VMM 1.2 – SPI Tutorial

Doug Smith, Doulos, February 2010

Introduction

In this tutorial, a simple Serial Peripheral Interface (SPI) design is used from OpenCores.org (http://www.opencores.org/project,spi). The object is to take you step-by-step through implementing a simple VMM verification environment and showcasing some of the new features of VMM 1.2. This tutorial will use a bottom-up approach in creating a verification testbench.

Getting Started

First, you need a working copy of Synopsys' VMM 1.2 library. VMM can be freely downloaded from www.vmmcentral.org. Download and follow the installation directions found in the included README.txt. If you are using a non-VCS simulator, then you will need to compile the additional regular expression string matching library included with the VMM distribution. See your simulator's documentation for linking in DPI code into your simulations.

Next, go to OpenCores.org and download the free SPI design. This design uses a Wishbone system interface to configure and control the SPI. If you are not familiar with the Wishbone or SPI protocols, then have a look at the included SPI documentation (also found here) and at the Wishbone specification found on OpenCores here.

You will also need to download the source code for this tutorial from the same location as this PDF file on www.doulos.com.

Interfacing with the design

The easiest way for our testbench to interact with the design under test is using a SystemVerilog interface. Our interface will have all of the Wishbone and SPI protocol signals and a modport for each protocol:

```
interface dut_intf;

     // Wishbone signals
     logic        wb_clk_i;         // master clock input
     logic        wb_rst_i;         // sync active high reset
     logic  [4:0] wb_adr_i;         // lower address bits
     logic [31:0] wb_dat_i;         // databus input
     logic [31:0] wb_dat_o;         // databus output
     logic  [3:0] wb_sel_i;         // byte select inputs
     logic        wb_we_i;          // write enable input
     logic        wb_stb_i;         // stobe/core select signal
     logic        wb_cyc_i;         // valid bus cycle input
     logic        wb_ack_o;         // bus cycle ack output
     logic        wb_err_o;         // termination w/ error
     logic        wb_int_o;         // int req signal output

     // SPI signals
     logic  [7:0] ss_pad_o;         // slave select
     logic        sclk_pad_o;       // serial clock
     logic        mosi_pad_o;       // master out slave in
     logic        miso_pad_i;       // master in slave out

     // Wishbone signals
     modport wb (output wb_clk_i, wb_rst_i, wb_adr_i, wb_dat_i,
                        wb_sel_i, wb_we_i, wb_stb_i, wb_cyc_i,
                  input wb_dat_o, wb_ack_o, wb_err_o, wb_int_o);

     // SPI signals
     modport spi (output miso_pad_i, input ss_pad_o, sclk_pad_o, mosi_pad_o);

endinterface
```

Since our SPI design is written in traditional Verilog, we need to create a wrapper around it to connect the signals of the design with our interface:

```
module spi_wrapper ( interface dut_if );

   // Instantiate and connect up the DUT
   spi_top spi (   // Wishbone interface
                .wb_clk_i ( dut_if.wb_clk_i ),
                .wb_rst_i ( dut_if.wb_rst_i ),
                .wb_adr_i ( dut_if.wb_adr_i ),
                .wb_dat_i ( dut_if.wb_dat_i ),
                .wb_dat_o ( dut_if.wb_dat_o ),
                .wb_sel_i ( dut_if.wb_sel_i ),
                .wb_we_i  ( dut_if.wb_we_i  ),
                .wb_stb_i ( dut_if.wb_stb_i ),
                .wb_cyc_i ( dut_if.wb_cyc_i ),
                .wb_ack_o ( dut_if.wb_ack_o ),
                .wb_err_o ( dut_if.wb_err_o ),
                .wb_int_o ( dut_if.wb_int_o ),

                // SPI signals
                .ss_pad_o  ( dut_if.ss_pad_o   ),
                .sclk_pad_o( dut_if.sclk_pad_o ),
                .mosi_pad_o( dut_if.mosi_pad_o ),
                .miso_pad_i( dut_if.miso_pad_i )
   );
```

```
endmodule
```

With our interfacing to the design complete, we can now start creating our VMM 1.2 class-based testbench in a bottom-up manner.

The transaction object

All communication between testbench components is made using *transaction objects*. Transaction objects in VMM are built by extending the *vmm_data* class. We will start by creating a transaction object for the Wishbone interface with the following members:

Address	Address of transaction
Data	Data payload
Kind	Specifies the data kind of the transaction (RX or TX)

VMM can automatically create convenience methods for our transaction object like copy(), print(), compare(), etc. This is accomplished by using the VMM shorthand data member macros to define all the members in the transaction object.

Our transaction can also include a class factory macro that will allow us to swap out derived classes from our testcases later using the class factory mechanism. For this to work, we will need to define a class constructor, copy(), and allocate() methods, but these are provided automatically for us when using the shorthand data member macros. The transaction object can be written as follows:

```
class wb_spi_trans extends vmm_data;

   // Fields in the SPI registers
   rand bit [AWIDTH-1:0] addr;
   rand bit [DWIDTH-1:0] data;
   rand trans_t          kind;

   `vmm_typename( wb_spi_trans )

   // Create the constructor, copy(), and allocate() functions
   `vmm_data_member_begin( wb_spi_trans )
        `vmm_data_member_scalar( addr,    DO_ALL )
        `vmm_data_member_scalar( data,    DO_ALL )
        `vmm_data_member_enum  ( kind,    DO_ALL )
   `vmm_data_member_end( wb_spi_trans )

   // Class factory so this transaction can be swapped with derived classes.
   `vmm_class_factory( wb_spi_trans )

endclass : wb_spi_trans
```

VMM can also create a parameterized channel class for us automatically by using the vmm_channel() macro so we will include that along with our transaction definition:

```
// Create a channel class for the wb_spi_trans called "wb_spi_trans_channel"
`vmm_channel ( wb_spi_trans )
```

For our Wishbone and SPI monitors, we will create a slightly different transaction. The SPI design can transfer up to 128 bits, but there is no way of knowing on the SPI interface how many bits need to be transferred so our Wishbone monitor will store each 32 bit data write to the SPI design's registers and then send the 128 bit data and the control register's character length (CHAR_LEN) to the scoreboard for checking with the SPI output. Also, this transaction will include coverage terms so we can see what random stimulus has been generated. Our monitor and scoreboard transaction will have the same members as above with the additional control register member:

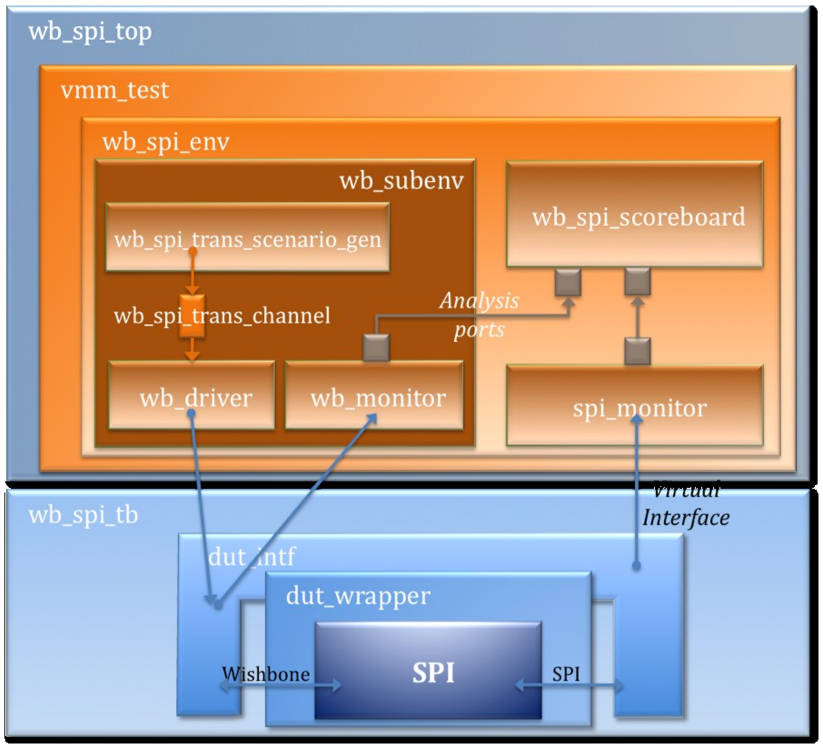
```
class mon_sb_trans extends vmm_data;

   // Fields in the SPI registers
   rand bit [AWIDTH-1:0]     addr;
   rand bit [(DWIDTH*4)-1:0] data;
   rand trans_t              kind;
   rand ctrl_t               ctrl;

   `vmm_typename( mon_sb_trans )

   // Define function coverage
   covergroup cg;
        coverpoint addr {
                bins valid[] = { SPI_TX_RX0, SPI_TX_RX1, SPI_TX_RX2,
                                 SPI_TX_RX3, SPI_CTRL, SPI_DIVIDER, SPI_SS };
                illegal_bins invalid = default;
        }
        char_len: coverpoint ctrl.char_len {
                bins tiny = { [1:43] };
                bins mid  = { [44:85] };
                bins big  = { 0, [86:127] };
        }
        coverpoint kind;
   endgroup

   `vmm_data_new( mon_sb_trans )
   function new();
        super.new( log );
        cg = new;       // Create the coverage
   endfunction

   // Function to sample the coverage
   function void sample_cov();
        cg.sample();
   endfunction


   // Create the constructor, copy(), and allocate() functions
   `vmm_data_member_begin( mon_sb_trans )
        `vmm_data_member_scalar( addr,    DO_ALL )
        `vmm_data_member_scalar( data,    DO_ALL )
        `vmm_data_member_scalar( ctrl,    DO_ALL )
        `vmm_data_member_enum  ( kind,    DO_ALL )
   `vmm_data_member_end( mon_sb_trans )

endclass : mon_sb_trans
```

Observe, the data width in this transaction is 4 times larger than the *wb_spi_trans*, and we have included the control register member *ctrl.* We have also included a constructor so we can instantiate the covergroup. By default, the *`vmm_data_member_begin/end* macros create an automatic constructor so we use the *`vmm_data_new* macro to prevent its creation and define our own. For convenience, we have also defined a function called *sample_cov()* so we can easily indicate when to sample the coverage.

Creating the testbench

In order to simplify the testbench for this tutorial, we will focus primarily on the read and write transactions through the Wishbone interface of our design and checking that the SPI interface correctly responds. All SystemVerilog testbenches require a module to instantiate the design so we need a top-level module that instantiates the interface (called *dut_intf*), the design wrapper (*dut_wrapper*), and the design itself. We will call this top-level testbench *wb_spi_tb*.

The class-based portion of the testbench will be constructed using VMM. Each test in VMM is derived from vmm_test and instantiates the environment that it wishes to execute on. The testbench environment is derived from vmm_env and instantiates the appropriate testbench components. For the Wishbone interface, we will create a self-contained verification unit called a *sub-environment*, which derives from vmm_subenv. Inside this sub-environment, we have a scenario generator that creates the stimulus, a driver to drive it, and a monitor to monitor the transactions and pass them over to the scoreboard. Since the environment is simplified for this tutorial, only a monitor will be needed to monitor and send the SPI output to the scoreboard. In order to create the VMM environment and start the test case, we will use a program block and call it *wb_spi_top*. This environment is illustrated below:



The Wishbone sub-environment

Wishbone driver

We'll start creating the Wishbone sub-environment by creating the driver. The driver receives transactions from a scenario generator through a VMM channel, and then converts the transaction into Wishbone read/write operations. With the new VMM 1.2 implicit phasing, we can simplify our component development by defining the driver's functionality across the different implicit phase methods.

The driver's interaction with the design happens through a virtual interface passed into the driver during the implicit *connect_ph* phase. The interface is placed in a class wrapper so that it can be easily passed to the driver using the VMM 1.2 configuration mechanism via the *vmm_opts::get_object_obj()* method. This allows us to swap out the interface later to inject errors, change the signal routing, or perform other functionality. In the *start_of_sim_ph()* phase, we'll check that the virtual interface is correctly connected before we start using it (this avoids a fatal error from a null pointer access).

```
class wb_driver extends vmm_xactor;

   virtual dut_intf.wb          dut_if;          // Virtual interface
   wb_spi_trans_channel         in_chan;

   `vmm_typename( wb_driver )

   // Constructor
   function new ( string name, string inst, vmm_group parent );
        super.new( name, inst,, parent );
   endfunction : new


   // Connect phase
   function void connect_ph();
        bit is_set;
        dut_if_wrapper  if_wrp;

        // Grab the interface wrapper for the virtual interface
        if ( $cast( if_wrp, vmm_opts::get_object_obj( is_set,
                                        this, "dut_intf" ))) begin
                if ( if_wrp != null )
                        this.dut_if = if_wrp.dut_if;
                else
                        `vmm_fatal( log, "Cannot find DUT interface!!" );
        end
   endfunction : connect_ph


   // Start of simulation phase
   function void start_of_sim_ph();
        if ( dut_if == null)
                `vmm_fatal( log, "Virtual interface not connected!" );
   endfunction

   ...
endclass
```

Next, we need to define the core functionality of the driver. If our driver was derived from the new *vmm_group* class, then we could define the functionality in the *run_ph()* method; however, we're going to use the recommended *vmm_xactor* class since it fits into the traditional VMM methodology and works well with the new features of VMM 1.2. The body of a vmm_xactor is placed inside of a *main()* method and forked off with the parent class' main() method:

```
// main() task - do the work here
task main;
   fork
      super.main();
      begin : main_fork
```

```
        `vmm_note(log, "Starting the WB driver ...");

        forever begin : drive_bus

            // Main functionality of the driver - drive the bus
            ...

        end : drive_bus
      end : main_fork
   join_none
endtask : main
```

Once the functionality is defined, then our driver is completed and ready to plug into the environment (see downloadable source code for the full Wishbone driver implementation).

Wishbone monitor

The Wishbone monitor looks identical in structure to the driver, except for some additional TLM analysis ports that will send monitor transactions to the scoreboard. Likewise, we will include some coverage terms in our monitor (though these could be placed in a separate coverage collector or in the scoreboard). Because we included an analysis port and covergroup, we will need to create these objects either in the monitor's constructor or implicit *build_ph* phase:

```
class wb_monitor extends vmm_xactor;

   // Interface to the WB DUT interface
   virtual dut_intf.wb          dut_if;

   // Communication port
   vmm_tlm_analysis_port #( wb_monitor, mon_sb_trans) sb_ap;

   mon_sb_trans                 trans = new;

   // Constructor
   function new ( string name, string inst, vmm_group parent );
        super.new( name, inst,, parent );
   endfunction : new


   // Build phase
   function void build_ph();
        super.build_ph();
        `vmm_note( this.log, "Building analysis port..." );
        sb_ap = new ( this, "Analysis port to scoreboard" );
   endfunction : build_ph

   // Connect phase
   function void connect_ph();
```

```
        bit is_set;
        dut_if_wrapper  if_wrp;

        // Grab the interface wrapper for the virtual interface
        if ( $cast( if_wrp, vmm_opts::get_object_obj( is_set,
                                        this, "dut_intf" ))) begin
                if ( if_wrp != null )
                        this.dut_if = if_wrp.dut_if;
                else
                        `vmm_fatal( log, "Cannot find DUT interface!!" );
        end
   endfunction : connect_ph


   // Start of simulation phase
   function void start_of_sim_ph();
        if ( dut_if == null)
                `vmm_fatal( log, "Virtual interface not connected!" );
   endfunction


   // main() body
   task main;
     fork
        super.main();
        begin : main_fork

              // Debug info
              `vmm_note( this.log, "Monitoring the WB bus");


              // Monitor the bus
              forever begin : monitor_bus

                  // Main monitor functionality - read the bus
                  ...

                  trans.sample_cov();     // Sample the coverage
                  ...
                  sb_ap.write( trans );   // Send transaction to scoreboard

              end : monitor_bus
          end : main_fork
     join_none
   endtask : main
endclass : wb_monitor
```

Notice that an analysis port is declared by specifying the initiator and transaction type as its parameter:

```
vmm_tlm_analysis_port #( wb_monitor, mon_sb_trans ) sb_ap;
```

Later in the scoreboard, we will define an analysis export that will implement the *write()* method used to send the data and check it in the scoreboard. (See the downloadable source code for the monitor's full implementation).

Wishbone scenario generator

The scenario generator generates transactions by running user-defined scenarios and then passing them on to the driver. While writing the scenarios themselves requires a bit of work, creating the scenario generator could not be easier. The scenario generator is created using one line of code:

```
`vmm_scenario_gen( wb_spi_trans, "WB/SPI Scenarios" )
```

That's it!! Nothing more complicated is required. Now, a scenario generator class called *wb_spi_trans_scenario_gen* is defined and we can use it in our Wishbone sub-environment.

Creating a sub-environment

With the scenario generator, driver, and monitor defined, we can start constructing our verification unit referred to as a VMM sub-environment. In the traditional VMM methodology, we would normally create our sub-environment by extending *vmm_subenv*, but so we can take advantage of the new VMM 1.2 implicit phasing, we will use a *vmm_group*.

Practically speaking, a sub-environment just creates a wrapper around our testbench components so we can form a self-contained verification unit. Its main purpose is to instantiate its components and connect them together. We will also use the new VMM 1.2 configuration mechanism to set the number of scenarios to generate by using the *vmm_unit_config_begin/end* macros. Using these macros, the testcase can pass a value into the sub-environment or we can set a default value if nothing is set. Taking a look at our testbench diagram, you will notice that a channel is used to connect the scenario generator and driver, which our sub-environment will connect for us. Since the scenario generator executes independently of the VMM 1.2 implicit phasing, we will start it in the *start_of_sim_ph* phase and stop it in the *shutdown_ph* phase.

We also need a way to end our testcase, but not before all the components are finished and inactive. This can be accomplished by using VMM's consensus mechanism. The consensus mechanism uses a simple voting system where components either consent or object to ending the test. Each component registers its vote with the consensus object, and its objecting vote prevents simulation from finishing. Within the voting components, a VMM notification object can be used to indicate if a component is idle or busy. The *register_xactor()* method forks a process to wait for these notifications and update the component's consensus vote. We will register each of these components in the sub-environment's *connect_ph* phase with the vmm_group's *vote* consensus member, which is used by the *run_ph* phase to wait for all the components to be idle before finishing simulation.

Here is the complete source of our Wishbone sub-environment:

```
class wb_subenv extends vmm_group;

   wb_monitor                    wb_mon;
   wb_driver                     wb_drv;
   wb_spi_trans_scenario_gen     scn_gen;
   wb_spi_trans_channel          gen_to_drv_chan;
   int                           num_scenarios;

   `vmm_typename( wb_subenv )

   // Configuration mechanism for controlling the number of scenarios
   `vmm_unit_config_begin( wb_subenv )
        `vmm_unit_config_rand_int( num_scenarios, 5, "Number of scenarios to
run", 0, "DO_ALL" )
   `vmm_unit_config_end( wb_subenv )



   // Constructor
   function new ( string name, string inst, vmm_group parent );
        super.new ( "wb_subenv", inst, parent );
   endfunction : new

   // Build phase
   function void build_ph();
        super.build_ph();

        `vmm_note( this.log, "Creating subenvironment ..." );
        gen_to_drv_chan = new( "gen_to_drv_chan", " Channel" );
        wb_drv          = new( "wb_driver", "wb_drv", this );
        wb_mon          = new( "wb_monitor", "wb_mon", this );
        scn_gen         = new( "scn_gen", 0 );
   endfunction : build_ph

   // Connect phase
   function void connect_ph();
        // Connect up the channel between the generator and the driver
        wb_drv.in_chan = gen_to_drv_chan;
        scn_gen.out_chan = gen_to_drv_chan;

        // Register end-of-test consensus
        vote.register_xactor( wb_drv );
        vote.register_xactor( wb_mon );
        vote.register_xactor( scn_gen );
        vote.register_channel( gen_to_drv_chan );
   endfunction : connect_ph

   // Start the stimulus generator phase
   function void start_of_sim_ph();
```

```
        super.start_of_sim_ph();

        // Use configuration value to set the number of scenarios
        scn_gen.stop_after_n_scenarios = num_scenarios;
        scn_gen.start_xactor();           // Start the scenario generator
                                          // (Note, the testcase sets the
                                          // scenario to run).
   endfunction

   // Shutdown phase
   task shutdown_ph();
        scn_gen.notify.wait_for(wb_spi_trans_scenario_gen::DONE);
        `vmm_note( this.log, "Scenario generation done..." );

        // Stop the scenario generator
        scn_gen.stop_xactor();
   endtask
endclass : wb_subenv
```

The SPI monitor and scoreboard

The remaining components needed to finish our testbench are a monitor for the SPI bus and a scoreboard checker. The structure of our SPI monitor is just like our Wishbone monitor. We include an analysis port to pass transactions over to the scoreboard and we provide the main functionality in the *main()* task. Since the SPI protocol is so simple, here is the full implementation of our SPI monitor:

```
class spi_monitor extends vmm_xactor;

   // Interface to the SPI DUT interface
   virtual dut_intf.spi dut_if;

   // Communication ports
   vmm_tlm_analysis_port #( spi_monitor, mon_sb_trans ) sb_ap;

   // Constructor
   function new ( string name, string inst, vmm_group parent );
        super.new( name, inst,, parent );
   endfunction : new


   // Build phase
   function void build_ph();
        super.build_ph();
        sb_ap = new ( this, "Analysis port to scoreboard" );
   endfunction : build_ph


   // Connect phase
```

```
function void connect_ph();
     bit is_set;
     dut_if_wrapper  if_wrp;

     // Grab the interface wrapper for the virtual interface
     if ( $cast( if_wrp, vmm_opts::get_object_obj( is_set,
                                 this, "dut_intf" ))) begin
             if ( if_wrp != null )
                     this.dut_if = if_wrp.dut_if;
             else
                     `vmm_fatal( log, "Cannot find DUT interface!!" );
     end
endfunction : connect_ph


// Start of simulation phase
function void start_of_sim_ph();
     if ( dut_if == null)
             `vmm_fatal( log, "Virtual interface not connected!" );
endfunction


// main() body
task main();
  fork
   super.main();
   begin : main_fork

     `vmm_note( this.log, "Monitoring the SPI bus ..." );

     forever begin
        bit [6:0] i = 0;
        mon_sb_trans trans = new;             // New transaction

        wait ( ~dut_if.ss_pad_o );            // Wait for a tx to begin

        this.notify.reset(XACTOR_IDLE);
        this.notify.indicate(XACTOR_BUSY);    // Don't end yet!

        trans = new;                          // New transaction

        `vmm_note( this.log, "SPI bus ready to transmit...");

        fork
             begin
                for ( i = 0; ~dut_if.ss_pad_o[0]; i++ ) begin
                        @(posedge dut_if.sclk_pad_o);
                        trans.data[i] = dut_if.mosi_pad_o;
                end
                end
                wait ( dut_if.ss_pad_o[0] );    // Transfer finished
           join_any
           disable fork;

           //
           // Send the transaction to the scoreboard.
           //
           if ( i ) begin              // Make sure something was transferred
                `vmm_note( this.log, "Sending transaction to scoreboard..."
);
                trans.display();
                sb_ap.write( trans );
           end

           this.notify.reset(XACTOR_BUSY);
           this.notify.indicate(XACTOR_IDLE);   // Ok, not busy
        end

      end : main_fork
     join_none
   endtask : main
endclass : spi_monitor
```

For our scoreboard, we simply need to extend the VMM data-stream *vmm_sb_ds* class, which has all the functionality needed for checking. In order for it to work with our custom transaction type, we need to define a *compare()* function, and create the implementation for our TLM analysis exports that receive the actual ("inp") and expected ("exp") data. Here is what our scoreboard will look like:

```
`include "vmm_sb.sv"

typedef vmm_sb_ds_typed#( mon_sb_trans ) sb_t;

class wb_spi_scoreboard extends sb_t;

        `vmm_tlm_analysis_export( _inp )
        `vmm_tlm_analysis_export( _exp )

        vmm_tlm_analysis_export_inp#( wb_spi_scoreboard, mon_sb_trans )
inp_ap = new( this, "input analysis port" );
        vmm_tlm_analysis_export_exp#( wb_spi_scoreboard, mon_sb_trans )
exp_ap = new( this, "expect analysis port" );


        function new( string name );
                super.new( name );
        endfunction
        function bit compare( mon_sb_trans actual, mon_sb_trans expected );
                bit     [127:0] mask = '0;

                // Create a mask based on the number of bits transmitted
                for (int i = 0; i < (expected.ctrl & 'h3f); i++)
                        mask[i] = 1;

                // Only need to compare the data
                return ((actual.data & mask) == (expected.data & mask));
        endfunction

        // Provide an implementation for the TLM analysis ports
        function void write_inp(int id=-1, mon_sb_trans trans);
                // Actual value so check to see if it's correct
                void'(this.expect_in_order( trans, id ));
        endfunction

        function void write_exp(int id=-1, mon_sb_trans trans);
                // Expect this transaction
                this.exp_insert( trans, id );
        endfunction

endclass : wb_spi_scoreboard
```

Since the scoreboard classes are not found in the standard VMM source files, *vmm_sb.sv* must be included. Prior to VMM 1.2, vmm_sb_ds was not parmeterized, but now we can create a specialized scoreboard that automatically handles our transaction without the need for up and down-casting by using the *vmm_sb_ds_typed* class. The *vmm_tlm_analysis_export* macro creates a parameterized analysis export that we can then instantiate inside our scoreboard and subsequently connect in our testbench environment. Also, the parameterized analysis export defines the *write()* method to call the specific task with the corresponding *write_<name>* so we can give each analysis export its own unique behavior. In this case, we define the "exp" analysis port to place the expected data into the scoreboard, and the "inp" analysis port to pass the actual data for comparison (using the *compare()* method).

The testbench environment

Now that each component in the testbench is defined, we can bring them all together into one unified testbench environment. If you recall in our testbench diagram, there are several TLM connections that need to be made between the monitors and the scoreboard. The TLM bind method will be used to connect these.

Since the monitors and drivers get a copy of the virtual interface through the VMM configuration mechanism, a testbench would not normally be required to also have a reference. However, our SPI design needs some specific initialization before running any tests. These initializations require driving signals into the design so the testbench environment also needs a reference to the virtual interface. A perfect place for these initializations is during the VMM 1.2 implicit phase called *reset_ph*. This phase will run before the testcase starts so the environment can bring up the design in a good state. Here is what our environment will look like:

```
class wb_spi_env extends vmm_group;

   virtual dut_intf      dut_if;
   wb_subenv            wb_sub;
   spi_monitor          spi_mon;
   wb_spi_scoreboard    wb_spi_sb;

   `vmm_typename( wb_spi_env )

   // Constructor
   function new ( string name, string inst, vmm_group parent = null );
        super.new( name, inst, parent );
   endfunction : new


   // Build the member objects
   function void build_ph();
        super.build_ph();

        // Create the WB subenvironment
        wb_sub = new( "wb_subenv", "wb_sub", this );

        // Create the SPI monitor
        spi_mon = new( "spi_monitor", "spi_mon", this );

        // Create the scoreboard
        wb_spi_sb = new( "wb_spi_sb" );

        `vmm_note( this.log, "Built wb_spi_env ..." );
   endfunction : build_ph


   // Connect everything together
   function void connect_ph();
        bit is_set;
        dut_if_wrapper  if_wrp;

        `vmm_note( this.log, "Connect phase... " );

        // Grab the interface wrapper for the virtual interface
        if ( $cast( if_wrp, vmm_opts::get_object_obj( is_set,
                                        this, "dut_intf" ))) begin
                if ( if_wrp != null )
                        this.dut_if = if_wrp.dut_if;
                else
                        `vmm_fatal( log, "Cannot find DUT interface!!" );
        end

        // Hook up the monitors to the scoreboard
        wb_sub.wb_mon.sb_ap.tlm_bind( wb_spi_sb.exp_ap );
        spi_mon.sb_ap.tlm_bind( wb_spi_sb.inp_ap );

        // Add monitor to end-of-test consensus
        vote.register_xactor( spi_mon );
   endfunction : connect_ph


   // Start of simulation phase
   function void start_of_sim_ph();
        if ( dut_if == null)
                `vmm_fatal( log, "Virtual interface not connected!" );
   endfunction


   // Reset the DUT
   task reset_ph();
        super.reset_ph();

        // Initial values
        `vmm_note( this.log, "Reseting the DUT ... " );
        dut_if.wb_adr_i  = {AWIDTH{1'bx}};
        dut_if.wb_dat_i  = {DWIDTH{1'bx}};
        dut_if.wb_cyc_i  = 1'b0;
        dut_if.wb_stb_i  = 1'bx;
        dut_if.wb_we_i   = 1'hx;
        dut_if.wb_sel_i  = {DWIDTH/8{1'bx}};

        // Reset the DUT
        dut_if.wb_rst_i = 0;
        #20;
        dut_if.wb_rst_i = 1;
        #200;
        dut_if.wb_rst_i = 0;
        #20;
        `vmm_note( this.log, "The DUT is now reset." );
   endtask : reset_ph

endclass : wb_spi_env
```

Creating a test case and scenario library

With the environment defined, we can now turn to creating stimulus for our design. While we often think of our testcases as creating our stimulus, when we use scenarios that often is not the case.

Instead, we define our stimulus in a library of hierarchical scenarios—i.e., scenario upon scenario calling other scenarios—and our tests simply decide which scenarios to execute.

To create our scenario library, we can start with the most basic Wishbone scenario—a read or write bus operation. When we defined our scenario generator as `vmm_scenario_gen(wb_spi_trans, …), the macro automatically defined for us a parameterized single-stream scenario called *wb_spi_trans_scenario*. Using this new class, we can create a simple bus operation as follows:

```
// wb_op_scn - Sequence to perform a WB operation (read or write)
class wb_op_scn extends wb_spi_trans_scenario;

        rand bit [AWIDTH-1:0] my_addr;
        rand bit [DWIDTH-1:0] my_data;
        rand trans_t          my_kind;

        `vmm_typename ( wb_op_scn )

        function new();
                define_scenario( "wb_op_scn", 0 );
        endfunction

        virtual task apply( wb_spi_trans_channel channel,
                            ref int unsigned n_inst );

                wb_spi_trans    tr = new();

                if ( tr.randomize with {
                        addr == my_addr;
                        data == my_data;
                        kind == my_kind;
                } ) begin
                        tr.display();
                        channel.put( tr );
                        n_inst++;         // Increment transaction count
                end
        endtask
endclass
```

The *my_addr*, *my_data*, and *my_kind* members are random control knobs for our scenario. When the scenario is invoked from other scenarios, it can be directed by setting either those knobs or leaving them to the *randomize()* function. The scenario generator requires an *apply()* method so we have included one that creates a transaction, randomizes it, and sends it on through the scenario generator's channel, which is passed as an argument to *apply()*. We also call *define_scenario()* in the constructor to register the scenario, define its maximum transaction stream length, and its scenario kind. A scenario by default will create a random number of transactions so we set our maximum transaction stream length to 0 since we want to control the creation of the transaction objects.

Now, we can create a read and write scenario by simply extending our basic Wishbone scenario and constraining the kind to be RX or TX, respectively:

```
// wb_read - Scenario to perform a WB read transaction
class wb_read_scn extends wb_op_scn;
        `vmm_typename ( wb_read_scn )

        function new();
                define_scenario( "wb_read_scn", 0 );
        endfunction

        constraint kind_c { my_kind == RX; }
endclass

// wb_write - Scenario to perform a WB write transaction
class wb_write_scn extends wb_op_scn;
        `vmm_typename ( wb_write_scn )

        function new();
                define_scenario( "wb_write_scn", 0 );
        endfunction

        constraint kind_c { my_kind == TX; }
endclass
```

We can instantiate the *wb_read_scn* or *wb_write_scn* scenarios inside of other scenarios to create what is called a ***hierarchical scenario***.

With these scenarios defined, we can now create a scenario that will write from the Wishbone interface to the SPI output by instantiating the read and/or write scenarios and calling its *apply()* method. Such a scenario is referred to as a *hierarchical scenario*. Our SPI design will start transmitting when its GO_BSY bit in the control register is set so we will create a scenario to setup all the SPI transfer registers and then tell it to go:

```
// wb_to_spi - Scenario to send data from the WB to the SPI interface.
class wb_to_spi_scn extends wb_spi_trans_scenario;
        rand bit [3:0][DWIDTH-1:0] data;
        rand ctrl_t      ctrl;                  // Controls DUT configuration
        rand divider_t   divider;
        rand ss_t        ss;
        wb_write_scn     wb_write = new();      // Instance of write scenario

        constraint divider_c {
          divider[31:16] == 0;                  // Reserved
          divider[15: 0] inside { [ 0 : 100 ] };  // Reasonable clk freq
        }
        constraint ctrl_c {
          // Configure control reg for SPI transfer (GO_BSY not set yet)
          ctrl[31:7] == ((A_SS_BIT | IE_BIT | LSB_BIT | TX_NEG | RX_NEG) >> 7
);
        }
        constraint ss_c { ss == '1; }           // Only one slave select

        `vmm_typename ( wb_to_spi_scn )

        function new();
                define_scenario( "wb_to_spi_scn", 0 );
        endfunction

        virtual task apply( wb_spi_trans_channel channel,
                            ref int unsigned n_inst );

                // Write data into the TX registers
                wb_write.randomize with { my_addr == SPI_TX_RX0;
                                          my_data == data[0]; };
                wb_write.apply( channel, n_inst );

                if ( ctrl.char_len > 32 || ctrl.char_len == 0 ) begin

                   wb_write.randomize with { my_addr == SPI_TX_RX1;
                                             my_data == data[1]; };
                   wb_write.apply( channel, n_inst );
                end

                if ( ctrl.char_len > 64 || ctrl.char_len == 0 ) begin
                   wb_write.randomize with { my_addr == SPI_TX_RX2;
                                             my_data == data[2]; };
                   wb_write.apply( channel, n_inst );
                end

                if ( ctrl.char_len > 96 || ctrl.char_len == 0 ) begin
                   wb_write.randomize with { my_addr == SPI_TX_RX3;
                                             my_data == data[3]; };
                   wb_write.apply( channel, n_inst );
                end

                // Write the configuration registers
                wb_write.randomize with { my_addr == SPI_DIVIDER;
                                          my_data == divider; };
                wb_write.apply( channel, n_inst );


                wb_write.randomize with { my_addr == SPI_SS;
                                          my_data == ss; };
                wb_write.apply( channel, n_inst );

                // Write to the control register but don't set the GO bit yet
                wb_write.randomize with { my_addr == SPI_CTRL;
                                          my_data == ctrl; };
                wb_write.apply( channel, n_inst );

                // Set the GO bit to perform the SPI transfer
                ctrl |= GO_BIT;
                wb_write.randomize with { my_addr == SPI_CTRL;
                                          my_data == ctrl; };
                wb_write.apply( channel, n_inst );

        endtask

        `vmm_class_factory( wb_to_spi_scn )
endclass
```

With our Wishbone to SPI transfer scenario defined, we can now turn to writing our first test case. Tests in VMM are derived from the *vmm_test* class. The test case will configure our SPI testbench environment and tell the scenario generator the appropriate scenario(s) to run—in this case, the *wb_to_spi_scn* scenario. The testcase will be passed a reference to the instantiated environment in its constructor and configure it in the *start_of_sim_ph* phase before the scenario generator executes in the *run_ph* phase. Here is what a simple test would look like:

```
class wb_test1 extends vmm_test;
        wb_spi_env my_env;

        `vmm_typename( wb_test1 )

        function new( string name, wb_spi_env my_env );
                super.new(name);
                this.my_env = my_env;
        endfunction

        function void start_of_sim_ph;
           wb_to_spi_scn   scn = new();

           // Remove the atomic scenario
           my_env.wb_sub.scn_gen.scenario_set.delete();

           // Register test scenario
           my_env.wb_sub.scn_gen.register_scenario( "wb_to_spi", scn );

           // Specify number of scenarios to run
           vmm_opts::set_int("num_scenarios", 1, this);
        endfunction

        // Wait until the scenario generator is finished
        virtual task shutdown_ph();
```

```
            my_env.wb_sub.scn_gen.notify.wait_for(wb_spi_trans_scenario_gen::
      DONE);
        endtask

endclass : wb_test1
```

This testcase registers with the scenario generator the scenario it wants to run (*wb_to_spi*) and then specifies the number of times for the generator to execute the scenario. It also waits on the generator to make sure that it does not finish before all the stimulus has been generated.

A program top and simulation

While we have a harness/wrapper around the design and a top-level class-based environment, somewhere we need to instantiate everything and kick off the VMM machinery. For that, we will use a program block. The program block will instantiate the design wrapper, the class-based testbench, and testcase and start the simulation. It will also setup the virtual interface wrapper that will be used in our driver and monitors. Our program top looks like the following:

```
program wb_spi_top;

   import wb_spi_pkg::*;                        // Load in the testbench

   wb_spi_env           e;
   wb_test1             t;
   dut_if_wrapper       i;

   /////////////////////////////////////
   // OK, now run the test
   /////////////////////////////////////
   initial
   begin
        // Create the environment
        e = new( "wb_spi_env", "e" );

        // Setup the DUT interface wrapper
        i = new( "dut_if_wrapper", wb_spi_tb.dut_if );
        vmm_opts::set_object( "dut_intf", i );

        // Tests
        t = new( "wb_test1", e );
        vmm_simulation::run_tests();    // Kick off VMM
   end

endprogram : wb_spi_top
```

Notice, all of our testbench and test classes are imported from a package that we create by simply using *`include* for all of the source code files into a package. We store a hierarchical reference to our interface into the *dut_if_wrapper* and then use the configuration mechanism so the driver and monitors can grab a reference to it inside our testbench. To start the VMM simulation, we invoke *vmm_simulation::run_tests()*, which starts executing *wb_test1* since it is the only test instantiated. The *run_tests()* method can also be controlled on the command line by using the following options:

+vmm_test_file=<filename>	File list of test to run
+vmm_test=<test>	Test case to run
+vmm_test=<test>+<test>+...	List of test cases to run
+vmm_test=ALL_TESTS	Run all declared tests

Now we can run our VCS simulation …

```
  vcs -R –sverilog file1.sv file2.sv …
```

and see our environment building, generating stimulus, driving, monitor, and checking:

```
Chronologic VCS simulator copyright 1991-2009
Contains Synopsys proprietary information.
Compiler version D-2009.12; Runtime version D-2009.12;  Feb 11 03:09 2010

Normal[NOTE] on wb_spi_env(e) at                    0:
    Built wb_spi_env ...
Normal[NOTE] on wb_subenv(e:wb_sub) at                    0:
    Creating subenvironment ...
Normal[NOTE] on wb_monitor(e:wb_sub:wb_mon) at                    0:
    Building analysis port...
Normal[NOTE] on wb_spi_env(e) at                    0:
    Connect phase...
Normal[NOTE] on vmm_simulation(class) at                    0:
    Running Test Case wb_test1
class wb_spi_trans (0.0.0)
addr='h0
data='h4f92090d
kind=TX
Normal[NOTE] on wb_spi_env(e) at                    0:
    Reseting the DUT ...
Normal[NOTE] on wb_spi_env(e) at                24000:
    The DUT is now reset.
Normal[NOTE] on spi_monitor(e:spi_mon) at                24000:
    Monitoring the SPI bus ...
Normal[NOTE] on wb_driver(e:wb_sub:wb_drv) at                24000:
    Starting the WB driver ...
Normal[NOTE] on wb_driver(e:wb_sub:wb_drv) at                24000:
    Getting packet ...
Normal[NOTE] on wb_driver(e:wb_sub:wb_drv) at                24000:
    Transmitting packet ...
class wb_spi_trans (0.0.0)
addr='h0
data='h4f92090d
kind=TX
Normal[NOTE] on wb_driver(e:wb_sub:wb_drv) at                24000:
    wb_write task: addr = 00000000, data = 4f92090d
Normal[NOTE] on wb_monitor(e:wb_sub:wb_mon) at                24000:
    Monitoring the WB bus
class wb_spi_trans (0.0.0)
addr='h4
data='h48aa5e94
kind=TX
Normal[NOTE] on wb_monitor(e:wb_sub:wb_mon) at                26500:
    addr = 00, data = 4f92090d
Normal[NOTE] on wb_driver(e:wb_sub:wb_drv) at                26600:
    Getting packet ...
Normal[NOTE] on wb_driver(e:wb_sub:wb_drv) at                26600:
    Transmitting packet ...
.
.
.
Normal[NOTE] on spi_monitor(e:spi_mon) at             39503600:
    Sending transaction to scoreboard...
class mon_sb_trans (0.0.0)
addr='h0
data='h166928d9dca2f8a4865b
ctrl='h0
kind=RX
Normal[NOTE] on Data Stream Scoreboard(wb_spi_sb) at            39503600:
    Checking data...
Normal[NOTE] on wb_subenv(e:wb_sub) at             39503600:
    Scenario generation done...
Simulation PASSED on /./ (/./) at             39503600 (0 warnings, 0 demoted
errors & 0 demoted warnings)
Normal[NOTE] on vmm_simulation(class) at             39503600:
    Test Case wb_test1 Done
$finish at simulation time             39503600
           V C S   S i m u l a t i o n   R e p o r t
Time: 395036000 ps
CPU Time:      0.110 seconds;       Data structure size:   0.0Mb
```

And there you have it!—a simple environment demonstrating the major features of VMM.