

Fig. 9. Testbench and Environment Utilizing Concrete and Abstract Classes

```
module uart_tb;
  uart_bfm uart_bfm_inst(
   // BFM port connections
  );
  uart dut (
      // DUT port connections...
    );
    class uart_driver_concrete extends uart_driver;

      `uvm_component_utils(uart_driver_concrete)

     task uart_write(input logic [7:0] bit_data, input logic [2:0] addr_w);
       uart_bfm_inst.uart_write(bit_data, addr_w);
     endtask: uart_write

     task uart_read(input logic [2:0] addr_r, output logic [7:0] read_data);
       uart_bfm_inst.uart_read(addr_r);
       read_data = datout;
     endtask: uart_read

     task uart_reset(input integer  num_clocks);
       uart_bfm_inst.uart_reset(num_clocks);
     endtask: uart_reset

     task run_phase(uvm_phase phase);
        forever begin
          seq_item_port.get_next_item(req_txn);
          @(posedge clk_in) #1;
          case(req_txn.trans_type)
            RESET: uart_reset(req_txn.num_clks);
            READ: uart_read(req_txn.rw_addr, req_txn.read_data);
            WRITE: uart_write(req_txn.write_data, req_txn.rw_addr);
          endcase
          seq_item_port.item_done();
        end
      endtask: run_phase
    endclass: uart_driver_concrete

    initial begin
      uart_driver::type_id::set_type_override(uart_driver_concrete::get_type());
      run_test();
    end
endmodule: uart_tb

virtual class uart_driver extends uvm_driver #(uart_seq_item, uart_seq_item);

    `uvm_component_utils(uart_driver)

    pure virtual task uart_write(logic [7:0] bit_data, logic [2:0] addr_w);
    pure virtual task uart_read(logic [2:0] addr_r, output logic [7:0] read_data);
    pure virtual task uart_reset(integer  num_clocks);

endclass: uart_driver
```

Fig. 10. Testbench & Environment Code Defining the Abstract & Concrete Classes

Note that in Fig. 9, there is no virtual interface. Instead of a virtual interface, this method uses a type override in the testbench code to connect the dynamic test environment to the static testbench. The derived class, uart_driver_concrete, has access to the legacy Verilog BFM by nature of being defined in the same scope where the BFM is instantiated. When the dynamic test environment's base class member, uart_driver, is created as a uart_driver_concrete through a type override, it will have the same scope, and therefore, also have access to the BFM. See Fig. 10. This use of base and derived class thus gives the illusion of a virtual interface. The "magic" that allows everything to come together is the factory pattern [12]; the base class is registered with the UVM factory and overridden at runtime.

If the BFM has a parameterized interface, the abstract/concrete class is a good choice for connecting to the DUT (see The Problem with Parameters).

C. *Tasks in Interface*

If a user is fortunate enough to have a BFM that can be modified, by far the easiest approach is to copy and paste the tasks right into a SystemVerilog interface. The static testbench will have the virtual interface, which contains the BFM tasks. The test environment will contain the shell UART agent. The diagram below is nearly identical to the VHDL example shown previously. The environment is the same, but the static testbench shown here in Fig. 11 is different in that the virtual interface connects directly to the DUT, rather than to a BFM wrapper.

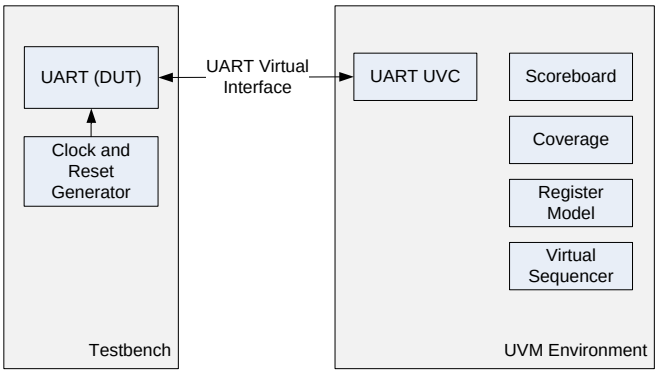

Fig. 11. Testbench and Environment for a Converted Legacy Verilog BFM

When a sequence is started on the UART agent, the uart_driver will call the appropriate task in the uart_uvc_if. This interface, the uart_uvc_if, is shown in Fig. 12.

```
interface uart_uvc_if (input clk, input reset);
  logic  [7:0]  datout;
  logic  [2:0]  addr;
  logic         cs;
  logic  [7:0]  datin;
  logic         nrw;
  // Additional interface signals...

  task uart_write;
    input [7:0] bit_data;
    input [2:0] addr_w;
    begin
      addr = addr_w;
      nrw = # (CLK_PRD) 1;
      datin = bit_data;
      cs = #(CLK_PRD) 0;
      cs = #(5*CLK_PRD) 1;
      nrw = #(CLK_PRD) 0;
      # CLK_PRD;
    end
  endtask

  task uart_read ;
    input    [2:0]  addr_r;
    output   [7:0]  read_data;
    begin
      // Code to drive a read transaction to the DUT...
    end
  endtask

  task uart_reset ;
    input integer num_clocks;
    begin
      // Code to drive a reset to the DUT...
    end
  endtask
endinterface: uart_uvc_if

class uart_driver extends uvm_driver #(uart_seq_item, uart_seq_item);
  virtual task drive_dut();
    if (req_txn.trans_type == RESET) begin
      vif.uart_reset(req_txn.num_clks);
    end else if (req_txn.trans_type == READ) begin
      vif.uart_read(req_txn.rw_addr, req_txn.read_data);
    end else begin
      vif.uart_write(req_txn.write_data, req_txn.rw_addr);
    end
  endtask: drive_dut
endclass: uart_driver
```

Fig. 12. BFM Copied into an Interface and Driver Task

IV. HARDWARE/SOFTWARE FLOW CONTROL

One problem encountered when verifying any SOC is answering the age-old question: which comes first, the software or the hardware? For companies with a large number of proficient software engineers, creating a simulation environment driven by code running on a processor makes sense. These companies verify the SOC by running real application code that was developed concurrent with the ASIC. On the other hand, if the software is verified using existing hardware, or after the real hardware is available, the simulation environment will be hardware centric. In both cases, there will be code running on the processor. The main difference is whether the hardware/UVM side or the software side is the main controller.

In either case, some type of synchronization/handshaking mechanism is required between the hardware and software. This synchronization is typically accomplished using shared memory (or a mailbox). The general flow of any test is as follows: the processor is loaded with code, the processor boots, and then the mailbox is monitored for commands that tell the hardware what to do (for example to send some Ethernet packets from the VIP to the DUT, or to send some data from the SPI port on the DUT to the SPI VIP). This is the "software-

centric" flow; the "hardware-centric" flow is similar, except once the processor finishes booting, control is passed over to the SystemVerilog side, and sequences are started. In the hardware-centric flow, the processor operations are distilled down to three commands: read, write, compare. This allows functional verification of the SOC hardware using a minimal amount of C (or assembly) code.

To pass control back and forth between the hardware and software requires monitoring and driving nodes inside the DUT, ideally without using hierarchical references to the shared memory. For the hardware-centric approach we will use a processor agent that writes commands (read, write, or compare) to a shared memory and sets the processor interrupt. The processor interrupt subroutine reads the shared memory, executes the requested command, and then clears the interrupt upon completion of the command.

The processor agent, shown in Fig. 13, is a standard UVM agent, except the driver has an extra interface. The event_if interface, defined in Fig. 14, contains events that the driver can use to control commands to the processor. For example, it could wait for the ev_isr_done event before calling item_done.

The monitor, shown in Fig. 14, sets these events anytime the memory contents are changed. In addition to being used by the driver, the events will be used by the sequences running in the test environment to control the execution. For example, at the event "ev_isr_done", a sequence would start sending packets.

To monitor the shared memory, we use a SystemVerilog bind construct, similar to the whitebox verification technique [3]. Writing or reading data to/from the shared memory is done using tasks in a virtual interface.

```
class firmware_agent extends uvm_agent;
  firmware_config    m_cfg;
  firmware_monitor   m_monitor;
  firmware_driver        m_driver;
  uvm_sequencer    #(firmware_item, firmware_item)  m_sequencer;

  virtual function void build_phase(uvm_phase phase);
     if(!uvm_config_db #(firmware_config)::get(this, "", "firmware_config", m_cfg)) begin
        `uvm_error("build_phase", "firmware_config not found")
      end
  endfunction: build_phase
  virtual function void connect_phase(uvm_phase phase);
    m_driver.vif_fw       = m_cfg.vif_fw;
    m_driver.vif_event = m_cfg.vif_event;
  endfunction: connect_phase

endclass: firmware_agent

interface firmware_if();
   task backdoor_write (logic [31:0] address, logic[31:0] data, logic [2:0] fw_cmd);
       top_tb.dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox[66:64] = fw_cmd;
       top_tb.dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox[63:32] = data;
       top_tb.dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox[31:0]   = address;
   endtask
   task backdoor_read (logic [31:0] address, logic[31:0] data, logic [2:0] fw_cmd);
      fw_cmd = top_tb.dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox[66:64];
      data       = top_tb.dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox[63:32];
      address = top_tb.dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox[31:0];
   endtask
endinterface: firmware_if

class firmware_driver extends uvm_driver #(firmware_item, firmware_item);
   virtual firmware_if    vif_fw;
   virtual event_if           vif_event;
     task run_phase(uvm_phase phase);
       forever begin
         seq_item_port.get_next_item(req_txn);
         case(req_txn.trans_type)
           READ: vif_fw.backdoor_read(req_txn.address, req_txn.data,req_txn.fw_cmd);
           WRITE: vif_fw.backdoor_write(req_txn.address, req_txn.data,req_tx.fw_cmd);
         endcase
         seq_item_port.item_done();
       end
     endtask: run_phase
endclass: firmware_driver
```

Fig. 13. Processor Agent

```
class firmware_monitor extends uvm_monitor;
 // Note: standard UVM phase code not shown
 firmware_item          mon_txn, t;
 virtual event_if       event_vif;

 task monitor_dut();
   forever begin
     @(event_vif.message) begin
      case (event_vif.message)
        boot_done: begin
          -> event_vif.boot_done_ev;
          `uvm_info({get_type_name(),":monitor_dut"},"Boot Done", UVM_NONE)
        end
        cpu_instr_start: begin
          -> event_vif.cpu_instr_start_ev;
          `uvm_info({get_type_name(),":monitor_dut"},"CPU Instruction Start", UVM_NONE)
        end
        cpu_instr_finish: begin
          -> event_vif.cpu_instr_finish_ev;
          `uvm_info({get_type_name(),":monitor_dut"},"CPU Instruction Finished", UVM_NONE)
        end
        cpu_isr_start: begin
          -> event_vif.cpu_isr_start_ev;
          `uvm_info({get_type_name(),":monitor_dut"},"CPU Interrupt Subroutine Start", UVM_NONE)
        end
        cpu_isr_finish: begin
          -> event_vif.cpu_isr_finish_ev;
          `uvm_info({get_type_name(),":monitor_dut"},"CPU Interrupt Subroutine Finished", UVM_NONE)
        end
        default : begin
          -> event_vif.warning_ev;
          `uvm_info({get_type_name(),":monitor_dut"},$psprintf("Invalid CPU message : %h",
                    firmware_vif.message), UVM_NONE)
        end
       endcase
       mon_txn.address = event_vif.address;
       mon_txn.data    = event_vif.data;
       mon_txn.fw_cmd  = event_vif.fw_cmd;
       $cast(t, mon_txn.clone());
       ap.write(t);
       `uvm_info({get_type_name(),":monitor_dut"}, t.convert2string(), UVM_MEDIUM)
     end
   end
 endtask: monitor_dut

interface event_if (input logic clock);
  logic [31:0] address;
  logic [31:0] data;
  logic [2:0]  fw_cmd;

  event boot_done_ev;
  event cpu_instr_start_ev;
  event cpu_instr_finish_ev;
  event cpu_isr_start_ev;
  event cpu_isr_finish_ev;
  event cpu_warning_ev;
endinterface: event_if

module top_tb();
    firmware_if      fw_if;

    top_wrapper     dut_wrapper ();

   bind dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox
        fw_event monitor_module_i (.*, .address(mem_addr), .data(mem_data),.fw_cmd(mem_cmd));

   initial begin
      uvm_config_db #(virtual firmware_if)::set(null, "uvm_test_top", "vif_firmware", fw_if);
   end
endmodule: top_tb

module fw_event   (
  input   logic                clock,
  input   logic [31:0]         address,
  input   logic [31:0]         data,
  input   logic [1:0]          fw_cmd  );

   import uvm_pkg::*;

   event_if ev_if(.*);
   initial begin
      uvm_config_db #(virtual event_if)::set(null, "uvm_test_top", "vif_event", ev_if);
   end
endmodule: fw_event
```

Fig. 14. Firmware Monitor

V. UVM Register Layer Backdoor Access

The UVM base class library has a built-in mechanism to allow backdoor access to registers, but there may be situations when registers are not accessible using the default DPI: a VHDL DUT or registers declared as multidimensional arrays, or the usage of the DPI has too great of an impact on simulation performance. There is a base class available for user-defined backdoor access that can be used instead. One approach [5] uses hierarchical access to the DUT to bypass the DPI and improve simulation performance, but the testbench is no longer completely decoupled from the DUT, and the register model cannot be contained in a package. An alternative is to move the hierarchical references to an interface.

In addition to keeping the test environment completely independent of the DUT, this approach has the added benefit of removing the DUT hierarchy from the register model, and can be used with a VHDL DUT (note: backdoor write from SystemVerilog to VHDL is only possible using simulator specific utilities). If the DUT hierarchy changes, but the registers remain the same, there is no need to update the register model since it doesn't contain any HDL hierarchical path constructs, i.e. add_hdl_path("top_tb.dut_wrapper.dut") or
status_reg_h.configure(this,null,"<hierarchical_path_to_register>").

```
class status_reg_backdoor extends uvm_status_reg_backdoor;
       `uvm_object_utils(status_reg_backdoor)
   // Task to write via backdoor

   virtual status_reg_backdoor_if status_reg_backdoor_vif;

   virtual task read(uvm_reg_item rw);
      do_pre_read(rw);
      status_reg_backdoor_vif.backdoor_read(rw);
      rw.status = UVM_IS_OK;
      do_post_read(rw);
   endtask

   virtual task write(uvm_reg_item rw);
      do_pre_write(rw);
      status_reg_backdoor_vif.backdoor_write(rw);
      rw.status = UVM_IS_OK;
      do_post_write(rw);
   endtask
endclass : status_reg_backdoor

interface reg_backdoor_if( );
  import uvm_pkg::*;
  task backdoor_read (uvm_reg_item rw);
   rw.value[0]=top_tb.dut_wrapper.dut.u_regs.status_rd_data_ff;
  endtask

  task backdoor_write (uvm_reg_item rw);
    top_tb.dut_wrapper.dut.u_regs.status_rd_data_ff= rw.value[0];
  endtask

endinterface : reg_backdoor_if

class top_reg_block extends uvm_reg_block;
   `uvm_object_utils(top_reg_block)
   rand status_reg   status_h;

  virtual function void build();
  m_status_backdoor =
   status_reg_backdoor::type_id::create("m_status_backdoor");
   status_h = status_reg::type_id::create("status_h");
   status_h.configure(this);
   status_h.build();
   status_h.set_backdoor(m_status_backdoor);
endfunction;

endclass: top_reg_block
```

Fig. 15. Register Backdoor Interface Example

Fig. 15 shows the backdoor read/write tasks for a SystemVerilog DUT. If the DUT is VHDL, a bind construct must be used as shown in Fig. 16.

```
module top_tb();
   reg_backdoor_if   status_reg_if(.*);

    top_wrapper     dut_wrapper ();

  initial begin
     uvm_config_db #(virtual reg_backdoor_if)::set(null,"*", "vif_reg_backdoor",
                          status_reg_if);
  end
endmodule

module vhdl_dut_tb();
  top_wrapper       dut_wrapper ();

  bind dut_wrapper.dut.u_top.u_regs vhdl_backdoor_module
        vhdl_backdoor_module_i (.*, .status(status_rd_ff));

endmodule

interface vhdl_reg_backdoor_if(input [31:0] status);
  import uvm_pkg;

  task backdoor_read (uvm_reg_item rw);
    rw.value[0]=status;
  endtask

  task backdoor_write (uvm_reg_item rw);
   `uvm_error("status_reg_backdoor", "Backdoor write to VHDL not allowed!")
  endtask

endinterface : reg_backdoor_if

module vhdl_backdoor_module(input wire [31:0] status );
   import uvm_pkg::*;

   vhdl_reg_backdoor_if vh_status_reg_if(.*);

   initial begin
      uvm_config_db #(virtual vhdl_reg_backdoor_if)::set(null,"*",
            "vif_vh_reg_backdoor", vh_status_reg_if);
   end

endmodule : vhdl_backdoor_module
```

Fig. 16 Register Backdoor Interface with VHDL DUT

VI. THE PROBLEM WITH PARAMETERS

For most designs that use industry standard communication or bus protocols, nearly all VIP use parameterized interfaces, like the one in Fig. 17. Parameterized interfaces require parameterized classes, which create a specialization for each combination of the generic class and actual parameter value. For designs with multiple combinations of parameters this will add complexity to the testbench and require using the type-based factory instead of the string-based factory. For a detailed description of using parameterized classes in OVM/UVM see [7].

```
interface ahb_vip_if #(parameter   NUM_MASTERS,
                                   NUM_SLAVES,
                                   ADDRESS_WIDTH,
                                   WDATA_WIDTH,
                                   RDATA_WIDTH)
                    (input clk, input reset);
  logic [ADDRESS_WIDTH-1:0]        haddr;
  logic [2:0]                      hburst;
  logic [3:0]                      hprot;
  logic [2:0]                      hsize;
  logic [1:0]                      htrans;
  logic [WDATA_WIDTH-1:0]          hwdata;
  logic                            hwrite;
  logic [RDATA_WIDTH-1:0]          hrdata;
  logic                            hreadyout;
  logic                            hresp;
  logic                            hsel;
  logic                            hready;
endinterface: ahb_vip_if

module top_tb ();
 typedef virtual ahb_vip_if #(1,10,32,32,32) vif_ahb_t;

 vif_ahb_t    uart_ahb_if_0();
 vif_ahb_t    spi_ahb_if_0();
 :
 :
endmodule
```

Fig. 17. Parameterized Interface Example

If your design contains only a minimal number of parameter values – multiple AHB agents all with the same address and data width for example – using a parameterized class is probably a good fit. On the other hand, if you have multiple parameterized interfaces that require many combinations of values, the best solution is to use an abstract base class/derived concrete class.

A. Abstract/Concrete Class

The abstract base class/concrete derived class really shines when specializations become too cumbersome. The example shown in Fig. 18 is implemented in much the same way as the example in section III.B of this paper. The testbench instantiates interfaces with varying parameters. The interface itself defines a concrete class, which extends an abstract base class and defines the tasks that will toggle the DUT's pins. This abstract class is a member of the driver and is overridden by the driver's call to get_config_object(). With this method, the only entity that needs to know the parameters for all the interfaces is the testbench itself.

```
module uart_tb;

  uart_uvc_if#( .DATA_WIDTH(32),
                .ADDR_WIDTH(4)) uart_if_0(.clk(clk_in), .reset(rst_in));
  uart_uvc_if#( .DATA_WIDTH(16),
                .ADDR_WIDTH(2)) uart_if_1(.clk(clk_in), .reset(rst_in));
  dut dut_inst ( // DUT port connections...);

 initial begin
   set_config_object("*","uart_if_0",uart_if_0.create_concrete_if("uart_if_0"),0);
   set_config_object("*","uart_if_1",uart_if_1.create_concrete_if("uart_if_1"),0);
   run_test();
 end
endmodule: uart_tb

interface uart_uvc_if #(DATA_WIDTH=8, ADDR_WIDTH=3)(input clk, input reset);

 logic  [DATA_WIDTH-1:0] datout;
 logic  [ADDR_WIDTH-1:0] addr;
 logic  [DATA_WIDTH-1:0] datin;

  class uart_if_concrete extends uart_if_base;
    task uart_write;
      input integer bit_data;
      input integer addr_w;
      begin
        addr = addr_w;
        nrw = # (CLK_PRD) 1;
        datin = bit_data;
        cs = #(CLK_PRD) 0;
        cs = #(5*CLK_PRD) 1;
        nrw = #(CLK_PRD) 0;
        # CLK_PRD;
      end
    endtask
  endclass

  uart_if_concrete concrete_if;

  function uart_if_base create_concrete_if(string name);
    concrete_if = new(name);
    return concrete_if;
  endfunction

endinterface: uart_uvc_if

class uart_driver#(integer ID=0) extends uvm_driver #(uart_seq_item, uart_seq_item);

  uart_if_base vif;

  function void build();
    super.build();
    get_config_object($sformatf("uart_if_%0d",ID),tmp,0)
  endfunction
endclass: uart_driver
```

Fig. 18. Abstract Concrete Parameterized Interface Example

VII. Conclusions

This paper illustrated where to apply the different testbench-DUT connection methods for a typical System on Chip (SOC) design. It included examples of how to apply the testbench to DUT connection methods [3] to a typical SOC design while adhering to a strict separation of dynamic and static elements. Using the techniques presented in this paper, users can create a highly reusable test environment that leverages their existing verification IP and allows replacing BFMs with UVCs as time permits.

The interface is the Swiss army knife of verification.

References

[1] "IEEE Standard for Verilog Hardware Description Language", IEEE STD 1364-2005, 2005

[2] "IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language", 1800-2012, 2012

[3] Dave Rich, "The Missing Link: The Testbench to DUT Connection", Proceedings of Design & Verification Conference , 2012

[4] Galen Blake, Steve Chappell, "One Compile to Rule Them All: An Elegant Solution for OVM/UVM Testbench Topologies", Proceedings of Design & Verification Conference, 2013

[5] Gaurav Gupta, Amit Sharma, Varun S, Abhisek Verma, "Switch the Gears of the UVM Register Package to cruise through the street named "Register Verification"", Proceedings of Design & Verification Conference , 2013

[6] Dave Rich, Jonathan Bromley, "Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches", Proceedings of Design & Verification Conference , 2008

[7] Bryan Ramirez, Michael Horn, "Parameters and OVM – Can't They Just Get Along?" Design & Verification Conference 2011

[8] Shashi Bhutada, "Polymorphic Interfaces: An Alternative for SystemVerilog Interfaces", Verification Horizons - Volume 7, Issue 3 – November, 2011

[9] Wayne Yun, Shihua Zhang, "Deploying Parameterized Interface with UVM", Proceedings of Design & Verification Conference, 2013

[10] John Cooley, "VHDL the new Latin", EE Times, April 7, 2003

[11] "Two Kingdoms Factory." Mentor Graphics Verification Academy. https://verificationacademy.com/cookbook/connect/twokingdomsfactory

[12] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns, Elements of Reusable Object-Oriented Software" Addison-Wesley Publishing Company, Reading Massachusetts, 1994.