VMM Primers

Version C-2009.06
June 2009

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of
>
> _____ and its employees. This is copy number_____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, HSIMplus, HSPICE-Link, iN-Tandem, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

List of the VMM Primers

The VMM primers provide introductions to the topics listed below.

- Creating a block-level environment that is reusable in a system-level verification environment:
  - VMM Primer: Composing Environments
- Configuring a memory allocation manager and using it to configure a design that requires memory buffers:
  - VMM Primer: Using the Memory Allocation Manager
- Writing VMM-compliant command-lever master transactors, also known as bus-functional models (BFM):
  - VMM Primer: Writing Command-Layer Master Transactors
- Writing VMM-compliant command-layer slave transactors:
  - VMM Primer: Writing Command-Layer Slave Transactors
- Writing VMM-compliant command-lever monitors:
  - VMM Primer: Writing Command-Layer Monitors
- Creating a scoreboard based on the Data Stream Scoreboard foundation classes and integrating it in a verification environment:
  - VMM Primer: Using the Data Stream Scoreboard
- Creating a RAL model of registers and memories, integrating it in a verification environment, and verifying the implementation of the registers and memories using the pre-defined tests:
  - VMM Primer: Using the Register Abstraction Layer

VMM Primer:
Composing Environments

Author:
Janick Bergeron

Version 1.2 / March 18, 2008

Introduction

The VMM Environment Composition application package is a set of methodology guidelines and support classes to help create system-level verification environments from block-level environments.

This primer is designed to learn how to create a block-level environment that will be reusable in a system-level verification environment. Other primers cover other aspects of developing VMM-compliant verification assets, such as transactors, verification environments and the Register Abstraction Layer package.

This primer assumes that you are familiar with VMM. If you need to ramp-up on VMM itself, you should read the other primers in this series.

The design used to illustrate how to use the VMM Environment Composition application package is the OpenCore Ethernet Media Access Controller (MAC). Despite being a fairly complex design, this primer focuses on a single aspect of its functionality: the verification of its transmission path. This design was chosen because it exhibits all of the challenges in reusing parts of its verification environment: It is configurable and it requires access to a shared resource. The system-level design will be a simple composition of two MAC blocks.

A complete verification environment that can verify the entire functionality of this design requires many more elements and additional capabilities.

This document is written in the same order you would create a verification environment with portions designed to be reused in a system-level verification environment. As such, you should read it in a sequential fashion. You can use the same sequence to create your own reusable sub-environments.

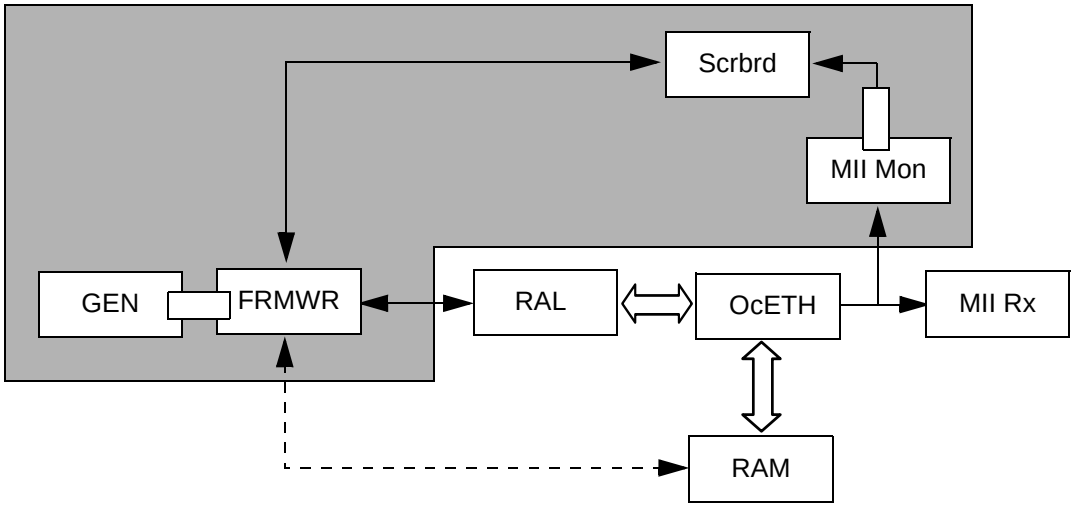
Step 1: Planning

Creating block-level environments that can be reused in a system-level environment does not happen by accident. They have to be structured and architected to make then reusable.

First, the entire block-level environment may not be reusable. Interfaces that are visible and/or controllable at the block level may no longer be available at the system level. Also, low-level random stimulus at the block level may need to carry well-formed higher-level information at the system-level. It is important to identify which portion(s) of the block-level environment that would be reusable in a system-level context.

Figure 1 on page 4 shows the structure of the verification environment surrounding the Ethernet MAC block. The firmware emulation transactor configures and controls the registers inside the design through a RAL model. The DMA memory, not being mapped in the DUT address space, is not included in the RAL model. The firmware emulation transactor accesses the DMA memory directly through backdoor read()/write() methods provided by the RAM model itself. The detailed operation of the firmware emulation layer to successfully transmit a frame is described in the Memory Allocation Manager Primer.

Figure 1 Block-Level Verification Environment



The shaded portion of the block-level environment will be made reusable in a system-level environment.

A passive monitor is used to extract information from the MII interface. The same information could have been extracted from the active MII receiver transactor. However, the MII interface may not be externally visible at the system-level and thus not in need of a receiver to properly terminate it. Using a monitor allows the self-checking portion of the block-level environment to be reused in an environment where that interface is internal.

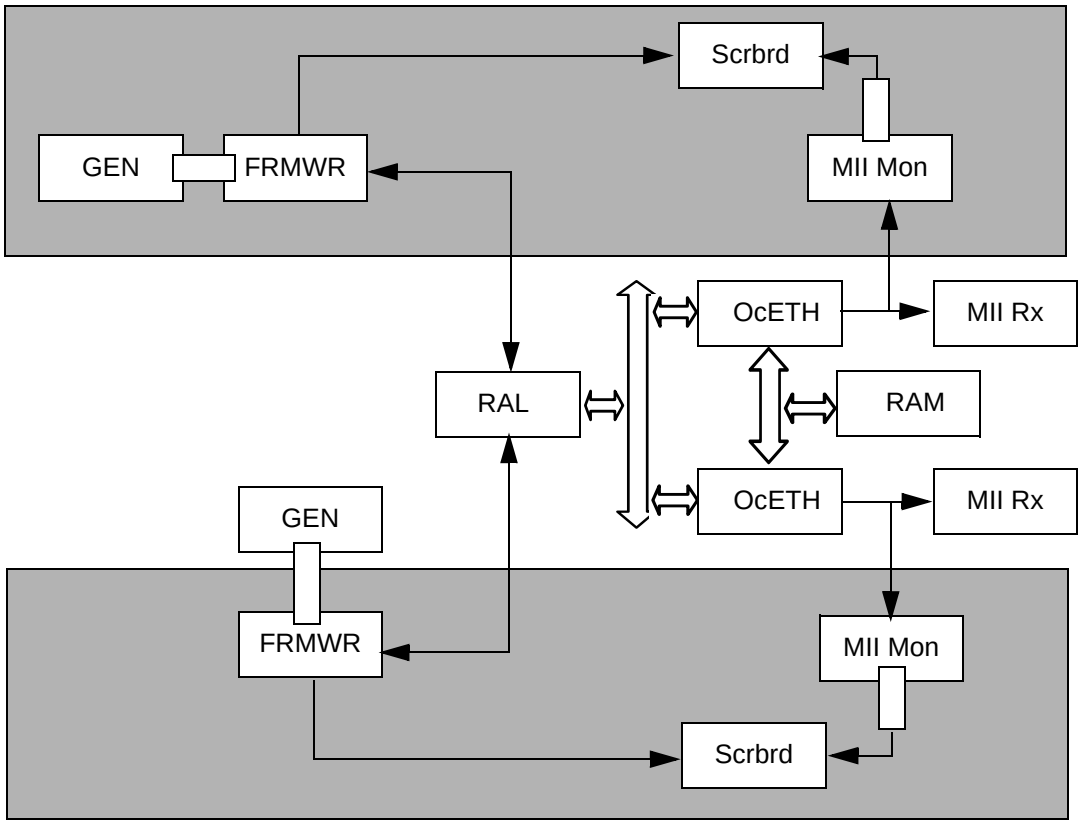
The generator that provides stimulus to the firmware emulation block will be made optional. This will enable the firmware layer to be fed from a high-level protocol stimulus source, should the need arise.

The Register Abstraction Layer is not included in the reusable portion because the address and means of accessing the registers will likely be different between the block-level environment and the system-level environment. It will be up to the system-level

environment to provide a reference to a suitable RAL model for the firmware emulation layer to be able to properly configure and control the corresponding block.

The system-level environment used in this primer is shown in Figure 2. The reusable sub-environment from Figure 1 is used twice with slightly different configurations.

Figure 2 System-Level Verification Environment



Step 2: Encapsulation

The portion of the block level environment that will be reusable, called a sub-environment, is encapsulated in a class extended from the `vmm_subenv` base class. The constructor arguments must include:

- An instance name.
- A sub-environment configuration descriptor.
- A `vmm_consensus` reference.
- All virtual interfaces and channels that cross the sub-environment boundary.

File: SubEnv/eth_subenv.sv

```
class oc_eth_subenv_cfg;
   ...
endclass: oc_eth_env_cfg

class oc_eth_env extends vmm_subenv;
   ...
   function new(string                  inst,
                oc_eth_env_cfg          cfg,
                vmm_consensus           end_test,
                virtual mii_if.passive mii_tx,
                eth_frame_channel       tx_chan,
                ...);
      super.new("OcEth SubEnv", inst, end_test);
      ...
   endfunction: new
   ...
endclass: oc_eth_subenv
```

The reference to the *tx_chan* argument is optional. If set to *null*, it will be assumed to not cross the sub-environment boundary and the generator will be internally allocated and connected. If it is not *null*, the reference will be used as-is and assumed to be properly fed from some source external to the sub-environment.

File: SubEnv/eth_subenv.sv

```
class oc_eth_subenv_cfg;
   ...
endclass: oc_eth_subenv_cfg

class oc_eth_subenv extends vmm_subenv;
   ...
   eth_frame_atomic_gen src;
   ...
   function new(string                  inst,
                oc_eth_env_cfg          cfg,
                vmm_consensus           end_test,
                virtual mii_if.passive mii_tx,
                eth_frame_channel       tx_chan,
                ...);
      super.new("OcEth SubEnv", inst, end_test);
      ...
      if (tx_chan == null) begin
          this.src = new("OcEth SubEnv Src", inst);
          tx_chan = this.src.out_chan;
          ...
      end
      ...
   endfunction: new
   ...
endclass: oc_eth_subenv
```

The sub-environment has additional logical interfaces required to properly interact with other components of the overall environment. These logical interfaces must also be provided as constructor arguments and include:

- A reference to a RAL model for the block. This will enable the firmware emulator to configure the block regardless of the physical interface or base address it will be located under.
- A reference to system memory for backdoor read/write. If the system memory were mapped into the space of the block, it could be accessed through the RAL model for that block. But such is not the case here.
- A reference to the memory allocation manager for the system memory. Since the system memory is a potentially shared resource, it my be managed by a shared resource manager.

File: SubEnv/eth_subenv.sv

```
class oc_eth_subenv extends vmm_subenv;
   ...
   eth_frame_atomic_gen src;
   ...
   function new(string                  inst,
                oc_eth_env_cfg          cfg,
                vmm_consensus           end_test,
                virtual mii_if.passive  mii_tx,
                eth_frame_channel       tx_chan,
                ral_block_oc_ethernet   ral,
                wb_ram                  dma_ram,
                vmm_mam                 dma_mam);
      super.new("OcEth SubEnv", inst, end_test);
      ...
      if (tx_chan == null) begin
          this.src = new("OcEth SubEnv Src", inst);
          tx_chan = this.src.out_chan;
          ...
      end
      ...
   endfunction: new
   ...
endclass: oc_eth_subenv
```

The subenvironment is then instantiated in the block-level environment and constructed, like any other transactor, in the `build()` step.

File: SubEnv/blk_env.sv

```
class test_cfg;
   rand oc_eth_subenv_cfg eth;
   ...
endclass: test_cfg
...
class blk_env extends vmm_ral_env;
   test_cfg              cfg;
   oc_eth_subenv         eth;
   ral_block_oc_ethernet ral_model;
   ...
   wb_ram                ram;
   wm_mam                dma_mam;
   ...
   function new();
      super.new();eth_sub
      this.cfg = new;
      this.ral_model = new();
      this.ral.set_model(this.ral_model);
      ...
   endfunction: new
   ...
   virtual function build();
      super.build();
      ...
      this.eth = new("Eth", this.cfg.eth, this.end_vote,
                     blk_top.mii.passive, null,
                     this.ral_model, this.ram,
                     this.dma_mam);
   endfunction: build
   ...
endclass: blk_env
```

Step 3: Configuration

The sub-environment configuration descriptor must contain all information necessary to configure the elements in the sub-environment.

File: SubEnv/eth_subenv.sv

```
class oc_eth_subenv_cfg;
   rand mii_cfg    mii;
   rand bit [47:0] dut_addr;
   rand bit [ 7:0] n_tx_bd;
   rand bit [15:0] max_frame_len;
   rand int unsigned run_for_n_frames;
   ...
endclass: oc_eth_subenv_cfg
```

Structural configuration is implemented in the constructor. It can be determined by the value of the virtual interfaces or channels. For example, the optional reference to a channel controls whether or not a generator is instantiated inside the sub-environment. This was implemented as part of Step 2. Structural configuration also includes appropriately configuring the transactors instantiated during construction.

File: SubEnv/eth_subenv.sv

```
class oc_eth_subenv_cfg;
   rand mii_cfg    mii;
   rand bit [47:0] dut_addr;
   rand bit [ 7:0] n_tx_bd;
   rand bit [15:0] max_frame_len;
   rand int unsigned run_for_n_frames;
   ...
endclass: oc_eth_subenv_cfg

class oc_eth_subenv extends vmm_subenv;
```

```
   ...
   eth_frame_atomic_gen src;
   mii_monitor          mon;
   frmwr_emulator       frmwr;

   function new(string                  inst,
                oc_eth_env_cfg          cfg,
                vmm_consensus           end_test,
                virtual mii_if.passive mii_tx,
                eth_frame_channel       tx_chan,
                ral_block_oc_ethernet   ral,
                wb_ram                  dma_ram,
                vmm_mam                 dma_mam);
      super.new("OcEth SubEnv", inst, end_test);
      ...
      if (tx_chan == null) begin
          this.src = new("OcEth SubEnv Src", inst);
          tx_chan = this.src.out_chan;

          this.src.stop_after_n_insts =
              this.cfg.run_for_n_frames;
          this.src.randomized_obj.src = this.cfg.dut_addr;
          this.src.randomized_obj.src.rand_mode(0);
      end
      ...
      this.mon = new(inst, 0, this.cfg.mii, this.mii_sigs);
      this.frmwr = new(inst, 0, this.cfg, tx_chan, this.sb,
         this.ral, this.dma_ram, this.dma_mam);
   endfunction: new
   ...
endclass: oc_eth_subenv
```

Functional configuration is implemented in the user-defined `vmm_subenv::configure()` method. This step configures the DUT to match the configuration of the sub-environment.

File: SubEnv/eth_subenv.sv

```
class oc_eth_subenv extends vmm_subenv;
   ...
   task configure();
      vmm_rw::status_e status;

      ral.HUGEN.set(0);
      ral.FULLD.set(1);
      ...
      ral.TXEN.write(status, 1);
      if (status !== vmm_rw::IS_OK) begin
         `vmm_error(ral.log, ...);
      end

      super.configured();
   endtask: configure
   ...
endclass: oc_eth_subenv
```

The call to `super.configured()` at the end of the `configure()` implements an error detection mechanism to ensure that a sub-environment is completely configured before it is started.

The `configure()` method of the sub-environment is then invoked from at the `cfg_dut()` step.

File: SubEnv/blk_env.sv

```
class blk_env extends vmm_ral_env;
   ...
   virtual task cfg_dut();
      super.cfg_dut();
      this.eth.configure();
   endtask: cfg_dut
   ...
endclass: blk_env
```

The configuration of the Memory Allocation Manager instance that is responsible for managing the system memory must match the configuration of that memory. This can be implemented by constraining the configuration of one to match the configuration of the other.

File: SubEnv/blk_env.sv

```
class test_cfg;
   ...
   rand wb_slave_cfg  ram;
   rand vmm_mam_cfg   mam;

   constraint test_cfg_valid {
      ram.port_size    == wb_cfg::DWORD;
      ram.min_addr     == 32'h0000_0000;
      ram.max_addr     == 32'hFFFF_FFFF;
      ...
      mam.n_bytes      == 4;
      mam.start_offset == ram.min_addr;
      mam.end_offset   == ram.max_addr;
   }
   ...
endclass: test_cfg
```

Step 4: Execution Sequence

The sub-environment must then be integrated in the block-level environment's execution sequence. First, the `vmm_subenv::start()` method must be extended to start all transactors and ancillary threads in the sub-environment.

File: SubEnv/eth_subenv.sv

```
class oc_eth_subenv extends vmm_subenv;
   ...
```

```
   virtual task start();
      super.start();

      if (this.cfg.run_for_n_frames > 0 && this.src != null)
      begin
         this.src.start_xactor();
      end
      this.frmwr.start_xactor();

   fork
      forever begin
         eth_frame fr;
         this.mon.to_phy_chan.get(fr);
         this.sb.received_by_phy_side(fr);
      end
   join_none
   ...
   endtask: start
   ...
endclass: oc_eth_subenv
```

The `start()` method of the sub-environment must then be called in the extension of the `vmm_env::start()` method.

File: SubEnv/blk_env.sv

```
class blk_env extends vmm_ral_env;
   ...
   virtual task start();
      super.start();
      ...
      this.eth.start();
      ...
   endtask: start
   ...
endclass: blk_env
```

The `vmm_subenv::stop()` and `vmm_subenv::cleanup()` methods must be similarly implemented then invoked from the `vmm_env::stop()` and `vmm_env::cleanup()` method extensions respectively.

File: SubEnv/eth_subenv.sv

```
class oc_eth_subenv extends vmm_subenv;
   ...
   virtual task stop();
      super.stop();
      if (this.src != null) this.src.stop_xactor();
   endtask: stop

   virtual task cleanup();
      super.cleanup();

      if (this.cfg.run_for_n_frames > 0) begin
         `vmm_error(this.log, $psprintf("%0d frames were
            not seen by the scoreboard",
            this.cfg.run_for_n_frames));
      end
   endtask: cleanup
endclass: oc_eth_subenv
```

File: SubEnv/blk_env.sv

```
class blk_env extends vmm_ral_env;
   ...
   virtual task stop();
      super.stop();
      this.eth.stop();
   endtask: stop

   virtual task cleanup();
      super.cleanup();
      this.eth.cleanup();
   endtask: cleanup
endclass: blk_env
```

The most important—and difficult to implement—is the `wait_for_end` step. Fortunately, the VMM Environment Composition package includes a new object that helps make the decision whether to end the test or not: `vmm_consensus`. An instance of that object already exists in the `vmm_env::end_vote` class property and it has already been passed into the `vmm_subenv::end_test` property via the constructor of the sub-environment (See Step 2).

The decision to end the test is now centralized in that one object but the contributors to that decision can be distributed over the entire verification environment. The sum of all contributions will be used to determine whether to end the test or not, regardless of how many contributors there are.

From the sub-environment's perspective, the test may end once the generator is done, all channels are empty and all transactors are idle. There is no wait-for-end method in the `subenv` base class as the end-of-test decision is implemented through the `vmm_consensus` instance. The various contributions to the end-of-test consensus are registered in the extension of the `vmm_subenv::start()` method.

File: SubEnv/eth_subenv.sv

```
class oc_eth_subenv extends vmm_subenv;
   ...
   virtual task start();
     super.start();

     if (this.cfg.run_for_n_frames > 0 && this.src != null)
        begin
           this.start_xactor();
        end
     this.frmwr.start_xactor();

     fork
```

```
      forever begin
         eth_frame fr;
         this.mon.to_phy_chan.get(fr);
         this.sb.received_by_phy_side(fr);
      end
   join_none

   if (this.src != null) begin
      this.end_test.register_notification(
         this.src.notify,
         eth_frame_atomic_gen::DONE);
   end
   this.end_test.register_channel(this.tx_chan);
   this.end_test.register_channel(this.mon.to_phy_chan);
   this.end_test.register_xactor(this.frmwr);
   this.end_test.register_xactor(this.mon);
endtask: start
...
endclass: oc_eth_subenv
```

From the block-level environment's perspective, the test may end when the sub-environment consents and all transactors and channels instantiated in the block-level environment itself are respectively idle and empty. The consent of the sub-environment is automatically taken care of by virtue of using the same `vmm_consensus` instance. The other contributions to the end-of-test decision are registered in the extension of the `vmm_env::start()` method.

File: SubEnv/blk_env.sv

```
class blk_env extends vmm_ral_env;
   ...
   virtual task start();
      super.start();
      ...
      this.eth.start();
      this.end_vote.register_channel(this.host.exec_chan);
      this.end_vote.register_xactor(this.phy);
```

```
      this.end_vote.register_xactor(this.host);
   endtask: start
   ...
endclass: blk_env
```

The implementation of the `wait_for_end` step is now only a matter of waiting for the end-of-test consensus to be reached.

File: SubEnv/blk_env.sv

```
class blk_env extends vmm_ral_env;
   ...
   virtual task wait_for_end();
      super.wait_for_end();
      this.end_vote.wait_for_consensus();
   endtask: wait_for_end
   ...
endclass: blk_env
```

Step 5: Block-Level Tests

The block level environment now completely integrates the sub-environment and is ready to perform block-level tests. For example, a trivial block-level test would limit the number of transmitted frames to perform initial debug.

File: SubEnv/blk_trivial_test.sv

```
program test;
   blk_env env = new;

   initial
   begin
      env.gen_cfg();
      env.cfg.eth.run_for_n_frames = 3;
      env.run();
   end
```

```
endprogram: test
```

The test may be run using the following command:

Command:

```
% make blk_tx
```

Step 6: System-Level Environment

Constructing the system-level environment using two instances of the sub-environment is very similar to constructing the block-level environment.

First, two configuration descriptors are required instead of one.

File: SubEnv/sys_env.sv

```
class test_cfg;
   rand oc_eth_subenv_cfg eth[2];
   ...
endclass: test_cfg
```

Two instances of the sub-environments are required, each connected to their respective physical interfaces. A single RAL model, RAM and Memory Allocation Manager is shared across both sub-environment because there are shared resources.

File: SubEnv/sys_env.sv

```
class sys_env extends vmm_ral_env;
   ...
   oc_eth_subenv eth[2];
   ...
   virtual function build();
      ...
```

```
      this.eth[0] = new("Eth0", this.cfg.eth[0],
         this.end_vote,
         sys_top.mii_0.passive, null,
         this.ral_model, this.ram, this.dma_mam);

      this.eth[1] = new("Eth0", this.cfg.eth[1],
         this.end_vote,
         sys_top.mii_1.passive, this.src.out_chan,
         this.ral_model, this.ram, this.dma_mam);
      ...
   endfunction: build
   ...
endclass: sys_env
```

Notice how the end-of-test `vmm_consensus` object is passed to both sub-environments. This allows both sub-environments to independently contribute to the end-of-test decision.

Also, notice how the `tx_chan` value is not *null* for the second instance of the sub-environment. This configures the sub-environment to use the external source provided—in this case an externally instantiated atomic generator.

Both sub-environments must be configured. It is a good idea to fork the configuration steps to perform as much of it as possible concurrently.

File: SubEnv/sys_env.sv

```
class sys_env extends vmm_ral_env;
   ...
   virtual task cfg_dut();
      super.cfg_dut();
      fork
         this.eth[0].configure();
         this.eth[1].configure();
      join
   endtask: cfg_dut
   ...
```

```
endclass: sys_env
```

The `start()`, `stop()` and `cleanup()` methods must similarly invoke their respective counterpart in both sub-environment instances.

File: SubEnv/sys_env.sv

```
class sys_env extends vmm_ral_env;
   ...
   virtual task stop();
      super.stop();
         this.eth[0].stop();
         this.eth[1].stop();
   endtask: stop
   ...
endclass: sys_env
```

However, the `wait_for_end()` task remains identical as the `vmm_consensus` object takes care of scaling the end-of-test decision, regardless of the number of contributors.

File: SubEnv/sys_env.sv

```
class sys_env extends vmm_ral_env;
   ...
   virtual task wait_for_end();
      super.wait_for_end();
      this.end_vote.wait_for_consensus();
   endtask: wait_for_end
   ...
endclass: sys_env
```

Step 7: System-Level Tests

The system-level environment now completely integrates two sub-environments and is ready to perform system-level tests. For example, a trivial system-level test would limit the number of transmitted frames to perform initial debug.

File: SubEnv/sys_trivial_test.sv

```
program test;
   sys_env env = new;

   initial
   begin
      env.gen_cfg();
      env.cfg.eth[0].run_for_n_frames = 3;
      env.cfg.eth[1].run_for_n_frames = 3;
      env.run();
   end
endprogram: test
```

The test may be run using the following command:

Command:

```
% make sys_tx
```

Step 8: Congratulations

You have now completed the development and integration of a reusable sub-environment. You can now speed-up the development of system-level tests by leveraging the work done in block-level environments.

You may consider reading other publications in this series to learn how to write VMM-compliant command-layer transactors, verification environments, integrate a Register Abstraction Layer model or a Data Stream Scoreboard, or use the Memory Allocation Manager.

VMM Primer:
Using the Memory Allocation Manager

Author:
Janick Bergeron

Version 1.1 / May 28, 2006

Introduction

The VMM Memory Allocation Manager is an element of the VMM Environment Composition application package that can be used to manage the dynamic allocation of memory regions in a large RAM. It is similar to C's **`malloc()`** and **`free()`** routines. It is useful for managing memory buffers required by designs to store temporary, such as DMA buffers. These memory buffers are usually configured as a buffer address value (pointer) through a descriptor or a linked list in the design.

This primer is designed to learn how to configure a memory allocation manager, and how to use it in a verification environment to properly configure a design that requires memory buffers. Other primers cover other aspects of developing VMM-compliant verification assets, such as transactors, verification environments and the Register Abstraction Layer package.

This primer assumes that you are familiar with VMM. If you need to ramp-up on VMM itself, you should read the other primers in this series.

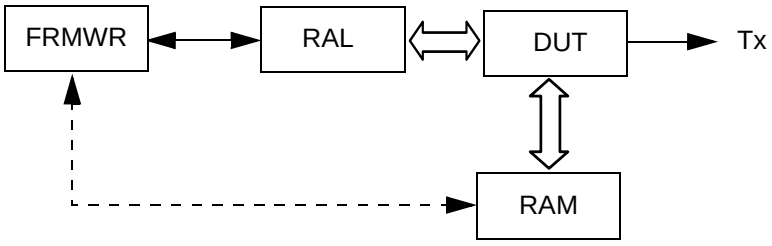
The design used to illustrate how to use the VMM Memory Allocation Manager is the OpenCore Ethernet Media Access Controller (MAC). Despite being a fairly complex design, this primer focuses on a single aspect of its functionality: the transmit DMA buffers. A complete verification environment that can verify this design requires many more elements than a memory allocation manager. This primer will only show the portions of the environment that make use of the VMM Memory Allocation Manager class.

This document is written in the same order you would incorporate a memory allocation manager in a verification environment and verify your design. As such, you should read it in a sequential fashion. You can use the same sequence to use a memory allocation manager and use it to verify your design.

The DUT

Figure 1 shows the (partial) structure of the verification environment surrounding the Ethernet MAC. The firmware emulation transactor configures and controls the registers inside the design through a RAL model. The DMA memory, not being mapped in the DUT address space, is not included in the RAL model. The firmware emulation transactor accesses the DMA memory directly through backdoor **`read()`**/**`write()`** methods provided by the RAM model itself.

Figure 1 A Partial Verification Environment



The firmware must first initialize the DUT as follows:

1. All Transmit Buffer Descriptors are marked as “not ready”.
2. Transmission is enabled.

To transmit a frame, the firmware must use the following procedure:

1. The address of the frame in the RAM is specified in the Transmit Buffer Descriptor.
2. Store the frame in consecutive locations in the RAM.
3. The number of bytes in the frame is specified in the Transmit Buffer Descriptor.
4. The Transmit Buffer Descriptor is marked as “ready”.
5. Wait for a “Buffer Transmitted” interrupt. A Transmit Buffer will now be available and automatically marked as “not ready” by the DUT.

The DUT contains a programmable number of Transmit Buffer Descriptors (TxBD), up to 128, as defined by the TX_BD_NUM register. Each TxBD is located in consecutive memory addresses and has the structure partially by the following RALF specification.

File: SubEnv/oc_ethernet.ralf

```
block oc_ethernet {
   bytes 4;
   ...
   register TX_BD_NUM {
      field TX_BD_NUM { bits 8; }
   }
   ...
   regfile TxBD[128] @... +2 {
      register TxBD_CTRL {
         bits 32;
         ...
         field WR;
         ...
         field RD;
         field LEN { bits 16; }
      }
      register TxPNT {
         bits 32;
         field PTR { bits 32; }
```

```
        }
    }
    . . .
}
```

There are two ways DMA buffers can be allocated: statically or dynamically.

With static buffer allocation, buffers are allocated when the TxBD are initialized. They are large enough to accommodate the largest possible frames to be transmitted and are reused to transmit different frames.

With dynamic buffer allocation, a new buffer is allocated for every frame to be transmitted. Buffers are sized according to the need of each frame. Each time, the TxBD is updated to refer to the newly allocated buffer.

An actual device driver would probably use a static buffer allocation approach. This approach minimizes the buffer allocation problem and restricts the memory usage to a well-defined area. But this approach is unlikely to expose several categories of functional bugs in the processing of transmit buffers. It would also require several different runs of randomly-allocated static buffers to cover different areas of the addressable space. Therefore, to exercise more of the addressable space and maximize the number of different buffer locations and sizes, a dynamic buffer allocation strategy will be used.

Step 1: Configuration

A Memory Allocation Manager will be used to manage the allocation of DMA buffers in the RAM. As such, it must be configured coherently with the configuration of the RAM itself.

The environment configuration descriptor contains an instance of the RAM model configuration descriptor. It must also contain an instance of the MAM configuration descriptor. Both are randomized and constrained to be coherent.

File: SubEnv/VIPs/wishbone/config.sv

```
class wb_cfg;
   typedef enum {BYTE, WORD, DWORD, QWORD} sizes_e;
   rand sizes_e port_size;
   ...
endclass: wb_cfg

class wb_slave_cfg extends wb_cfg;
   rand bit [63:0] min_addr;
   rand bit [63:0] max_addr;
   ...
endclass: wb_slave_cfg
```

File: SubEnv/blk_env.sv

```
class test_cfg;
   ...
   rand wb_slave_cfg ram;
   rand vmm_mam_cfg  mam;
   ...
   constraint test_cfg_valid {
      ...
      mam.n_bytes      == 1;
      mam.start_offset == ram.min_addr;
      mam.end_offset   == ram.max_addr;
      ...
   }
   ...
endclass: test_cfg
```

When the test configuration descriptor is randomized in the extension of the **`vmm_env::gen_cfg()`** method, the configuration of the memory allocation manager will match the configuration of the memory it manages: they will both have the same number of bytes per address and the same address range.

File: SubEnv/blk_env.sv:

```
virtual function void gen_cfg();
   bit ok;
      super.gen_cfg();
      ok = this.cfg.randomize();
   ...
endfunction: gen_cfg
```

Note that the memory allocation mode and locality remain unconstrained. This will hopefully uncover bugs when DMA buffers are reused, located in adjacent locations or in widely different addresses.

Step 2: Buffer Initialization

The TxBD must be initialized to the “not ready” state before transmission is enabled. This is accomplished in the extension of the **`vmm_env::cfg_dut()`** step. Once the buffer descriptors have been initialized, transmission can be enabled.

File: SubEnv/eth_subenv.sv

```
virtual task configure();
   ...
   ral.TX_BD_NUM.set(cfg.n_tx_bd);
   ...
   begin
      int bd_addr = 0;
```

```
      ...
      repeat (cfg.n_tx_bd) begin
         ...
         ral.TxBD[bd_addr].RD.write(status, 0);
         ...
         bd_addr++;
      end
      ...
   end
   ...
   ral.TXEN.write(status, 1);
   ...
endtask: configure
```

The DUT is now ready to transmit frames. It is waiting for the first TxBD to be specified as “ready”.

Step 3: Frame Transmission

The firmware emulation transactor is responsible for initiating the transmission of frames it receives from the frame generator. When a frame and a transmit buffer are available, a DMA buffer of the appropriate length needs to be allocated and the frame data transferred to the memory corresponding to the newly allocated buffer.

File: SubEnv/eth_subenv.sv

```
class frmwr_emulator extends vmm_xactor;
  ...
  local task tx_driver();
    logic [7:0] bytes[];
    forever begin
      eth_frame fr;
      ...
      if (!drop) begin
```

```
        vmm_mam_region bfr;
        ...
        bfr = this.dma_mam.request_region(fr.byte_size());
        ...
        len = fr.byte_size();
        bytes = new [len + (4 - len % 4)];
        fr.byte_pack(bytes);
        for (int i = 0; i < len; i += 4) begin
           this.ram.write(bfr.get_start_offset() + i,
            {bytes[i], bytes[i+1],
            bytes[i+2], bytes[i+3]});
        end
        ...
      end
    end
  endtask: tx_driver
  ...
endclass: frmwr_emulator
```

Of course, once the frame is laid in the DMA buffer, all that remains is to update the buffer descriptor to point to the new DMA buffer and mark it as “ready”.

File: SubEnv/eth_subenv.sv

```
class frmwr_emulator extends vmm_xactor;
  ...
  local task tx_driver();
    logic [7:0] bytes[];
    forever begin
      eth_frame fr;
      ...
      if (!drop) begin
        vmm_mam_region bfr;
        ...
        bfr = this.dma_mam.request_region(fr.byte_size());
        ...
        len = fr.byte_size();
        bytes = new [len + (4 - len % 4)];
        fr.byte_pack(bytes);
        for (int i = 0; i < len; i += 4) begin
```

```
            this.ram.write(bfr.get_start_offset() + i,
              {bytes[i], bytes[i+1], bytes[i+2], bytes[i+3]});
          end
          ...
          ral.TxBD[bd_addr].PTR.write(status,
            bfr.get_start_offset());
          ...
          cfg.dma_bfrs[bd_addr] = bfr;
          ral.TxBD[bd_addr].LEN.set(len);
          ral.TxBD[bd_addr].RD.write(status, 1);
          ...
        end
      end
    endtask: tx_driver
    ...
endclass: frmwr_emulator
```

Step 4: Frame Completion

To avoid memory leakage, and allow a greedy memory allocation mode to allocate previously-used memory, it is necessary to free buffers once the frame they contained has been transmitted.

This is implemented in the interrupt service routine. When an interrupt signals that a frame has been transmitted, all of the Transmit Buffer Descriptors that are newly marked as “not ready” are assumed to have been transmitted. The memory allocated for this DMA buffer is then released.

File: SubEnv/eth_subenv.sv

```
class frmwr_emulator extends vmm_xactor;
   ...
   local task service_irq();
   ...
   forever begin
      ...
```

```
      if (TXE || TXB) begin
         ...
         while (...) begin
            ...
            ral.TxBD[bd_addr].RD.read(status, RD);
            ...
            if (RD) break;
               ...
            this.dma_mam.release_region(
               this.cfg.dma_bfrs[bd_addr]);
         end
      end
      ...
   end
   endtask: service_irq
   ...
endclass: frmwr_emulator
```

Step 5: Congratulations

You have now completed the integration of a VMM Memory Allocation Manager in a verification environment. You can now exercise more functionality of the design and have the opportunity to uncover more functional bugs through the random allocation of DMA buffers..

You may consider reading other publications in this series to learn how to write VMM-compliant command-layer transactors, verification environments or integrate a Register Abstraction Layer model or a Data Stream Scoreboard

VMM Primer: Writing Command-Layer Master Transactors

Author:
Janick Bergeron

Version 1.3 / December 1, 2006

Introduction

The *Verification Methodology Manual for SystemVerilog* book was never written as a training book. It was designed to be a reference document that defines what is—and is not—compliant with the methodology described within it.

This primer is designed to learn how to write VMM-compliant command-layer master transactors—transactors with a transaction-level interface on one side and pin wiggling on the other, also known as bus-functional models (BFM). Other primers will eventually cover other aspects of developing VMM-compliant verification assets, such as slave transactors, monitors, functional-layer transactors, generators, assertions and verification environments.

The protocol used in this primer was selected for its simplicity. Because of its simplicity, it does not require the use of many elements of the VMM standard library. It is sufficient to achieve the goal of demonstrating, step by step, how to create a simple VMM-compliant master transactor.

This document is written in the same order you would implement a command-layer transactor. As such, you should read it in a sequential fashion. You can use the same sequence to create your own specific transactor.

A word of caution however: it may be tempting to stop reading this primer half way through, as soon as a functional transactor is available. VMM compliance is a matter of degree. Once a certain minimum level of functionality is met, a transactor may be declared VMM compliant. But additional VMM functionality—such as

callbacks—will make it much easier to use in different verification environments. Therefore, you should read—and apply—this primer in its entirety.

This primer will show how to apply the various VMM guidelines, not attempt to justify them or present alternatives. Throughout the primer, references are made to rule numbers and table numbers: Those are rules and tables in the *Verification Methodology Manual for SystemVerilog*. If you are interested in learning more about the justifications of various techniques and approaches used in this primer, you should refer to the VMM book under the relevant quoted rules and recommendations.

The Protocol

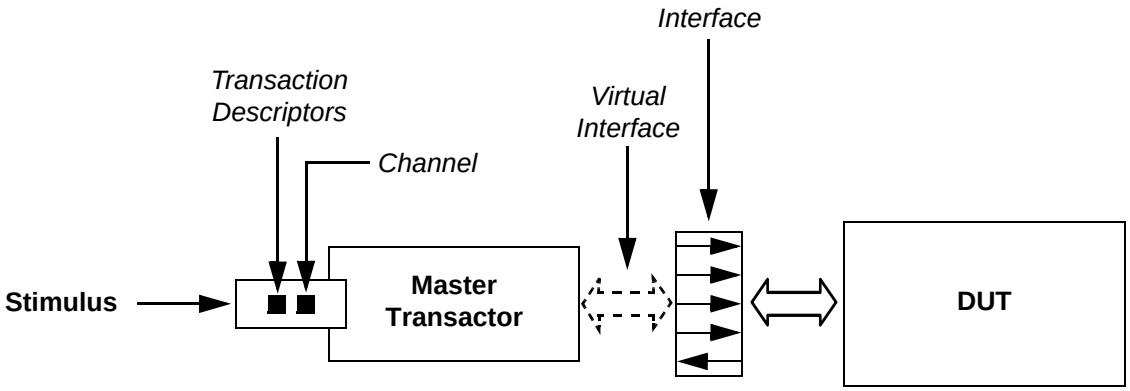
The protocol used in this primer is the AMBA™ Peripheral Bus (APB) protocol. It is a simple single-master address-based parallel bus providing atomic individual read and write cycles. The protocol specification can be found in the AMBA™ Specification (Rev 2.0) available from ARM (http://arm.com).

When writing a reusable transactor, you have to think about all possible applications it may be used in, not just the device you are using it for the first time. Therefore, even though the device in this primer only supports 8 address bits and 16 data bits, the APB transactors should be written for the entire 32-bit of address and data information.

The Verification Components

Figure 1 illustrates the various components that will be created throughout this primer. A command-layer master transactor interfaces directly to the DUT signals and initiates transactions upon requests on a transaction-level interface.

Figure 1 Components Used in this Primer



Step 1: The Interface

The first step is to define the physical signals used by the protocol to exchange information between a master and a slave. A single exchange of information (a READ or a WRITE operation) is called a *transaction*. There may be multiple slaves on an APB bus but there can only be one master. Slaves are differentiated by responding to different address ranges.

The signals are declared inside an **`interface`** (Rule 4-4). The name of the interface is prefixed with "apb_" to identify that it belongs to the APB protocol (Rule 4-5). The entire content of the file declaring

the interface is embedded in an **`ifndef/`define/`endif** construct. This is an old C trick that allows the file to be included multiple times, whenever required, without causing multiple-definition errors.

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if;
   ...
endinterface: apb_if

`endif
```

The signals, listed in the AMBA™ Specification in Section 2.4, are declared as **wires** (Rule 4-6) inside the **interface**.

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if(input bit pclk);
   wire [31:0] paddr;
   wire        psel;
   wire        penable;
   wire        pwrite;
   wire [31:0] prdata;
   wire [31:0] pwdata;
   ...
endinterface: apb_if

`endif
```

Because this is a synchronous protocol, **clocking** blocks are used to define the direction and sampling of the signals (Rule 4-7, 4-11).

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if(input bit pclk);
   wire [31:0] paddr;
   wire        psel;
   wire        penable;
   wire        pwrite;
   wire [31:0] prdata;
   wire [31:0] pwdata;

   clocking mck @(posedge pclk);
      output paddr, psel, penable, pwrite, pwdata;
      input  prdata;
   endclocking: mck
   ...
endinterface: apb_if

`endif
```

The **`clocking`** block defining the synchronous signals is specified in the **`modport`** for the APB master transactor (Rule 4-9, 4-11, 4-12). The clock signal need not be specified as it is implicit in the clocking block.

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if(input bit pclk);
   wire [31:0] paddr;
   wire        psel;
   wire        penable;
   wire        pwrite;
   wire [31:0] prdata;
   wire [31:0] pwdata;

   clocking mck @(posedge pclk);
```

```
      output paddr, psel, penable, pwrite, pwdata;
      input  prdata;
   endclocking: mck

   modport master(clocking mck);

endinterface: apb_if

`endif
```

The **interface** declaration is now sufficient for writing a master APB transactor. To be fully compliant, it should eventually include a **modport** for a slave and a passive **monitor** transactor (Rule 4-9). These can be added later, when these transactors will be written.

Step 2: Connecting to the DUT

The **interface** may now be connected to the DUT. It is instantiated in a top-level **module**, alongside of the DUT instantiation (Rule 4-13). The connection to the DUT pins are specified using a hierarchical reference to the wires in the **interface** instance.

File: Command_Master_Xactor/tb_top.sv

```
module tb_top;
   ...
   apb_if apb0(...);
   ...
   slave_ip dut_slv(...,
                    .apb_addr   (apb0.paddr[7:0]  ),
                    .apb_sel    (apb0.psel        ),
                    .apb_enable (apb0.penable     ),
                    .apb_write  (apb0.pwrite      ),
                    .apb_rdata  (apb0.prdata[15:0]),
                    .apb_wdata  (apb0.pwdata[15:0]),
                    ...);
```

```
endmodule: tb_top
```

This top-level **module** also contains the clock generators (Rule 4-15), using the **bit** type (Rule 4-17) and ensuring that no clock edges will occur at time zero (Rule 4-16).

File: Command_Master_Xactor/tb_top.sv

```
module tb_top;
   bit clk = 0;
   apb_if apb0(clk);
   ...

   my_design dut(...,
                 .apb_addr   apb0.paddr[7:0]),
                 .apb_sel    (apb0.psel),
                 .apb_enable (apb0.penable),
                 .apb_write  (apb0.pwrite),
                 .apb_rdata  (apb0.prdata[15:0]),
                 .apb_wdata  (apb0.pwdata[15:0]),
                 ...
                 .clk        (clk));

   always #10 clk = ~clk;
endmodule: tb_top
```

Step 3: The Transaction Descriptor

The next step is to define the APB transaction descriptor. Traditionally, tasks would have been defined, one for the READ transaction and one for the WRITE transaction.

File: apb/apb_master.sv

```
task read(input  bit    [31:0] addr,
          output logic [31:0] data);
```

```
task write(input bit [31:0] addr,
           input bit [31:0] data);
```

This works well for directed tests, but not at all for random tests. A random test requires a transaction descriptor (Rule 4-54). This descriptor is a **`class`** (Rule 4-53) extended from the **`vmm_data`** class (Rule 4-55), containing a public **`rand`** property enumerating the directed tasks (Rule 4-60, 4-62) and public **`rand`** properties for each **`task`** argument (Rule 4-59, 4-62). If an argument is the same across multiple tasks, a single property can be used. It also needs a **`static vmm_log`** property instance used to issue messages from the transaction descriptor. This instance of the message service interface is passed to the **`vmm_data`** constructor (Rule 4-58).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   rand enum {READ, WRITE} kind;
   rand bit   [31:0] addr;
   rand logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new
   ...
endclass: apb_rw
...
`endif
```

Note how the same property is used for `data`. It will be interpreted differently depending on the transaction kind (Rule 4-71). In a WRITE transaction, it is interpreted as the data to be written. In a READ transaction, the random content is initially ignored and it will be replaced by the data value that was read. The type for the `data` property is **`logic`** as it is a superset of the **`bit`** type and will allow the description of READ cycles to reflect unknown results.

Although the transaction descriptor is not yet VMM-compliant, it has the minimum functionality to be used by a transactor. A transaction-level interface will be required to transfer transaction descriptors to a transactor to be executed. This is done using the **`vmm_channel** macro (Rule 4-56).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   rand enum {READ, WRITE} kind;
   rand bit   [31:0] addr;
   rand logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new
   ...
endclass: apb_rw

`vmm_channel(apb_rw)
...
`endif
```

Step 4: The Master Transactor

The master transactor can now be started. It is a class (Rule 4-91) derived from the **`vmm_xactor`** base class (4-92).

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV
...
class apb_master extends vmm_xactor;
   ...
endclass: apb_master

`endif
```

The task implementing the READ and WRITE cycles are implemented in this class. They are declared **`virtual`** so the transactor may be extended to modify the behavior of these tasks if so required. They are also declared **`protected`** to prevent them from being called from outside the class and create concurrent bus access problems.

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_master extends vmm_xactor;
   ...
   virtual protected task read(input  bit   [31:0] addr,
                               output logic [31:0] data);
      ...
   endtask: read

   virtual protected task write(input bit [31:0] addr,
                                input bit [31:0] data);
```

```
      ...
   endtask: write
   ...
endclass: apb_master

`endif
```

The transactor needs a transaction-level interface to receive transactions to be executed and a physical-level interface to wiggle pins. The former is done using a **`vmm_channel`** instance and the latter is done using a **`virtual modport`**. Both are passed to the transactor as constructor arguments (Rule 4-108, 4-113) and both are saved in public properties (Rule 4-109, 4-112).

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_master extends vmm_xactor;
   apb_rw_channel in_chan;
   virtual apb_if.master sigs;

   function new(string name,
                int unsigned stream_id,
                virtual apb_if.master sigs,
                apb_rw_channel in_chan = null);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      if (in_chan == null)
         in_chan = new("APB Master Input Channel", name);
      this.in_chan = in_chan;
   ..
   endfunction: new
   ...
   virtual protected task read(input  bit     [31:0] addr,
                               output logic [31:0] data);
```

```
      ...
   endtask: read

   virtual protected task write(input bit [31:0] addr,
                                input bit [31:0] data);
      ...
   endtask: write
   ...
endclass: apb_master

`endif
```

When the transactor is reset, the input channel must be flushed and the critical output signals must be driven to their idle state. This is accomplished in the extension of the **`vmm_xactor::reset_xactor()`** method (Table A-8). This method may be called in the transactor constructor to initialize the output signals to their idle state, or explicit signal assignments may be used in the constructor.

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_master extends vmm_xactor;
   apb_rw_channel in_chan;
   virtual apb_if.master sigs;

   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.master sigs,
                apb_rw_channel        in_chan = null);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      if (in_chan == null)
         in_chan = new("APB Master Input Channel", name);
```

```
      this.in_chan = in_chan;
      this.sigs.mck.psel    <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: new

   virtual function void reset_xactor(
         reset_e rst_typ = SOFT_RST);
      super.reset(rst_typ);
      this.in_chan.flush();
      this.sigs.mck.psel <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: reset_xactor
   ...
   virtual protected task read(input bit    [31:0] addr,
                               output logic [31:0] data);
      ...
   endtask: read

   virtual protected task write(input bit [31:0] addr,
                                input bit [31:0] data);
      ...
   endtask: write
   ...
endclass: apb_master

`endif
```

The transaction descriptors are pulled from the input channel and translated into method calls in the `main()` task (Rule 4-93). The most flexible transaction execution mechanism uses the active slot as it supports block, nonblocking and out-of-order execution models (Rules 4-121, 4-122, 4-123, 4-124 and 4-129). Because the protocol supports being suspended between transactions, the **`vmm_xactor::wait_if_stopped_or_empty()`** method is used to suspend the execution of the transactor if it is stopped.

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
```

```
`define APB_MASTER__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_master extends vmm_xactor;
   apb_rw_channel        in_chan;
   virtual apb_if.master sigs;

   function new(string                 name,
                int unsigned           stream_id,
                virtual apb_if.master  sigs,
                apb_rw_channel         in_chan = null);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      if (in_chan == null)
         in_chan = new("APB Master Input Channel", name);
      this.in_chan = in chan;
      this.sigs.mck.psel    <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: new

   virtual function void reset_xactor(
         reset_e rst_typ = SOFT_RST);
      super.reset(rst_typ);
      this.in_chan.flush();
      this.sigs.mck.psel <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      @ (this.sigs.mck);
      forever begin
         apb_rw tr;
         this.wait_if_stopped_or_empty(this.in_chan);
         this.in_chan.activate(tr);
         ...
         this.in_chan.start();
         case (tr.kind)
            apb_rw::READ : this.read(tr.addr, tr.data);
            apb_rw::WRITE: this.write(tr.addr, tr.data);
```

```
            endcase
            this.in_chan.complete();
            ...
            this.in_chan.remove();
         end
      endtask: main

      virtual protected task read(input bit    [31:0] addr,
                                  output logic [31:0] data);
      ...
      endtask: read

      virtual protected task write(input bit [31:0] addr,
                                   input bit [31:0] data);
         ...
      endtask: write

endclass: apb_master

`endif
```

The READ and WRITE tasks are coded exactly as they would be if good old Verilog was used. It is a simple matter of assigning output signals to the proper value then sampling input signals at the right point in time. The only difference is that the physical signals are accessed through the **`clocking`** block of the **`virtual modport`** instead of pins on a module and they can only be assigned using nonblocking assignments. Similarly, the active clock edge is defined by waiting on the **`clocking`** block itself, not an edge of an input signal (Rule 4-7, 4-12).

File: apb/apb_master.sv

```
...
      virtual protected task read(input bit    [31:0] addr,
                                  output logic [31:0] data);
         this.sigs.mck.paddr   <= addr;
         this.sigs.mck.pwrite  <= '0;
         this.sigs.mck.psel    <= '1;
```

```
      @ (this.sigs.mck);
      this.sigs.mck.penable <= '1;
      @ (this.sigs.mck);
      data = this.sigs.mck.prdata;
      this.sigs.mck.psel    <= '0;
      this.sigs.mck.penable <= '0;
   endtask: read

   virtual protected task write(input bit [31:0] addr,
                                input bit [31:0] data);
      this.sigs.mck.paddr   <= addr;
      this.sigs.mck.pwdata  <= data;
      this.sigs.mck.pwrite  <= '1;
      this.sigs.mck.psel    <= '1;
      @ (this.sigs.mck);
      this.sigs.mck.penable <= '1;
      @ (this.sigs.mck);
      this.sigs.mck.psel    <= '0;
      this.sigs.mck.penable <= '0;
   endtask: write
```

Step 5: The First Test

Although not completely VMM-compliant, the transactor can now be used to exercise the DUT. It is instantiated in a verification environment class, extended from the **`vmm_env`** base class. The transactor is constructed in the extension of the **`vmm_env::build()`** method (Rule 4-34, 4-35) and started in the extension of the **`vmm_env::start()`** method (Rule 4-41). Note that if the transactor is required to configure the DUT in the extension of the **`vmm_env::cfg_dut()`** method, it needs to be started in that latter method. The reference to the *interface* encapsulating the APB physical signals is made using a hierarchical reference in the extension of the **`vmm_env::build()`** method.

File: Command_Master_Xactor/tb_env.sv

```
`ifndef TB_ENV__SV
`define TB_ENV__SV

`include "vmm.sv"
`include "apb_master.sv"

class tb_env extends vmm_env;
   apb_master mst;

   virtual function void build();
      super.build();
      this.mst = new("0", 0, tb_top.apb0);
   endfunction: build

   virtual task start();
      super.start();
      this.mst.start_xactor();
   endtask: start
   ...
endclass: tb_env

`endif
```

A simple test to perform a write followed by a read of the same address can now be written and executed to verify the correct operation of the transactor and the DUT interface. The test is written in a **`program`** (Rule 4-27) that instantiates the verification environment (Rule 4-28). The directed stimulus is created by instantiating transaction descriptors appropriately filled. It is a good idea to randomize these descriptors, only constraining those properties that are needed for the directed test. This way, any additional property will be randomized instead of defaulting to always the same value.

File: Command_Master_Xactor/test_simple.sv

```
`include "tb_env.sv"
```

```
program simple_test;

vmm_log log = new("Test", "Simple");
tb_env env = new;
initial begin
   apb_rw rd, wr;
   bit ok;

   env.start();

   wr = new;
   ok = wr.randomize() with {
      kind == WRITE;
   };
   if (!ok) begin
      `vmm_fatal(log, "Unable to randomize WRITE cycle");
   end
   env.mst.in_chan.put(wr);

   rd = new;
   ok = rd.randomize() with {
      kind == READ;
      addr == wr.addr;
   };
   if (!ok) begin
      `vmm_fatal(log, "Unable to randomize READ cycle");
   end
   env.mst.in_chan.put(rd);

   if (rd.data[15:0] !== wr.data[15:0]) begin
      `vmm_error(log, "Readback value != write value");
   end

   log.report();
   $finish();
end

endprogram
```

This test can be run many times, each time with a different seed, to verify the transactor and the DUT using different addresses.

Step 6: Extension Points

The transactor, as presently coded, will provide basic functionality. You may be tempted to stop here because you now have a transactor that can perform READ and WRITE cycles with identical capabilities to one you would have written using the old Verilog language.

The problem is that the transactor, as coded, is not very reusable. It will not be possible to modify the behavior of this transactor—for example to introduce delays between transactions, to synchronize the start of a transaction with some other external event, or modify a transaction to inject errors—without modifying the transactor itself or constantly rewrite the **`apb_master::read()`** and **`apb_master::write()`** virtual methods.

A callback method allows a user to extend the behavior of a transactor without having to modify the transactor itself. Callback methods should be provided before and after a transaction executes (Recommendations 4-155 and 4-156). The “pre-transaction” callback method allows errors to be injected and delays to be inserted. The “post-transaction” callback method allows delays to be inserted and the result of the transaction to be recorded in a functional coverage model or checked against an expected response.

The callback methods are first defined as **`virtual`** tasks or **`virtual void`** functions (Rule 4-160) in a callback façade class extended from the **`vmm_xactor_callbacks`** base class (Rule 4-

159). It is a good idea to create a mechanism in the “pre-transaction” callback method to allow an entire transaction to be skipped or dropped (Rule 4-160).

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include "apb_if.sv"
`include "apb_rw.sv"

typedef class apb_master;
class apb_master_cbs extends vmm_xactor_callbacks;
   virtual task pre_cycle(apb_master xactor,
                          apb_rw cycle,
                          ref bit drop);
   endtask: pre_cycle

   virtual task post_cycle(apb_master xactor,
                           apb_rw cycle);
   endtask: post_cycle
endclass: apb_master_cbs

class apb_master extends vmm_xactor;
   apb_rw_channel in_chan;
   virtual apb_if.master sigs;

   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.master sigs,
                apb_rw_channel        in_chan = null);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      if (in_chan == null)
         in_chan = new("APB Master Input Channel", name);
      this.in_chan = in chan;
      this.sigs.mck.psel <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: new
```

```
   virtual function void reset_xactor(
         reset_e rst_typ = SOFT_RST);
      super.reset(rst_typ);
      this.in_chan.flush();
      this.sigs.mck.psel <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      @ (this.sigs.mck);
      forever begin
         apb_rw tr;
         this.wait_if_stopped_or_empty(this.in_chan);
         this.in_chan.activate(tr);
         ...
         this.in_chan.start();
         case (tr.kind)
            apb_rw::READ : this.read(tr.addr, tr.data);
            apb_rw::WRITE: this.write(tr.addr, tr.data);
         endcase
         this.in_chan.complete();
         ...
         this.in_chan.remove();
      end
   endtask: main

   virtual protected task read(input  bit   [31:0] addr,
                               output logic [31:0] data);
      ...
   endtask: read

   virtual protected task write(input bit [31:0] addr,
                                input bit [31:0] data);
   ...
   endtask: write

endclass: apb_master

`endif
```

Next, the appropriate callback method needs to be invoked at the appropriate point in the execution of the transaction, using the **`vmm_callback()** macro (Rule 4-163).

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include "apb_if.sv"
`include "apb_rw.sv"

typedef class apb_master;
class apb_master_cbs extends vmm_xactor_callbacks;
   virtual task pre_cycle(apb_master xactor,
                          apb_rw     cycle,
                          ref bit    drop);
   endtask: pre_cycle

   virtual task post_cycle(apb_master xactor,
                           apb_rw     cycle);
   endtask: post_cycle
endclass: apb_master_cbs

class apb_master extends vmm_xactor;
   apb_rw_channel in_chan;
   virtual apb_if.master sigs;

   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.master sigs,
                apb_rw_channel        in_chan = null);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      if (in_chan == null)
         in_chan = new("APB Master Input Channel", name);
      this.in_chan = in chan;
      this.sigs.mck.psel <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: new
```

```
virtual function void reset_xactor(
      reset_e rst_typ = SOFT_RST);
   super.reset(rst_typ);
   this.in_chan.flush();
   this.sigs.mck.psel <= '0;
   this.sigs.mck.penable <= '0;
endfunction: reset_xactor

virtual protected task main();
   super.main();
   @ (this.sigs.mck);
   forever begin
      apb_rw tr;
      bit drop;

      this.wait_if_stopped_or_empty(this.in_chan);
      this.in_chan.activate(tr);

      drop = 0;
      `vmm_callback(apb_master_cbs, pre_cycle(
         this, tr, drop));
      if (drop) begin
         this.in_chan.remove();
         continue;
      end

      this.in_chan.start();
      case (tr.kind)
         apb_rw::READ : this.read(tr.addr, tr.data);
         apb_rw::WRITE: this.write(tr.addr, tr.data);
      endcase
      this.in_chan.complete();

      `vmm_callback(apb_master_cbs, post_cycle(
         this, tr));

      this.in_chan.remove();
   end
endtask: main

virtual protected task read(input  bit    [31:0] addr,
                            output logic [31:0] data);
```

```
      ...
   endtask: read

   virtual protected task write(input bit [31:0] addr,
                                input bit [31:0] data);
   ...
   endtask: write

endclass: apb_master

`endif
```

Step 7: Debug Messages

To be truly reusable, it should be possible to understand what the transactor does and debug its operation without having to inspect the source code. This capability may even be a basic requirement if you plan on shipping encrypted or compiled code.

Debug messages should be added at judicious points to indicate what the transactor is about to do, is doing or has done. These debug messages are inserted using the **`vmm_trace()**, **`vmm_debug()** or **`vmm_verbose()** macros (Recommendation 4-51).

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include "apb_if.sv"
`include "apb_rw.sv"

typedef class apb_master;
class apb_master_cbs extends vmm_xactor_callbacks;
   virtual task pre_cycle(apb_master xactor,
                          apb_rw     cycle,
                          ref bit    drop);
```

```
   endtask: pre_cycle

   virtual task post_cycle(apb_master xactor,
                           apb_rw     cycle);
   endtask: post_cycle
endclass: apb_master_cbs

class apb_master extends vmm_xactor;
   apb_rw_channel in_chan;
   virtual apb_if.master sigs;

   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.master sigs,
                apb_rw_channel        in_chan = null);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      if (in_chan == null)
         in_chan = new("APB Master Input Channel", name);
      this.in_chan = in chan;
      this.sigs.mck.psel <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: new

   virtual function void reset_xactor(
         reset_e rst_typ = SOFT_RST);
      super.reset(rst_typ);
      this.in_chan.flush();
      this.sigs.mck.psel <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      @ (this.sigs.mck);
      forever begin
         apb_rw tr;
         bit drop;

         this.wait_if_stopped_or_empty(this.in_chan);
         this.in_chan.activate(tr);
```

```
            drop = 0;
            `vmm_callback(apb_master_cbs, pre_cycle(
               this, tr, drop));
            if (drop) begin
               `vmm_debug(log, {"Dropping transaction...\n",
                  tr.psdisplay("   ")});
               this.in_chan.remove();
               continue;
            end

         `vmm_trace(log, {"Starting transaction...\n",
            tr.psdisplay("   ")});

            this.in_chan.start();
            case (tr.kind)
               apb_rw::READ : this.read(tr.addr, tr.data);
               apb_rw::WRITE: this.write(tr.addr, tr.data);
            endcase
            this.in_chan.complete();

            `vmm_trace(log, {"Completed transaction...\n",
                             tr.psdisplay("   ")});

            `vmm_callback(apb_master_cbs, post_cycle(
               this, tr));

            this.in_chan.remove();
         end
      endtask: main

      virtual protected task read(input bit    [31:0] addr,
                                  output logic [31:0] data);
         ...
      endtask: read

      virtual protected task write(input bit [31:0] addr,
                                   input bit [31:0] data);
         ...
      endtask: write

   endclass: apb_master
```

```
`endif
```

Step 8: Standard Methods

You can now run the “simple test” with the latest version of the transactor and increase the message verbosity to see debug messages displayed as the test executes the various transactions.

File: Command_Master_Xactor/Makefile

```
% vcs -sverilog -ntb_opts vmm +vmm_log_default=trace ...
```

But the messages will be not very useful as they do not display the content of the executed transactions. That’s because the **`psdisplay()`** method, defined in the **`vmm_data`** base class does not know about the content of the APB transaction descriptor. For that method to display the information that is relevant for the APB transaction, it is necessary to overload this method in the transaction descriptor class (Rule 4-76).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   rand enum {READ, WRITE} kind;
   rand bit   [31:0] addr;
   rand logic [31:0] data;

   function new();
      super.new(this.log);
```

```
   endfunction: new
   ...
   virtual function string psdisplay(string prefix = "");
      $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
      this.kind.name(), this.addr, this.data);
   endfunction: psdisplay
   ...
endclass: apb_rw

`vmm_channel(apb_rw)
...
`endif
```

Re-running the test with increased verbosity now yields useful and meaningful debug messages. The **`vmm_data::psdisplay()`** method is one of the pre-defined methods in the **`vmm_data`** base class that users expect to be provided to simplify and abstract common operations on transaction descriptors. These common operations include creating transaction descriptors (**`vmm_data::allocate()`**), copying transaction descriptors (**`vmm_data::copy()`**), comparing transaction descriptors (**`vmm_data::compare()`**) and checking that the content of transaction descriptors is valid (**`vmm_data::is_valid()`**) (Rule 4-76).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   rand enum {READ, WRITE} kind;
   rand bit   [31:0] addr;
   rand logic [31:0] data;
```

```
function new();
   super.new(this.log);
endfunction: new

virtual function vmm_data allocate();
   apb_rw tr = new;
   return tr;
endfunction: allocate

virtual function vmm_data copy(vmm_data to = null);
   apb_rw tr;
   if (to == null) tr = new;
   else if (!$cast(tr, to)) begin
      `vmm_fatal(log, "Cannot copy into non-apb_rw \n
        instance");
      return null;
   end

   super.copy_data(tr);
   tr.kind = this.kind;
   tr.addr = this.addr;
   tr.data = this.data;
   return tr;
endfunction: copy

virtual function string psdisplay(string prefix = "");
   $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
            this.kind.name(), this.addr, this.data);
endfunction: psdisplay

virtual function bit is_valid(bit silent = 1,
                              int kind = -1);
   return 1;
endfunction: is_valid

virtual function bit compare(input vmm_data to,
                             output string  diff,
                             input int      kind = -1);
   apb_rw tr;

   if (to == null) begin
```

```
            `vmm_fatal(log, "Cannot compare to NULL \n
               reference");
            return 0;
         end
         else if (!$cast(tr, to)) begin
            `vmm_fatal(log, "Cannot compare against \n
               non-apb_rw  instance");
            return 0;
         end

         if (this.kind != tr.kind) begin
            $sformat(diff, "Kind %s !== %s", this.kind,
               tr.kind);
            return 0;
         end

         if (this.addr !== tr.addr) begin
            $sformat(diff, "Addr 0x%h !== 0x%h", this.addr,
               tr.addr);
            return 0;
         end

         if (this.data !== tr.data) begin
            $sformat(diff, "Data 0x%h !== 0x%h", this.data,
               tr.data);
            return 0;
         end

         return 1;
      endfunction: compare

endclass: apb_rw

`vmm_channel(apb_rw)
...
`endif
```

Three other standard methods, **`vmm_data:byte_size()`**, **`vmm_data::byte_pack()`** and **`vmm_data::byte_unpack()`** should also be overloaded for packet-oriented transactions, where the content of the transaction is transmitted over a physical interface (Recommendation 4-77).

Step 9: Transaction Generator

To promote the use of random stimulus, it is a good idea to pre-define random transaction generators whenever transaction descriptors are defined. It is a simple matter of using the **`` `vmm_atomic_gen() ``** and **`` `vmm_scenario_gen() ``** macros (Recommendation 5-23, 5-24) in the transaction descriptor file.

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   rand enum {READ, WRITE} kind;
   rand bit   [31:0] addr;
   rand logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new

   virtual function vmm_data allocate();
      apb_rw tr = new;
      return tr;
   endfunction: allocate
```

```
virtual function vmm_data copy(vmm_data to = null);
   apb_rw tr;

   if (to == null) tr = new;
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot copy into non-apb_rw \n
           instance");
      return null;
   end

   super.copy_data(tr);
   tr.kind = this.kind;
   tr.addr = this.addr;
   tr.data = this.data;

   return tr;
endfunction: copy

virtual function string psdisplay(string prefix = "");
   $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
      this.kind.name(), this.addr, this.data);
endfunction: psdisplay

virtual function bit is_valid(bit silent = 1,
                              int kind   = -1);
   return 1;
endfunction: is_valid

virtual function bit compare(input vmm_data to,
                             output string  diff,
                             input int      kind = -1);
   apb_rw tr;

   if (to == null) begin
      `vmm_fatal(log, "Cannot compare to NULL \n
        reference");
      return 0;
   end
   else if (!$cast(tr, to)) begin
      `vmm_fatal(log, "Cannot compare against \n
        non-apb_rw instance");
```

```
            return 0;
         end

         if (this.kind != tr.kind) begin
         $sformat(diff, "Kind %s !== %s", this.kind, tr.kind);
            return 0;
         end

         if (this.addr !== tr.addr) begin
            $sformat(diff, "Addr 0x%h !== 0x%h", this.addr, \n
               tr.addr);
            return 0;
         end

         if (this.data !== tr.data) begin
            $sformat(diff, "Data 0x%h !== 0x%h", this.data, \n
               tr.data);
            return 0;
         end

         return 1;
      endfunction: compare

endclass: apb_rw

`vmm_channel(apb_rw)
`vmm_atomic_gen(apb_rw, "APB Bus Cycle")
`vmm_scenario_gen(apb_rw, "APB Bus Cycle")

`endif
```

Step 10: Top-Level File

To help users include all necessary files, without having to know the detailed filenames and file structure of the transactor, interface and transaction descriptor, it is a good idea to create a top-level file that will automatically include all source files that make up the verification IP for a protocol.

File: apb/apb.sv

```
`ifndef APB__SV
`define APB__SV

`include "vmm.sv"
`include "apb_if.sv"
`include "apb_rw.sv"
`include "apb_master.sv"

`endif
```

In this example, we implemented only a master transactor; but a complete VIP for a protocol would also include a slave transactor and a passive monitor transactor. All of these transactors would be included in the top-level file.

Step 11: Congratulations!

You have now completed the creation of a VMM-compliant command-layer master transactor!

Upon reading this primer, you probably realized that there is much code that is similar across different master transactors. Wouldn't be nice if you could simply cut-and-paste from an existing VMM-compliant master transactor and only modify what is unique or different for your protocol? That can easily be done using the **`vmmgen`** tool provided with VCS 2006.06-6. Based on a few simple question and answers, it will create a template for various components of a VMM-compliant master transactor.

Command

```
% vmmgen –l sv
```

The relevant templates for writing command-layer master transactors are:

1. Physical interface declaration
2. Transaction Descriptor
3. Driver, Physical-level, Half-duplex

Note that the menu number used to select the appropriate template may differ from the number in the above list.

You may consider reading other publications in this series to learn how to write VMM-compliant command-layer slave transactors, command-layer passive monitor transactors, functional-layer transactors or verification environments.

VMM Primer:
Writing Command-Layer
Slave Transactors

Author:
Janick Bergeron

Version 0.4 / May17, 2007

Introduction

The *Verification Methodology Manual for SystemVerilog* book was never written as a training book. It was designed to be a reference document that defines what is—and is not—compliant with the methodology described within it.

This primer is designed to learn how to write VMM-compliant command-layer slave transactors—transactors that react to pin wiggling and supply the necessary response back. Other primers will eventually cover other aspects of developing VMM-compliant verification assets, such as master transactors, monitors, functional-layer transactors, generators, assertions and verification environments.

The protocol used in this primer was selected for its simplicity. Because of its simplicity, it does not require the use of many elements of the VMM standard library. It is sufficient to achieve the goal of demonstrating, step by step, how to create a simple VMM-compliant slave transactor.

This document is written in the same order you would implement a command-layer slave transactor. As such, you should read it in a sequential fashion. You can use the same sequence to create your own specific transactor.

A word of caution however: it may be tempting to stop reading this primer half way through, as soon as a functional transactor is available. VMM compliance is a matter of degree. Once a certain minimum level of functionality is met, a slave transactor may be declared VMM compliant. But additional VMM functionality—such as

a request/response model to a higher-level transactor or callbacks—will make it much easier to use in different verification environments. Therefore, you should read—and apply—this primer in its entirety.

This primer will show how to apply the various VMM guidelines, not attempt to justify them or present alternatives. If you are interested in learning more about the justifications of various techniques and approaches used in this primer, you should refer to the VMM book under the relevant quoted rules and recommendations.

The Protocol

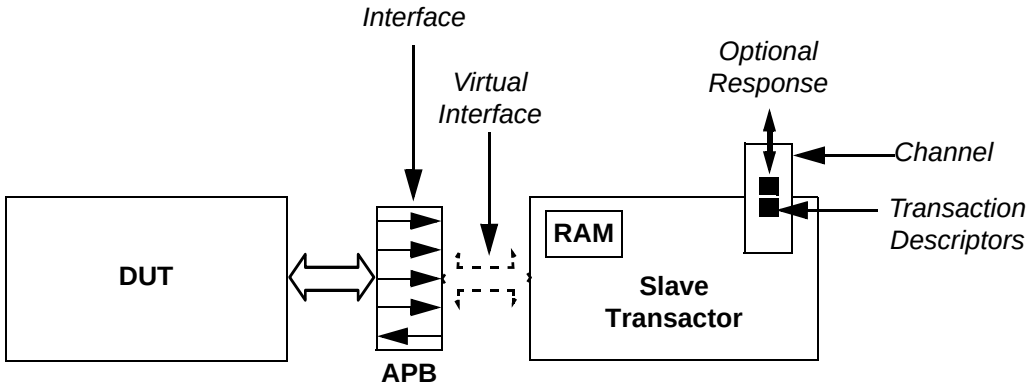
The protocol used in this primer is the AMBA™ Peripheral Bus (APB) protocol. It is a simple single-master address-based parallel bus providing atomic individual read and write cycles. The protocol specification can be found in the AMBA™ Specification (Rev 2.0) available from ARM (http://arm.com).

When writing a reusable slave transactor, you have to think about all possible applications it may be used in, not just the device you are using it for the first time. Therefore, even though the device you will be initially using with this transactor supports 8 address bits and 16 data bits, the APB slave transactor should be written for the entire 32-bit of address and data information.

The Verification Components

Figure 1 illustrates the various components that will be created throughout this primer. A command-layer slave transactor interfaces directly to the DUT signals and optionally requests transaction data fulfillment on a transaction-level interface.

Figure 1 Components Used in This Primer



Step 1: The Interface

The first step is to define the physical signals used by the protocol to exchange information between a master and a slave. A single exchange of information (a READ or a WRITE operation) is called a *transaction*. There may be multiple slaves on an APB bus but there can only be one master. Slaves are differentiated by responding to different address ranges.

The signals are declared inside an *interface* (Rule 4-4). The name of the interface is prefixed with “apb_” to identify that it belongs to the APB protocol (Rule 4-5). The entire content of the file declaring the

`interface` is embedded in an **`` `ifndef/`define/`endif ``** construct. This is an old C trick that allows the file to be included multiple time, whenever required, without causing multiple-definition errors.

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if;
   ...
endinterface: apb_if

`endif
```

The signals, listed in the AMBA™ Specification in Section 2.4, are declared as **`wires`** (Rule 4-6) inside the **`interface`**.

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if(input bit pclk);
   wire [31:0] paddr;
   wire        psel;
   wire        penable;
   wire        pwrite;
   wire [31:0] prdata;
   wire [31:0] pwdata;
   ...
endinterface: apb_if

`endif
```

Because this is a synchronous protocol, **`clocking`** blocks are used to define the direction and sampling of the signals (Rule 4-7, 4-11).

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if(input bit pclk);
   wire [31:0] paddr;
   wire        psel;
   wire        penable;
   wire        pwrite;
   wire [31:0] prdata;
   wire [31:0] pwdata;

   clocking sck @(posedge pclk);
      input  paddr, psel, penable, pwrite, pwdata;
      output prdata;
   endclocking: sck
   ...
endinterface: apb_if

`endif
```

The **`clocking`** block defining the synchronous signals is specified in the **`modport`** for the APB slave transactor (Rule 4-9, 4-11, 4-12). The clock signal need not be specified as it is implicit in the **`clocking`** block.

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if(input bit pclk);
   wire [31:0] paddr;
   wire        psel;
   wire        penable;
   wire        pwrite;
   wire [31:0] prdata;
   wire [31:0] pwdata;

   clocking sck @(posedge pclk);
      input  paddr, psel, penable, pwrite, pwdata;
      output prdata;
   endclocking: sck
```

```
   modport slave(clocking sck);

endinterface: apb_if

`endif
```

The **interface** declaration is now sufficient for writing a APB slave transactor. To be fully compliant, it should eventually include a **modport** for a master and a passive monitor transactor (Rule 4-9). These can be added later, when these transactors will be written.

Step 2: Connecting to the DUT

The interface may now be connected to the DUT. It is instantiated in a top-level **module**, alongside of the DUT instantiation (Rule 4-13). The connection to the DUT pins are specified using a hierarchical reference to the **wires** in the **interface** instance.

File: Command_Slave_Xactor/tb_top.sv

```
module tb_top;
   ...
   apb_if apb0(...);
   ...
   master_ip dut_mst(...,
                     .apb_addr   (apb0.paddr[7:0]  ),
                     .apb_sel    (apb0.psel        ),
                     .apb_enable (apb0.penable     ),
                     .apb_write  (apb0.pwrite      ),
                     .apb_rdata  (apb0.prdata[15:0]),
                     .apb_wdata  (apb0.pwdata[15:0]),
                     ...);

endmodule: tb_top
```

This top-level module also contains the clock generators (Rule 4-15), using the **`bit`** type (Rule 4-17) and ensuring that no clock edges will occur at time zero (Rule 4-16).

File: Command_Slave_Xactor/tb_top.sv

```
module tb_top;
   bit clk = 0;
   apb_if apb0(clk);
   ...

   my_design dut(...,
                 .apb_addr   (apb0.paddr[7:0]   ),
                 .apb_sel    (apb0.psel         ),
                 .apb_enable (apb0.penable      ),
                 .apb_write  (apb0.pwrite       ),
                 .apb_rdata  (apb0.prdata[15:0]),
                 .apb_wdata  (apb0.pwdata[15:0]),
                 ...
                 .clk        (clk));

   always #10 clk = ~clk;
endmodule: tb_top
```

Step 3: The Transaction Descriptor

The next step is to define the APB transaction descriptor (Rule 4-54). This descriptor is a class (Rule 4-53) extended from the **`vmm_data`** class (Rule 4-55), containing a public property enumerating the various transactions that can be observed and reacted to by the slave transactor (Rule 4-60, 4-62) and public properties for each parameter or value in the transaction (Rule 4-59, 4-62). It also needs a **`static vmm_log`** property instance used to issue messages from the transaction descriptor. This instance of the message service interface is passed to the **`vmm_data`** constructor (Rule 4-58).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   enum {READ, WRITE} kind;
   bit   [31:0] addr;
   logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new
   ...
endclass: apb_rw
...
`endif
```

A single property is used for “data”, despite the fact that the APB bus has separate read and write data buses. Because the APB does not support concurrent read/write transactions, there can only be one data value valid at any given time. The **`data`** class property is interpreted differently depending on the transaction kind (Rule 4-71). In a WRITE transaction, it is interpreted as the data to be written. In a READ transaction, it is the data value that is to be driven as the read value. The type for the **`data`** property is **`logic`** as it will allow the description of READ cycles to reflect unknown results.

Although the transaction descriptor is not yet VMM-compliant, it is has the minimum functionality to be used by a slave transactor.

Step 4: The Slave Transactor

The slave transactor can now be started. It is a class (Rule 4-91) derived from the **`vmm_xactor`** base class (4-92).

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV
...
class apb_slave extends vmm_xactor;
   ...
endclass: apb_slave

`endif
```

The transactor needs a physical-level interface to observe and react to transactions. The physical-level interface is done using a **`virtual modport`** passed to the transactor as a constructor argument (Rule 4-108) and is saved in a public property (Rule 4-109).

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   ...
   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.slave  sigs,
                ...);
      super.new("APB Slave", name, stream_id);
      this.sigs = sigs;
      ...
```

```
   endfunction: new
   ...
endclass: apb_slave

`endif
```

The transaction descriptors are filled in as much as possible from observations on the physical interface and a response is determined and driven in reply in the `main()` task (Rule 4-93). The observation and reaction to READ and WRITE transactions is coded exactly as it would be if good old Verilog was used. It is a simple matter of sampling input signals and driving an appropriate response at the right point in time. The only difference is that the physical signals are accessed through the **`clocking`** block of the **`virtual modport`** instead of pins on a module. The active clock edge is defined by waiting on the **`clocking`** block itself, not an edge of an input signal (Rule 4-7, 4-12).

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   ...
   function new(string               name,
                int unsigned         stream_id,
                virtual apb_if.slave sigs,
                ...);
      super.new("APB Slave", name, stream_id);
      this.sigs = sigs;
      ...
   endfunction: new
   ...
   virtual protected task main();
      super.main();
      forever begin
```

```
            apb_rw tr;

            this.sigs.sck.prdata <= 'z;
            ...
            // Wait for a SETUP cycle
            do @ (this.sigs.sck);
            while (this.sigs.sck.psel !== 1'b1 ||
                   this.sigs.sck.penable !== 1'b0);
            tr = new;

            tr.kind = (this.sigs.sck.pwrite) ?
                        apb_rw::WRITE : apb_rw::READ;
            tr.addr = this.sigs.sck.paddr;

            if (tr.kind == apb_rw::READ) begin
               ...
               this.sigs.sck.prdata <= tr.data;
               @ (this.sigs.sck);
            end
            else begin
               @ (this.sigs.sck);
               tr.data = this.sigs.sck.pwdata;
               ...
            end
            if (this.sigs.sck.penable !== 1'b1) begin
              `vmm_error(this.log, "APB protocol violation:\n
                    SETUP cycle not followed by ENABLE cycle");
            end
            ...
      end
      endtask: main
endclass: apb_slave

`endif
```

Once a slave transactor recognizes the start of a transaction, it must not be stopped until it completes the APB transaction. Otherwise, if the transactor is stopped in the middle of a transaction stream, it will violate the protocol. Therefore, the **`vmm_xactor::wait_if_stopped()`** method is called only before the beginning of transaction.

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   ...
   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.slave sigs,
                ...);
      super.new("APB Slave", name, stream_id);
      this.sigs = sigs;
      ...
   endfunction: new
   ...
   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;

         this.sigs.sck.prdata <= 'z;
         this.wait_if_stopped();
         ...
         // Wait for a SETUP cycle
         do @ (this.sigs.sck);
         while (this.sigs.sck.psel !== 1'b1 ||
                this.sigs.sck.penable !== 1'b0);
         tr = new;

         tr.kind = (this.sigs.sck.pwrite) ?
                      apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.sck.paddr;

         if (tr.kind == apb_rw::READ) begin
            ...
            this.sigs.sck.prdata <= tr.data;
            @ (this.sigs.sck);
         end
         else begin
            @ (this.sigs.sck);
            tr.data = this.sigs.sck.pwdata;
            ...
```

```
            end
            if (this.sigs.sck.penable !== 1'b1) begin
               `vmm_error(this.log, "APB protocol violation:\n
                     SETUP cycle not followed by ENABLE cycle");
            end
            ...
   end
   endtask: main
endclass: apb_slave

`endif
```

But what is the slave to do with the transaction data? The simplest behavior to implement is a RAM response. Data from WRITE cycles is written at the specified address. Data returned in READ cycles is taken from the specified address. It is unrealistic to model a 32-bit RAM of 32-bit words as a fixed-sized array: it would take a lot of memory when only a very few locations needed to be used. Instead, an associative array should be used. If an address is read that has never been written before, unknowns ('bx) are returned.

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   ...
   local bit [31:0] ram[*];

   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.slave sigs,
                ...);
      super.new("APB Slave", name, stream_id);
      this.sigs = sigs;
      ...
   endfunction: new
```

```
   ...
   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;

         this.sigs.sck.prdata <= 'z;
         this.wait_if_stopped();
         ...
         // Wait for a SETUP cycle
         do @ (this.sigs.sck);
         while (this.sigs.sck.psel !== 1'b1 ||
                this.sigs.sck.penable !== 1'b0);
         tr = new;

         tr.kind = (this.sigs.sck.pwrite) ?
                      apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.sck.paddr;

         if (tr.kind == apb_rw::READ) begin
            if (!this.ram.exists(tr.addr)) tr.data = 'x;
            else tr.data = this.ram[tr.addr];
            ...
            this.sigs.sck.prdata <= tr.data;
            @ (this.sigs.sck);
         end
         else begin
            @ (this.sigs.sck);
            tr.data = this.sigs.sck.pwdata;
            ...
            this.ram[tr.addr] = tr.data;
         end
         if (this.sigs.sck.penable !== 1'b1) begin
           `vmm_error(this.log, "APB protocol violation:\n
                SETUP cycle not followed by ENABLE cycle");
         end
         ...
   end
   endtask: main
endclass: apb_slave

`endif
```

Step 5: Reporting Transactions

The slave transactor must have the ability to inform the testbench that a specific transaction has been observed and reacted to and what—if any—data was supplied in answer to the transaction. Simply displaying the transactions is not very useful as it will require that the results be manually checked every time. Observed transactions can be reported by indicating a notification in the **`vmm_xactor::notify`** property. The observed transaction, being derived from **`vmm_data`**, is attached to a newly defined "RESPONSE" notification and can be recovered by all interested parties waiting on the notification.

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   ...
   typedef enum {RESPONSE} notifications_e;
   ...
   local bit [31:0] ram[*];

   function new(string               name,
                int unsigned         stream_id,
                virtual apb_if.slave sigs,
                ...);
      super.new("APB Slave", name, stream_id);
      this.sigs = sigs;
      ...
      this.notify.configure(RESPONSE);
   endfunction: new
   ...
   virtual protected task main();
      super.main();
```

```
      forever begin
         apb_rw tr;

         this.sigs.sck.prdata <= 'z;
         this.wait_if_stopped();
         ...
         // Wait for a SETUP cycle
         do @ (this.sigs.sck);
         while (this.sigs.sck.psel !== 1'b1 ||
                this.sigs.sck.penable !== 1'b0);
         tr = new;

         tr.kind = (this.sigs.sck.pwrite) ?
                      apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.sck.paddr;

         if (tr.kind == apb_rw::READ) begin
            if (!this.ram.exists(tr.addr)) tr.data = 'x;
            else tr.data = this.ram[tr.addr];
            ...
            this.sigs.sck.prdata <= tr.data;
            @ (this.sigs.sck);
         end
         else begin
            @ (this.sigs.sck);
            tr.data = this.sigs.sck.pwdata;
            ...
            this.ram[tr.addr] = tr.data;
         end
         if (this.sigs.sck.penable !== 1'b1) begin
           `vmm_error(this.log, "APB protocol violation:\n
                SETUP cycle not followed by ENABLE cycle");
         end
         ...
         this.notify.indicate(RESPONSE, tr);
   end
   endtask: main
endclass: apb_slave

`endif
```

Step 6: The First Test

Although not completely VMM-compliant, the slave transactor can now be used to reply to activity on an APB bus. It is instantiated in a verification environment class, extended from the **`vmm_env`** base class. The slave transactor is constructed in the extension of the **`vmm_env::build()`** method (Rule 4-34, 4-35) and started in the extension of the **`vmm_env::start()`** method (Rule 4-41). The reference to the **`interface`** encapsulating the APB physical signals is made using a hierarchical reference in the extension of the **`vmm_env::build()`** method.

File: Command_Slave_Xactor/tb_env.sv

```
`ifndef TB_ENV__SV
`define TB_ENV__SV

`include "vmm.sv"
`include "apb_slave.sv"

class tb_env extends vmm_env;
   apb_slave slv;
   ...
   virtual function void build();
      super.build();
      this.slv = new("0", 0, tb_top.apb0);
   endfunction: build

   virtual task start();
      super.start();
      this.slv.start_xactor();
      ...
   endtask: start
   ...
endclass: tb_env

`endif
```

As the simulation progress, the slave transactor reports the completed transactions. The responses of the slave transactor can also be used to determine that it is time to end the test when enough APB transactions have been executed.

File: Command_Slave_Xactor/tb_env.sv

```
`ifndef TB_ENV__SV
`define TB_ENV__SV

`include "vmm.sv"
`include "apb_slave.sv"

class tb_env extends vmm_env;
   apb_slave slv;

   int stop_after = 10;

   virtual function void build();
      super.build();
      this.slv = new("0", 0, tb_top.apb0);
   endfunction: build

   virtual task start();
      super.start();
      this.slv.start_xactor();

      fork
         forever begin
            apb_rw tr;
            this.slv.notify.wait_for(apb_slave::RESPONSE);
            this.stop_after--;
            if (this.stop_after <= 0) -> this.end_test;

            $cast(tr,
             this.slv.notify.status(apb_slave::RESPONSE));
            tr.display("Responded: ");
         end
      join_none
   endtask: start

   virtual task wait_for_end();
      super.wait_for_end();
      @ (this.end_test);
```

```
   endtask: wait_for_end

endclass: tb_env

`endif
```

A test can control how long the simulation should run by simply setting the value of the **`tb_env::stop_after`** property. The test is written in a **`program`** (Rule 4-27) that instantiates the verification environment (Rule 4-28).

File: Command_Slave_Xactor/test_simple

```
`include "tb_env.sv"

program simple_test;

vmm_log log = new("Test", "Simple");
tb_env env = new;
initial begin
   env.stop_after = 5;
   env.run();

   $finish();
end

endprogram
```

Step 7: Standard Methods

The transaction responses reported by the slave transactors are not very useful as they do not show the content of the transactions. That's because the **`psdisplay()`** method, defined in the **`vmm_data`** base class does not know about the content of the APB transaction descriptor. For that method to display the information that is relevant for the APB transaction, it is necessary to overload this method in the transaction descriptor class (Rule 4-76).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   enum {READ, WRITE} kind;
   bit   [31:0] addr;
   logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new
   ...
   virtual function string psdisplay(string prefix = "");
     $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
              this.kind.name(), this.addr, this.data);
   endfunction: psdisplay
   ...
endclass: apb_rw
...
`endif
```

Re-running the test now yields useful and meaningful transaction response reporting. The **`vmm_data::psdisplay()`** method is one of the pre-defined methods in the **`vmm_data`** base class that users expect to be provided to simplify and abstract common operations on transaction descriptors. These common operations include creating transaction descriptors (**`vmm_data::allocate()`**), copying transaction descriptors (**`vmm_data::copy()`**), comparing transaction descriptors (**`vmm_data::compare()`**) and checking that the content of transaction descriptors is valid (**`vmm_data::is_valid()`**) (Rule 4-76).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV
```

```
`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   enum {READ, WRITE} kind;
   bit   [31:0] addr;
   logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new

   virtual function vmm_data allocate();
      apb_rw tr = new;
      return tr;
   endfunction: allocate

   virtual function vmm_data copy(vmm_data to = null);
      apb_rw tr;

      if (to == null) tr = new;
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot copy into non-apb_rw \n
                     instance");
         return null;
      end

      super.copy_data(tr);
      tr.kind = this.kind;
      tr.addr = this.addr;
      tr.data = this.data;

      return tr;
   endfunction: copy

   virtual function string psdisplay(string prefix = "");
      $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
               this.kind.name(), this.addr, this.data);
   endfunction: psdisplay

   virtual function bit is_valid(bit silent = 1,
                                 int kind   = -1);
      return 1;
   endfunction: is_valid
```

```
   virtual function bit compare(input  vmm_data to,
                                output string   diff,
                               input  int      kind = -1);
      apb_rw tr;

      if (to == null) begin
       `vmm_fatal(log, "Cannot compare to NULL reference");
         return 0;
      end
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot compare against \n
                     non-apb_rw  instance");
         return 0;
      end

      if (this.kind != tr.kind) begin
       $sformat(diff, "Kind %s !== %s", this.kind, tr.kind);
         return 0;
      end

      if (this.addr !== tr.addr) begin
         $sformat(diff, "Addr 0x%h !== 0x%h", this.addr,
                  tr.addr);
         return 0;
      end

      if (this.data !== tr.data) begin
         $sformat(diff, "Data 0x%h !== 0x%h", this.data,
                  tr.data);
         return 0;
      end

      return 1;
   endfunction: compare

endclass: apb_rw
...
`endif
```

Three other standard methods, **`vmm_data:byte_size()`**, **`vmm_data::byte_pack()`** and **`vmm_data::byte_unpack()`** should also be overloaded for packet-oriented transactions, where the content of the transaction is transmitted over a physical interface (Recommendation 4-77).

Step 8: Debug Messages

To be truly reusable, it should be possible to understand what the slave transactor does and debug its operation without having to inspect the source code. This capability may even be a basic requirement if you plan on shipping encrypted or compiled code.

Debug messages should be added at judicious points to indicate what the slave transactor is about to do, is doing or has done. These debug messages are inserted using the **`vmm_trace()`**, **`vmm_debug()`** or **`vmm_verbose()`** macros (Recommendation 4-51).

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   ...
   typedef enum {RESPONSE} notifications_e;
   ...
   local bit [31:0] ram[*];

   function new(string               name,
                int unsigned         stream_id,
                virtual apb_if.slave sigs,
```

```
                ...);
    super.new("APB Slave", name, stream_id);
    this.sigs = sigs;
    ...
    this.notify.configure(RESPONSE);
endfunction: new
...
virtual protected task main();
    super.main();
    forever begin
       apb_rw tr;

       this.sigs.sck.prdata <= 'z;
       this.wait_if_stopped();

   `vmm_trace(log, "Waiting for start of transaction...");
       ...
       // Wait for a SETUP cycle
       do @ (this.sigs.sck);
       while (this.sigs.sck.psel !== 1'b1 ||
              this.sigs.sck.penable !== 1'b0);
       tr = new;

       tr.kind = (this.sigs.sck.pwrite) ?
                    apb_rw::WRITE : apb_rw::READ;
       tr.addr = this.sigs.sck.paddr;

      `vmm_trace(log, {"Responding to transaction...\n",
                       tr.psdisplay("   ")});
       if (tr.kind == apb_rw::READ) begin
          if (!this.ram.exists(tr.addr)) tr.data = 'x;
          else tr.data = this.ram[tr.addr];
          ...
          this.sigs.sck.prdata <= tr.data;
          @ (this.sigs.sck);
       end
       else begin
          @ (this.sigs.sck);
          tr.data = this.sigs.sck.pwdata;
          ...
          this.ram[tr.addr] = tr.data;
       end
       if (this.sigs.sck.penable !== 1'b1) begin
         `vmm_error(this.log, "APB protocol violation:\n
               SETUP cycle not followed by ENABLE cycle");
       end
```

```
            ...
            `vmm_trace(log, {"Responded to transaction...\n",
                             tr.psdisplay("    ")});
            this.notify.indicate(RESPONSE, tr);
      end
      endtask: main
endclass: apb_slave

`endif
```

You can now run the “simple test” with the latest version of the slave transactor and increase the message verbosity to see debug messages displayed as the slave transactors observes and responds to the various transactions.

File: Makefile

```
% vcs -sverilog -ntb_opts vmm +vmm_log_default=trace ...
```

Step 9: Slave Configuration

The slave transactor, as currently coded, responds to any and all transactions on the APB bus. It would not be able to cooperate with other slaves on the same bus mapped to different address ranges. The slave must thus be configurable to limit its response to a specified address range (Rule 4-104). The slave is configured using a configuration descriptor (Rule 4-105). The start and end addresses are given default values for the entire address range but can also be randomized to create a random slave configuration. A constraint block is specified to guarantee that any random configuration is valid (Rule 4-80). The slave configuration is specified as an optional argument to the constructor (Rule 4-106).

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
```

```
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"

class apb_slave_cfg;
   rand bit [31:0] start_addr = 32'h0000_0000;
   rand bit [31:0] end_addr   = 32'hFFFF_FFFF;

   constraint apb_slave_cfg_valid {
      end_addr >= start_addr;
   }
   ...
endclass: apb_slave_cfg
...
class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   local apb_slave_cfg cfg;
   ...
   typedef enum {RESPONSE} notifications_e;
   ...
   local bit [31:0] ram[*];

   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.slave  sigs,
                apb_slave_cfg         cfg = null,
                ...);
      super.new("APB Slave", name, stream_id);
      this.sigs = sigs;

      if (cfg == null) cfg = new;
      this.cfg = cfg;
      ...
      this.notify.configure(RESPONSE);
   endfunction: new
   ...
   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;

         this.sigs.sck.prdata <= 'z;
         this.wait_if_stopped();

     `vmm_trace(log, "Waiting for start of transaction...");
```

```
            ...
            // Wait for a SETUP cycle
            do @ (this.sigs.sck);
            while (this.sigs.sck.psel !== 1'b1 ||
                   this.sigs.sck.penable !== 1'b0 ||
                 this.sigs.sck.paddr < this.cfg.start_addr ||
                  this.sigs.sck.paddr > this.cfg.end_addr);
            tr = new;

            tr.kind = (this.sigs.sck.pwrite) ?
                         apb_rw::WRITE : apb_rw::READ;
            tr.addr = this.sigs.sck.paddr;

           `vmm_trace(log, {"Responding to transaction...\n",
                            tr.psdisplay("   ")});
            if (tr.kind == apb_rw::READ) begin
               if (!this.ram.exists(tr.addr)) tr.data = 'x;
               else tr.data = this.ram[tr.addr];
               ...
               this.sigs.sck.prdata <= tr.data;
               @ (this.sigs.sck);
            end
            else begin
               @ (this.sigs.sck);
               tr.data = this.sigs.sck.pwdata;
               ...
               this.ram[tr.addr] = tr.data;
            end
            if (this.sigs.sck.penable !== 1'b1) begin
              `vmm_error(this.log, "APB protocol violation:\n
                   SETUP cycle not followed by ENABLE cycle");
            end
            ...
            `vmm_trace(log, {"Responded to transaction...\n",
                            tr.psdisplay("   ")});
            this.notify.indicate(RESPONSE, tr);
      end
      endtask: main
endclass: apb_slave

`endif
```

The configuration descriptor is saved in a local class property in the transactor to avoid the configuration from being modified by directly modifying the configuration descriptor. This occurrence of a direct

modification would not be visible to the slave transactor. Because of the simplicity of the slave functionality and its configuration, it would not have any ill effects, but transactors implementing more complex protocols may very well need to be told that they are being reconfigured. To that effect, a **`reconfigure()`** method is provided (Rule 4-107).

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"

class apb_slave_cfg;
   rand bit [31:0] start_addr = 32'h0000_0000;
   rand bit [31:0] end_addr   = 32'hFFFF_FFFF;

   constraint apb_slave_cfg_valid {
      end_addr >= start_addr;
   }
   ...
endclass: apb_slave_cfg
...
class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   local apb_slave_cfg cfg;
   ...
   typedef enum {RESPONSE} notifications_e;
   ...
   local bit [31:0] ram[*];

   function new(string               name,
                int unsigned         stream_id,
                virtual apb_if.slave sigs,
                apb_slave_cfg        cfg = null,
                ...);
      super.new("APB Slave", name, stream_id);
      this.sigs = sigs;

      if (cfg == null) cfg = new;
      this.cfg = cfg;
      ...
```

```
   this.notify.configure(RESPONSE);
endfunction: new

virtual function void reconfigure(apb_slave_cfg cfg);
   this.cfg = cfg;
endfunction: reconfigure
...
virtual protected task main();
   super.main();
   forever begin
      apb_rw tr;

      this.sigs.sck.prdata <= 'z;
      this.wait_if_stopped();

  `vmm_trace(log, "Waiting for start of transaction...");
      ...
      // Wait for a SETUP cycle
      do @ (this.sigs.sck);
      while (this.sigs.sck.psel !== 1'b1 ||
             this.sigs.sck.penable !== 1'b0 ||
           this.sigs.sck.paddr < this.cfg.start_addr ||
             this.sigs.sck.paddr > this.cfg.end_addr);
      tr = new;

      tr.kind = (this.sigs.sck.pwrite) ?
                   apb_rw::WRITE : apb_rw::READ;
      tr.addr = this.sigs.sck.paddr;

     `vmm_trace(log, {"Responding to transaction...\n",
                      tr.psdisplay("   ")});
      if (tr.kind == apb_rw::READ) begin
         if (!this.ram.exists(tr.addr)) tr.data = 'x;
         else tr.data = this.ram[tr.addr];
         ...
         this.sigs.sck.prdata <= tr.data;
         @ (this.sigs.sck);
      end
      else begin
         @ (this.sigs.sck);
         tr.data = this.sigs.sck.pwdata;
         ...
         this.ram[tr.addr] = tr.data;
      end
      if (this.sigs.sck.penable !== 1'b1) begin
        `vmm_error(this.log, "APB protocol violation:\n
```

```
                    SETUP cycle not followed by ENABLE cycle");
          end
          ...
          `vmm_trace(log, {"Responded to transaction...\n",
                           tr.psdisplay("   ")});
          this.notify.indicate(RESPONSE, tr);
   end
   endtask: main
endclass: apb_slave

`endif
```

It is not necessary to derive the configuration descriptor from **vmm_data** as it is not a transaction descriptor and is not called upon to flow through **vmm_channels** or be attached to **vmm_notify** indications. Nevertheless, the **psdisplay()** method should be provided, with a signature identical to **vmm::psdisplay()** to make it easier to report the current transactor configuration during simulation.

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"

class apb_slave_cfg;
   rand bit [31:0] start_addr = 32'h0000_0000;
   rand bit [31:0] end_addr   = 32'hFFFF_FFFF;

   constraint apb_slave_cfg_valid {
      end_addr >= start_addr;
   }

   virtual function string psdisplay(string prefix = "");
     $sformat(psdisplay, "%sAPB Slave Config: [`h%h-`h%h]",
              prefix, this.start_addr, this.end_addr);
   endfunction: psdisplay
endclass: apb_slave_cfg
...
class apb_slave extends vmm_xactor;
```

```
virtual apb_if.slave sigs;
local apb_slave_cfg cfg;
...
typedef enum {RESPONSE} notifications_e;
...
local bit [31:0] ram[*];

function new(string                name,
             int unsigned          stream_id,
             virtual apb_if.slave sigs,
             apb_slave_cfg         cfg = null,
             ...);
   super.new("APB Slave", name, stream_id);
   this.sigs = sigs;

   if (cfg == null) cfg = new;
   this.cfg = cfg;
   ...
   this.notify.configure(RESPONSE);
endfunction: new

virtual function void reconfigure(apb_slave_cfg cfg);
   this.cfg = cfg;
endfunction: reconfigure
...
virtual protected task main();
   super.main();
   forever begin
      apb_rw tr;

      this.sigs.sck.prdata <= 'z;
      this.wait_if_stopped();

  `vmm_trace(log, "Waiting for start of transaction...");
      ...
      // Wait for a SETUP cycle
      do @ (this.sigs.sck);
      while (this.sigs.sck.psel !== 1'b1 ||
             this.sigs.sck.penable !== 1'b0 ||
           this.sigs.sck.paddr < this.cfg.start_addr ||
             this.sigs.sck.paddr > this.cfg.end_addr);
      tr = new;

      tr.kind = (this.sigs.sck.pwrite) ?
                  apb_rw::WRITE : apb_rw::READ;
      tr.addr = this.sigs.sck.paddr;
```

```
            `vmm_trace(log, {"Responding to transaction...\n",
                             tr.psdisplay("   ")});
            if (tr.kind == apb_rw::READ) begin
               if (!this.ram.exists(tr.addr)) tr.data = 'x;
               else tr.data = this.ram[tr.addr];
               ...
               this.sigs.sck.prdata <= tr.data;
               @ (this.sigs.sck);
            end
            else begin
               @ (this.sigs.sck);
               tr.data = this.sigs.sck.pwdata;
               ...
               this.ram[tr.addr] = tr.data;
            end
            if (this.sigs.sck.penable !== 1'b1) begin
              `vmm_error(this.log, "APB protocol violation:\n
                    SETUP cycle not followed by ENABLE cycle");
            end
            ...
            `vmm_trace(log, {"Responded to transaction...\n",
                             tr.psdisplay("   ")});
            this.notify.indicate(RESPONSE, tr);
      end
      endtask: main
endclass: apb_slave

`endif
```

Step 10: Backdoor Interface

To help predict or control the response of the slave, it is useful to have a procedural interface to the content of the RAM. This interface allows querying the RAM at a specific address as well as setting values at specific addresses. Optionally, a procedure could be provided to delete the data value at a specified address to reclaim its memory from the associated array.

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"

class apb_slave_cfg;
   rand bit [31:0] start_addr = 32'h0000_0000;
   rand bit [31:0] end_addr   = 32'hFFFF_FFFF;

   constraint apb_slave_cfg_valid {
      end_addr >= start_addr;
   }

   virtual function string psdisplay(string prefix = "");
     $sformat(psdisplay, "%sAPB Slave Config: [`h%h-`h%h]",
              prefix, this.start_addr, this.end_addr);
   endfunction: psdisplay
endclass: apb_slave_cfg
...
class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   local apb_slave_cfg cfg;
   ...
   typedef enum {RESPONSE} notifications_e;
   ...
   local bit [31:0] ram[*];

   function new(string               name,
                int unsigned         stream_id,
                virtual apb_if.slave sigs,
                apb_slave_cfg        cfg = null,
                ...);
      super.new("APB Slave", name, stream_id);
      this.sigs = sigs;

      if (cfg == null) cfg = new;
      this.cfg = cfg;
      ...
      this.notify.configure(RESPONSE);
   endfunction: new

   virtual function void reconfigure(apb_slave_cfg cfg);
      this.cfg = cfg;
   endfunction: reconfigure
```

```
virtual function void poke(bit [31:0] addr,
                           bit [31:0] data);
   if (addr < this.cfg.start_addr ||
       addr > this.cfg.end_addr) begin
      `vmm_error(this.log, "Out-of-range poke");
      return;
   end
   this.ram[addr] = data;
endfunction: poke

virtual function bit [31:0] peek(bit [31:0] addr);
   if (addr < this.cfg.start_addr ||
       addr > this.cfg.end_addr) begin
      `vmm_error(this.log, "Out-of-range peek");
      return 'x;
   end
  return (this.ram.exists(addr)) ? this.ram[addr] : 'x;
endfunction: peek
...
virtual protected task main();
   super.main();
   forever begin
      apb_rw tr;

      this.sigs.sck.prdata <= 'z;
      this.wait_if_stopped();

  `vmm_trace(log, "Waiting for start of transaction...");
      ...
      // Wait for a SETUP cycle
      do @ (this.sigs.sck);
      while (this.sigs.sck.psel !== 1'b1 ||
             this.sigs.sck.penable !== 1'b0 ||
           this.sigs.sck.paddr < this.cfg.start_addr ||
             this.sigs.sck.paddr > this.cfg.end_addr);
      tr = new;

      tr.kind = (this.sigs.sck.pwrite) ?
                   apb_rw::WRITE : apb_rw::READ;
      tr.addr = this.sigs.sck.paddr;

     `vmm_trace(log, {"Responding to transaction...\n",
                      tr.psdisplay("   ")});
      if (tr.kind == apb_rw::READ) begin
         if (!this.ram.exists(tr.addr)) tr.data = 'x;
```

```
            else tr.data = this.ram[tr.addr];
            ...
            this.sigs.sck.prdata <= tr.data;
            @ (this.sigs.sck);
         end
         else begin
            @ (this.sigs.sck);
            tr.data = this.sigs.sck.pwdata;
            ...
            this.ram[tr.addr] = tr.data;
         end
         if (this.sigs.sck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation:\n
                  SETUP cycle not followed by ENABLE cycle");
         end
         ...
         `vmm_trace(log, {"Responded to transaction...\n",
                          tr.psdisplay("   ")});
         this.notify.indicate(RESPONSE, tr);
   end
   endtask: main
endclass: apb_slave

`endif
```

Step 11: Extension Points

This slave transactor should behave correctly by default. Although the correct behavior is desired most of the time, functional verification also entails the injection of errors to verify that the design reacts properly to those errors. In this simple protocol, the only errors that could be injected are wrong addresses and wrong data. A more complex protocol would probably have parity bits that could be corrupted or responses that could be delayed.

A callback method allows a user to extend the behavior of a slave transactor without having to modify the transactor itself. Callback methods should be provided before the response to a transaction is sent back (Recommendation 4-156, 4-158) and after the transaction

response has completed (Recommendation 4-155). The “pre-response” callback method allows errors to be injected and delays to be inserted. The “post-response” callback method allows delays to be inserted and the result of the transaction to be recorded in a functional coverage model or checked against an expected response.

The callback methods are first defined as **`virtual void functions`** (Rule 4-160) in a callback façade class extended from the **`vmm_xactor_callbacks`** base class (Rule 4-159).

Next, the appropriate callback method needs to be invoked at the appropriate point in the execution of the monitor, using the **`` `vmm_callback() ``** macro (Rule 4-163).

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"

class apb_slave_cfg;
   rand bit [31:0] start_addr = 32'h0000_0000;
   rand bit [31:0] end_addr   = 32'hFFFF_FFFF;

   constraint apb_slave_cfg_valid {
      end_addr >= start_addr;
   }

   virtual function string psdisplay(string prefix = "");
     $sformat(psdisplay, "%sAPB Slave Config: [`h%h-`h%h]",
              prefix, this.start_addr, this.end_addr);
   endfunction: psdisplay
endclass: apb_slave_cfg

typedef class apb_slave;
class apb_slave_cbs extends vmm_xactor_callbacks;
   virtual function void pre_response(apb_slave xact,
                                      apb_rw    cycle);
```

```
   endfunction: pre_response

   virtual function void post_response(apb_slave xactor,
                                       apb_rw    cycle);
   endfunction: post_response
endclass: apb_slave_cbs

class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   local apb_slave_cfg cfg;
   ...
   typedef enum {RESPONSE} notifications_e;
   ...
   local bit [31:0] ram[*];

   function new(string               name,
                int unsigned         stream_id,
                virtual apb_if.slave sigs,
                apb_slave_cfg        cfg = null,
                ...);
      super.new("APB Slave", name, stream_id);
      this.sigs = sigs;

      if (cfg == null) cfg = new;
      this.cfg = cfg;
      ...
      this.notify.configure(RESPONSE);
   endfunction: new

   virtual function void reconfigure(apb_slave_cfg cfg);
      this.cfg = cfg;
   endfunction: reconfigure

   virtual function void poke(bit [31:0] addr,
                              bit [31:0] data);
      if (addr < this.cfg.start_addr ||
          addr > this.cfg.end_addr) begin
         `vmm_error(this.log, "Out-of-range poke");
         return;
      end
      this.ram[addr] = data;
   endfunction: poke

   virtual function bit [31:0] peek(bit [31:0] addr);
      if (addr < this.cfg.start_addr ||
          addr > this.cfg.end_addr) begin
```

```
      `vmm_error(this.log, "Out-of-range peek");
      return 'x;
   end
  return (this.ram.exists(addr)) ? this.ram[addr] : 'x;
endfunction: peek
...
virtual protected task main();
   super.main();
   forever begin
      apb_rw tr;

      this.sigs.sck.prdata <= 'z;
      this.wait_if_stopped();

  `vmm_trace(log, "Waiting for start of transaction...");
      ...
      // Wait for a SETUP cycle
      do @ (this.sigs.sck);
      while (this.sigs.sck.psel !== 1'b1 ||
             this.sigs.sck.penable !== 1'b0 ||
           this.sigs.sck.paddr < this.cfg.start_addr ||
             this.sigs.sck.paddr > this.cfg.end_addr);
      tr = new;

      tr.kind = (this.sigs.sck.pwrite) ?
                   apb_rw::WRITE : apb_rw::READ;
      tr.addr = this.sigs.sck.paddr;

     `vmm_trace(log, {"Responding to transaction...\n",
                      tr.psdisplay("   ")});
      if (tr.kind == apb_rw::READ) begin
         if (!this.ram.exists(tr.addr)) tr.data = 'x;
         else tr.data = this.ram[tr.addr];

        `vmm_callback(apb_slave_cbs, pre_response(this,
                      tr));

         this.sigs.sck.prdata <= tr.data;
         @ (this.sigs.sck);
      end
      else begin
         @ (this.sigs.sck);
         tr.data = this.sigs.sck.pwdata;

        `vmm_callback(apb_slave_cbs, pre_response(this,
                      tr));
```

```
            this.ram[tr.addr] = tr.data;
          end
          if (this.sigs.sck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation:\n
                 SETUP cycle not followed by ENABLE cycle");
          end

          `vmm_callback(apb_slave_cbs, post_response(this,
                        tr));

          `vmm_trace(log, {"Responded to transaction...\n",
                           tr.psdisplay("   ")});
          this.notify.indicate(RESPONSE, tr);
    end
    endtask: main
endclass: apb_slave

`endif
```

Step 12: Transaction Response

The slave transactor is currently hard-coded to respond like a RAM. That is fine in most cases, but it makes the transactor unsuitable for providing a different response or to use it to write transaction-level models of slave devices. A transaction-level interface can be used to request a response to an APB transaction from a higher-level transactor or model (Rule 4-111). A transaction-level interface carrying APB transaction descriptor is defined by using the **`vmm_channel** macro.

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
```

```
static vmm_log log = new("apb_rw", "class");

rand enum {READ, WRITE} kind;
rand bit   [31:0] addr;
rand logic [31:0] data;

function new();
   super.new(this.log);
endfunction: new

virtual function vmm_data allocate();
   apb_rw tr = new;
   return tr;
endfunction: allocate

virtual function vmm_data copy(vmm_data to = null);
   apb_rw tr;

   if (to == null) tr = new;
   else if (!$cast(tr, to)) begin
      `vmm_fatal(log, "Cannot copy into non-apb_rw \n
                 instance");
      return null;
   end

   super.copy_data(tr);
   tr.kind = this.kind;
   tr.addr = this.addr;
   tr.data = this.data;

   return tr;
endfunction: copy

virtual function string psdisplay(string prefix = "");
   $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
            this.kind.name(), this.addr, this.data);
endfunction: psdisplay

virtual function bit is_valid(bit silent = 1,
                              int kind   = -1);
   return 1;
endfunction: is_valid

virtual function bit compare(input  vmm_data to,
                             output string   diff,
                            input  int      kind = -1);
```

```
      apb_rw tr;

      if (to == null) begin
       `vmm_fatal(log, "Cannot compare to NULL reference");
        return 0;
      end
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot compare against \n
                     non-apb_rw  instance");
         return 0;
      end

      if (this.kind != tr.kind) begin
       $sformat(diff, "Kind %s !== %s", this.kind, tr.kind);
         return 0;
      end

      if (this.addr !== tr.addr) begin
         $sformat(diff, "Addr 0x%h !== 0x%h", this.addr,
                  tr.addr);
         return 0;
      end

      if (this.data !== tr.data) begin
         $sformat(diff, "Data 0x%h !== 0x%h", this.data,
                  tr.data);
         return 0;
      end

      return 1;
   endfunction: compare

endclass: apb_rw

`vmm_channel(apb_rw)

`endif
```

The response request channel is specified as an optional argument to the constructor (Rule 4-113) and stored in a local property (Rule 4-112). If no channel is specified, the response defaults to the RAM response.

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"

class apb_slave_cfg;
   rand bit [31:0] start_addr = 32'h0000_0000;
   rand bit [31:0] end_addr   = 32'hFFFF_FFFF;

   constraint apb_slave_cfg_valid {
      end_addr >= start_addr;
   }

   virtual function string psdisplay(string prefix = "");
     $sformat(psdisplay, "%sAPB Slave Config: [`h%h-`h%h]",
              prefix, this.start_addr, this.end_addr);
   endfunction: psdisplay
endclass: apb_slave_cfg

typedef class apb_slave;
class apb_slave_cbs extends vmm_xactor_callbacks;
   virtual function void pre_response(apb_slave xact,
                                      apb_rw    cycle);
   endfunction: pre_response

   virtual function void post_response(apb_slave xactor,
                                       apb_rw    cycle);
   endfunction: post_response
endclass: apb_slave_cbs

class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   local apb_slave_cfg cfg;
   apb_rw_channel resp_chan;
   typedef enum {RESPONSE} notifications_e;
   ...
   local bit [31:0] ram[*];

   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.slave  sigs,
                apb_slave_cfg         cfg = null,
                apb_rw_channel        resp_chan = null,
                ...);
```

```
   super.new("APB Slave", name, stream_id);
   this.sigs = sigs;

   if (cfg == null) cfg = new;
   this.cfg = cfg;
   this.resp_chan = resp_chan;
   ...
   this.notify.configure(RESPONSE);
endfunction: new

virtual function void reconfigure(apb_slave_cfg cfg);
   this.cfg = cfg;
endfunction: reconfigure

virtual function void poke(bit [31:0] addr,
                           bit [31:0] data);
   if (addr < this.cfg.start_addr ||
       addr > this.cfg.end_addr) begin
      `vmm_error(this.log, "Out-of-range poke");
      return;
   end
   this.ram[addr] = data;
endfunction: poke

virtual function bit [31:0] peek(bit [31:0] addr);
   if (addr < this.cfg.start_addr ||
       addr > this.cfg.end_addr) begin
      `vmm_error(this.log, "Out-of-range peek");
      return 'x;
   end
  return (this.ram.exists(addr)) ? this.ram[addr] : 'x;
endfunction: peek
...
virtual protected task main();
   super.main();
   forever begin
      apb_rw tr;

      this.sigs.sck.prdata <= 'z;
      this.wait_if_stopped();

  `vmm_trace(log, "Waiting for start of transaction...");
      ...
      // Wait for a SETUP cycle
      do @ (this.sigs.sck);
      while (this.sigs.sck.psel !== 1'b1 ||
```

```
        this.sigs.sck.penable !== 1'b0 ||
      this.sigs.sck.paddr < this.cfg.start_addr ||
        this.sigs.sck.paddr > this.cfg.end_addr);
tr = new;

tr.kind = (this.sigs.sck.pwrite) ?
             apb_rw::WRITE : apb_rw::READ;
tr.addr = this.sigs.sck.paddr;

`vmm_trace(log, {"Responding to transaction...\n",
                 tr.psdisplay("   ")});
if (tr.kind == apb_rw::READ) begin
   if (this.resp_chan == null) begin
     if (!this.ram.exists(tr.addr)) tr.data = 'x;
      else tr.data = this.ram[tr.addr];
   end
   else begin
      ...
      this.resp_chan.put(tr);
      ...
   end

  `vmm_callback(apb_slave_cbs, pre_response(this,
                tr));

   this.sigs.sck.prdata <= tr.data;
   @ (this.sigs.sck);
end
else begin
   @ (this.sigs.sck);
   tr.data = this.sigs.sck.pwdata;

  `vmm_callback(apb_slave_cbs, pre_response(this,
                tr));

   if (this.resp_chan == null)
      this.ram[tr.addr] = tr.data;
   else this.resp_chan.sneak(tr);
end
if (this.sigs.sck.penable !== 1'b1) begin
  `vmm_error(this.log, "APB protocol violation:\n
       SETUP cycle not followed by ENABLE cycle");
end

`vmm_callback(apb_slave_cbs, post_response(this,
              tr));
```

```
            `vmm_trace(log, {"Responded to transaction...\n",
                             tr.psdisplay("   ")});
            this.notify.indicate(RESPONSE, tr);
      end
      endtask: main
endclass: apb_slave

`endif
```

When requesting the response to a WRITE transaction, the slave transactor uses the **`vmm_channel::sneak()`** method to put the transaction descriptor on the response request channel. Because there is no feedback on WRITE cycles in the APB protocol, the slave transactor simply notifies the response transactor of the WRITE transaction and does not need to wait for a response. When requesting the response to a READ transaction, a blocking response model is expected from the response transactor. When the **`vmm_channel::put()`** method returns, the transaction descriptor contains the data to return to the DUT. It is assumed that consumer at the other end of `resp_chan` provides the read data and ensures that **`put()`** method blocks till the read data is provided. It is a good idea to ensure that the response is provided in a timely fashion by the response transactor by forking a timer thread.

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"

class apb_slave_cfg;
   rand bit [31:0] start_addr = 32'h0000_0000;
   rand bit [31:0] end_addr   = 32'hFFFF_FFFF;

   constraint apb_slave_cfg_valid {

      end_addr >= start_addr;
```

```
   }

   virtual function string psdisplay(string prefix = "");
     $sformat(psdisplay, "%sAPB Slave Config: [`h%h-`h%h]",
              prefix, this.start_addr, this.end_addr);
   endfunction: psdisplay
endclass: apb_slave_cfg

typedef class apb_slave;
class apb_slave_cbs extends vmm_xactor_callbacks;
   virtual function void pre_response(apb_slave xact,
                                      apb_rw    cycle);
   endfunction: pre_response

   virtual function void post_response(apb_slave xactor,
                                       apb_rw    cycle);
   endfunction: post_response
endclass: apb_slave_cbs

class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   local apb_slave_cfg cfg;
   apb_rw_channel resp_chan;
   typedef enum {RESPONSE} notifications_e;
   ...
   local bit [31:0] ram[*];

   function new(string               name,
                int unsigned         stream_id,
                virtual apb_if.slave sigs,
                apb_slave_cfg        cfg = null,
                apb_rw_channel       resp_chan = null,
                ...);
      super.new("APB Slave", name, stream_id);
      this.sigs = sigs;

      if (cfg == null) cfg = new;
      this.cfg = cfg;
      this.resp_chan = resp_chan;
      ...
      this.notify.configure(RESPONSE);
   endfunction: new

   virtual function void reconfigure(apb_slave_cfg cfg);
      this.cfg = cfg;
   endfunction: reconfigure
```

```
virtual function void poke(bit [31:0] addr,
                           bit [31:0] data);
   if (addr < this.cfg.start_addr ||
       addr > this.cfg.end_addr) begin
      `vmm_error(this.log, "Out-of-range poke");
      return;
   end
   this.ram[addr] = data;
endfunction: poke

virtual function bit [31:0] peek(bit [31:0] addr);
   if (addr < this.cfg.start_addr ||
       addr > this.cfg.end_addr) begin
      `vmm_error(this.log, "Out-of-range peek");
      return 'x;
   end
  return (this.ram.exists(addr)) ? this.ram[addr] : 'x;
endfunction: peek
...
virtual protected task main();
   super.main();
   forever begin
      apb_rw tr;

      this.sigs.sck.prdata <= 'z;
      this.wait_if_stopped();

  `vmm_trace(log, "Waiting for start of transaction...");
      ...
      // Wait for a SETUP cycle
      do @ (this.sigs.sck);
      while (this.sigs.sck.psel !== 1'b1 ||
             this.sigs.sck.penable !== 1'b0 ||
           this.sigs.sck.paddr < this.cfg.start_addr ||
             this.sigs.sck.paddr > this.cfg.end_addr);
      tr = new;

      tr.kind = (this.sigs.sck.pwrite) ?
                  apb_rw::WRITE : apb_rw::READ;
      tr.addr = this.sigs.sck.paddr;

     `vmm_trace(log, {"Responding to transaction...\n",
                      tr.psdisplay("   ")});
      if (tr.kind == apb_rw::READ) begin
         if (this.resp_chan == null) begin
```

```
      if (!this.ram.exists(tr.addr)) tr.data = 'x;
       else tr.data = this.ram[tr.addr];
   end
   else begin
      bit abort = 0;
      fork
         begin
            fork
              begin
                 @ (this.sigs.sck);
       `vmm_error(this.log, "No response in time");
                 abort = 1;
              end
              this.resp_chan.put(tr);
            join_any
            disable fork;
         end
      join
      if (abort) continue;
   end

  `vmm_callback(apb_slave_cbs, pre_response(this,
                tr));

   this.sigs.sck.prdata <= tr.data;
   @ (this.sigs.sck);
end
else begin
   @ (this.sigs.sck);
   tr.data = this.sigs.sck.pwdata;

  `vmm_callback(apb_slave_cbs, pre_response(this,
                tr));

   if (this.resp_chan == null)
      this.ram[tr.addr] = tr.data;
   else this.resp_chan.sneak(tr);
end
if (this.sigs.sck.penable !== 1'b1) begin
  `vmm_error(this.log, "APB protocol violation:\n
       SETUP cycle not followed by ENABLE cycle");
end

`vmm_callback(apb_slave_cbs, post_response(this,
              tr));
```

```
         `vmm_trace(log, {"Responded to transaction...\n",
                          tr.psdisplay("   ")});
         this.notify.indicate(RESPONSE, tr);
   end
   endtask: main
endclass: apb_slave

`endif
```

When the transactor is reset, the response channel must be flushed and the output signals must be driven to their idle state. This behavior is accomplished in the extension of the **`vmm_xactor::reset_xactor()`** method (Table A-8).

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"

class apb_slave_cfg;
   rand bit [31:0] start_addr = 32'h0000_0000;
   rand bit [31:0] end_addr   = 32'hFFFF_FFFF;

   constraint apb_slave_cfg_valid {
      end_addr >= start_addr;
   }

   virtual function string psdisplay(string prefix = "");
     $sformat(psdisplay, "%sAPB Slave Config: [`h%h-`h%h]",
              prefix, this.start_addr, this.end_addr);
   endfunction: psdisplay
endclass: apb_slave_cfg

typedef class apb_slave;
class apb_slave_cbs extends vmm_xactor_callbacks;
   virtual function void pre_response(apb_slave xact,
                                      apb_rw    cycle);
   endfunction: pre_response

   virtual function void post_response(apb_slave xactor,
                                       apb_rw    cycle);
```

```
   endfunction: post_response
endclass: apb_slave_cbs

class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   local apb_slave_cfg cfg;
   apb_rw_channel resp_chan;
   typedef enum {RESPONSE} notifications_e;
   ...
   local bit [31:0] ram[*];

   function new(string               name,
                int unsigned         stream_id,
                virtual apb_if.slave sigs,
                apb_slave_cfg        cfg = null,
                apb_rw_channel       resp_chan = null,
                ...);
      super.new("APB Slave", name, stream_id);
      this.sigs = sigs;

      if (cfg == null) cfg = new;
      this.cfg = cfg;
      this.resp_chan = resp_chan;
      ...
      this.notify.configure(RESPONSE);
   endfunction: new

   virtual function void reconfigure(apb_slave_cfg cfg);
      this.cfg = cfg;
   endfunction: reconfigure

   virtual function void poke(bit [31:0] addr,
                              bit [31:0] data);
      if (addr < this.cfg.start_addr ||
          addr > this.cfg.end_addr) begin
         `vmm_error(this.log, "Out-of-range poke");
         return;
      end
      this.ram[addr] = data;
   endfunction: poke

   virtual function bit [31:0] peek(bit [31:0] addr);
      if (addr < this.cfg.start_addr ||
          addr > this.cfg.end_addr) begin
         `vmm_error(this.log, "Out-of-range peek");
         return 'x;
```

```
   end
   return (this.ram.exists(addr)) ? this.ram[addr] : 'x;
endfunction: peek

virtual function void reset_xactor(reset_e rst_typ =
                                   SOFT_RST);
   super.reset_xactor(rst_typ);
   this.resp_chan.flush();
   this.sigs.sck.prdata <= 'z;
endfunction: reset_xactor

virtual protected task main();
   super.main();
   forever begin
      apb_rw tr;

      this.sigs.sck.prdata <= 'z;
      this.wait_if_stopped();

  `vmm_trace(log, "Waiting for start of transaction...");

      // Wait for a SETUP cycle
      do @ (this.sigs.sck);
      while (this.sigs.sck.psel !== 1'b1 ||
             this.sigs.sck.penable !== 1'b0 ||
           this.sigs.sck.paddr < this.cfg.start_addr ||
             this.sigs.sck.paddr > this.cfg.end_addr);
      tr = new;

      tr.kind = (this.sigs.sck.pwrite) ?
                  apb_rw::WRITE : apb_rw::READ;
      tr.addr = this.sigs.sck.paddr;

      `vmm_trace(log, {"Responding to transaction...\n",
                       tr.psdisplay("   ")});
      if (tr.kind == apb_rw::READ) begin
         if (this.resp_chan == null) begin
           if (!this.ram.exists(tr.addr)) tr.data = 'x;
            else tr.data = this.ram[tr.addr];
         end
         else begin
            bit abort = 0;
            fork
               begin
                  fork
                    begin
```

```
                        @ (this.sigs.sck);
                    `vmm_error(this.log, "No response in time");
                        abort = 1;
                      end
                      this.resp_chan.put(tr);
                    join_any
                    disable fork;
                  end
                join
                if (abort) continue;
              end

              `vmm_callback(apb_slave_cbs, pre_response(this
                        tr));

              this.sigs.sck.prdata <= tr.data;
              @ (this.sigs.sck);
            end
            else begin
              @ (this.sigs.sck);
              tr.data = this.sigs.sck.pwdata;

              `vmm_callback(apb_slave_cbs, pre_response(this,
                        tr));

              if (this.resp_chan == null)
                this.ram[tr.addr] = tr.data;
              else this.resp_chan.sneak(tr);
            end
            if (this.sigs.sck.penable !== 1'b1) begin
              `vmm_error(this.log, "APB protocol violation:\n
                   SETUP cycle not followed by ENABLE cycle");
            end

            `vmm_callback(apb_slave_cbs, post_response(this,
                        tr));

            `vmm_trace(log, {"Responded to transaction...\n",
                        tr.psdisplay("   ")});
            this.notify.indicate(RESPONSE, tr);
        end
        endtask: main
endclass: apb_slave

`endif
```

It may be useful for the higher-level functions using the transactions reported by the slave transactor to know when the transaction was started and when it ended. These transaction endpoints are recorded in the transaction descriptor itself by the slave transactor indicating the **`vmm_data::STARTED`** and **`vmm_data::ENDED`** notifications (Rule 4-142).

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"

class apb_slave_cfg;
   rand bit [31:0] start_addr = 32'h0000_0000;
   rand bit [31:0] end_addr   = 32'hFFFF_FFFF;

   constraint apb_slave_cfg_valid {
      end_addr >= start_addr;
   }

   virtual function string psdisplay(string prefix = "");
     $sformat(psdisplay, "%sAPB Slave Config: [`h%h-`h%h]",
              prefix, this.start_addr, this.end_addr);
   endfunction: psdisplay
endclass: apb_slave_cfg

typedef class apb_slave;
class apb_slave_cbs extends vmm_xactor_callbacks;
   virtual function void pre_response(apb_slave xact,
                                      apb_rw    cycle);
   endfunction: pre_response

   virtual function void post_response(apb_slave xactor,
                                       apb_rw    cycle);
   endfunction: post_response
endclass: apb_slave_cbs

class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   local apb_slave_cfg cfg;
   apb_rw_channel resp_chan;
```

```
typedef enum {RESPONSE} notifications_e;
...
local bit [31:0] ram[*];

function new(string               name,
             int unsigned         stream_id,
             virtual apb_if.slave sigs,
             apb_slave_cfg        cfg = null,
             apb_rw_channel       resp_chan = null,
             ...);
   super.new("APB Slave", name, stream_id);
   this.sigs = sigs;

   if (cfg == null) cfg = new;
   this.cfg = cfg;
   this.resp_chan = resp_chan;
   ...
   this.notify.configure(RESPONSE);
endfunction: new

virtual function void reconfigure(apb_slave_cfg cfg);
   this.cfg = cfg;
endfunction: reconfigure

virtual function void poke(bit [31:0] addr,
                           bit [31:0] data);
   if (addr < this.cfg.start_addr ||
       addr > this.cfg.end_addr) begin
      `vmm_error(this.log, "Out-of-range poke");
      return;
   end
   this.ram[addr] = data;
endfunction: poke

virtual function bit [31:0] peek(bit [31:0] addr);
   if (addr < this.cfg.start_addr ||
       addr > this.cfg.end_addr) begin
      `vmm_error(this.log, "Out-of-range peek");
      return 'x;
   end
  return (this.ram.exists(addr)) ? this.ram[addr] : 'x;
endfunction: peek

virtual function void reset_xactor(reset_e rst_typ =
                                   SOFT_RST);
   super.reset_xactor(rst_typ);
```

```
   this.resp_chan.flush();
   this.sigs.sck.prdata <= 'z;
endfunction: reset_xactor

virtual protected task main();
   super.main();
   forever begin
      apb_rw tr;

      this.sigs.sck.prdata <= 'z;
      this.wait_if_stopped();

  `vmm_trace(log, "Waiting for start of transaction...");

      // Wait for a SETUP cycle
      do @ (this.sigs.sck);
      while (this.sigs.sck.psel !== 1'b1 ||
             this.sigs.sck.penable !== 1'b0 ||
            this.sigs.sck.paddr < this.cfg.start_addr ||
             this.sigs.sck.paddr > this.cfg.end_addr);
      tr = new;

      tr.notify.indicate(vmm_data::STARTED);
      tr.kind = (this.sigs.sck.pwrite) ?
                   apb_rw::WRITE : apb_rw::READ;
      tr.addr = this.sigs.sck.paddr;

     `vmm_trace(log, {"Responding to transaction...\n",
                      tr.psdisplay("   ")});
      if (tr.kind == apb_rw::READ) begin
         if (this.resp_chan == null) begin
           if (!this.ram.exists(tr.addr)) tr.data = 'x;
            else tr.data = this.ram[tr.addr];
         end
         else begin
            bit abort = 0;
            fork
               begin
                  fork
                    begin
                       @ (this.sigs.sck);
                   `vmm_error(this.log, "No response in time");
                       abort = 1;
                    end
                    this.resp_chan.put(tr);
                  join_any
```

```
                  disable fork;
               end
            join
            if (abort) continue;
         end

         `vmm_callback(apb_slave_cbs, pre_response(this,
                       tr));

         this.sigs.sck.prdata <= tr.data;
         @ (this.sigs.sck);
      end
      else begin
         @ (this.sigs.sck);
         tr.data = this.sigs.sck.pwdata;

         `vmm_callback(apb_slave_cbs, pre_response(this,
                       tr));

         if (this.resp_chan == null)
            this.ram[tr.addr] = tr.data;
         else this.resp_chan.sneak(tr);
      end
      if (this.sigs.sck.penable !== 1'b1) begin
         `vmm_error(this.log, "APB protocol violation:\n
              SETUP cycle not followed by ENABLE cycle");
      end
      tr.notify.indicate(vmm_data::ENDED);

      `vmm_callback(apb_slave_cbs, post_response(this,
                    tr));

      `vmm_trace(log, {"Responded to transaction...\n",
                       tr.psdisplay("   ")});
      this.notify.indicate(RESPONSE, tr);
   end
   endtask: main
endclass: apb_slave

`endif
```

Step 13: Random Responses

To promote the use of random stimulus and make the same transaction descriptor usable in transaction generators, all public properties in a transaction descriptor should be declared as *rand* (Rules 4-59, 4-60, 4-62). It is also a good idea to pre-define random transaction generators whenever transaction descriptors are defined. It is a simple matter of using the **`vmm_atomic_gen()** and **`vmm_scenario_gen()** macros (Recommendation 5-23, 5-24) in the transaction descriptor file.

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   rand enum {READ, WRITE} kind;
   rand bit   [31:0] addr;
   rand logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new

   virtual function vmm_data allocate();
      apb_rw tr = new;
      return tr;
   endfunction: allocate

   virtual function vmm_data copy(vmm_data to = null);
      apb_rw tr;

      if (to == null) tr = new;
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot copy into non-apb_rw \n
                     instance");
```

```
      return null;

   end

   super.copy_data(tr);
   tr.kind = this.kind;
   tr.addr = this.addr;
   tr.data = this.data;

   return tr;
endfunction: copy

virtual function string psdisplay(string prefix = "");
  $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
           this.kind.name(), this.addr, this.data);
endfunction: psdisplay

virtual function bit is_valid(bit silent = 1,
                              int kind   = -1);
   return 1;
endfunction: is_valid

virtual function bit compare(input  vmm_data to,
                             output string   diff,
                            input  int      kind = -1);
   apb_rw tr;

   if (to == null) begin
    `vmm_fatal(log, "Cannot compare to NULL reference");
      return 0;
   end
   else if (!$cast(tr, to)) begin
    `vmm_fatal(log, "Cannot compare against non-apb_rw \n
              instance");
      return 0;
   end

   if (this.kind != tr.kind) begin
    $sformat(diff, "Kind %s !== %s", this.kind, tr.kind);
      return 0;
   end

   if (this.addr !== tr.addr) begin
      $sformat(diff, "Addr 0x%h !== 0x%h", this.addr,
               tr.addr);
      return 0;
```

```
      end

      if (this.data !== tr.data) begin
         $sformat(diff, "Data 0x%h !== 0x%h", this.data,
                  tr.data);
         return 0;
      end

      return 1;
   endfunction: compare

endclass: apb_rw

`vmm_channel(apb_rw)
`vmm_atomic_gen(apb_rw, "APB Bus Cycle")
`vmm_scenario_gen(apb_rw, "APB Bus Cycle")

`endif
```

Random stimulus can also be used in generating the response to READ cycles. This effect can be accomplished by turning off the **`rand_mode`** for the **`apb_rw::kind`** and **`apb_rw::addr`** properties then randomizing the remaining properties.

File: Command_Slave_Xactor/tb_env.sv

```
`ifndef TB_ENV__SV
`define TB_ENV__SV

`include "vmm.sv"
`include "apb_slave.sv"

class tb_env extends vmm_env;
   apb_slave slv;
   apb_rw_channel resp_chan;

   int stop_after = 10;

   virtual function void build();
      super.build();
      this.resp_chan = new("APB Response", "0");
      this.slv = new("0", 0, tb_top.apb0, null,
                     this.resp_chan);
   endfunction: build
```

```
   virtual task start();
      super.start();
      this.slv.start_xactor();

      fork
         forever begin
            apb_rw tr;
            this.resp_chan.peek(tr);
            if (tr.kind == apb_rw::READ) begin
               tr.kind.rand_mode(0);
               tr.addr.rand_mode(0);
               if (!tr.randomize()) begin
                  `vmm_error(log,
                      "Unable to randomize APB response");
               end
            end
            this.resp_chan.get(tr);
         end

         forever begin
            apb_rw tr;
            this.slv.notify.wait_for(apb_slave::RESPONSE);
            this.stop_after--;
            if (this.stop_after <= 0) -> this.end_test;
            $cast(tr,
             this.slv.notify.status(apb_slave::RESPONSE));
            tr.display("Responded: ");
         end
      join_none
   endtask: start

   virtual task wait_for_end();
      super.wait_for_end();
      @ (this.end_test);
   endtask: wait_for_end

endclass: tb_env

`endif
```

Step 14: Annotating Responses

The transaction descriptor created by the slave transactor is always of type `apb_rw` because of the following statement:

```
tr = new;
```

This allocates a new instance of an object of the same type as the “tr” variable. However, a user may wish to annotate the transaction descriptor with additional information inside an extension of the “pre_response” callback method. Unfortunately, the `apw_rw` transaction descriptor cannot be written to meet the unpredictable needs of users for annotating it with arbitrary information.

Using a factory pattern, a user can cause the slave transactor to instantiate an extension of the `apb_rw` class (Rule 4-115) that will then be filled in by the transactor but can also provide additional properties and methods for user-specified annotations.

File: apb/apb_slave.sv

```
`ifndef APB_SLAVE__SV
`define APB_SLAVE__SV

`include "apb_if.sv"
`include "apb_rw.sv"

class apb_slave_cfg;
   rand bit [31:0] start_addr = 32'h0000_0000;
   rand bit [31:0] end_addr   = 32'hFFFF_FFFF;

   constraint apb_slave_cfg_valid {
      end_addr >= start_addr;
   }

   virtual function string psdisplay(string prefix = "");
     $sformat(psdisplay, "%sAPB Slave Config: [`h%h-`h%h]",
                prefix, this.start_addr, this.end_addr);
   endfunction: psdisplay
```

```
endclass: apb_slave_cfg

typedef class apb_slave;
class apb_slave_cbs extends vmm_xactor_callbacks;
   virtual function void pre_response(apb_slave xact,
                                      apb_rw    cycle);
   endfunction: pre_response

   virtual function void post_response(apb_slave xactor,
                                       apb_rw    cycle);
   endfunction: post_response
endclass: apb_slave_cbs

class apb_slave extends vmm_xactor;
   virtual apb_if.slave sigs;
   local apb_slave_cfg cfg;
   apb_rw_channel resp_chan;
   typedef enum {RESPONSE} notifications_e;

   apb_rw tr_factory;

   local bit [31:0] ram[*];

   function new(string               name,
                int unsigned         stream_id,
                virtual apb_if.slave sigs,
                apb_slave_cfg        cfg = null,
                apb_rw_channel       resp_chan = null,
                apb_rw               tr_factory = null);
      super.new("APB Slave", name, stream_id);
      this.sigs = sigs;

      if (cfg == null) cfg = new;
      this.cfg = cfg;
      this.resp_chan = resp_chan;
      if (tr_factory == null) tr_factory = new;
      this.tr_factory = tr_factory;

      this.notify.configure(RESPONSE);
   endfunction: new

   virtual function void reconfigure(apb_slave_cfg cfg);
      this.cfg = cfg;
   endfunction: reconfigure

   virtual function void poke(bit [31:0] addr,
```

```
                                bit [31:0] data);
   if (addr < this.cfg.start_addr ||
       addr > this.cfg.end_addr) begin
      `vmm_error(this.log, "Out-of-range poke");
      return;
   end
   this.ram[addr] = data;
endfunction: poke

virtual function bit [31:0] peek(bit [31:0] addr);
   if (addr < this.cfg.start_addr ||
       addr > this.cfg.end_addr) begin
      `vmm_error(this.log, "Out-of-range peek");
      return 'x;
   end
  return (this.ram.exists(addr)) ? this.ram[addr] : 'x;
endfunction: peek

virtual function void reset_xactor(reset_e rst_typ =
                                   SOFT_RST);
   super.reset_xactor(rst_typ);
   this.resp_chan.flush();
   this.sigs.sck.prdata <= 'z;
endfunction: reset_xactor

virtual protected task main();
   super.main();
   forever begin
      apb_rw tr;

      this.sigs.sck.prdata <= 'z;
      this.wait_if_stopped();

  `vmm_trace(log, "Waiting for start of transaction...");

      // Wait for a SETUP cycle
      do @ (this.sigs.sck);
      while (this.sigs.sck.psel !== 1'b1 ||
             this.sigs.sck.penable !== 1'b0 ||
           this.sigs.sck.paddr < this.cfg.start_addr ||
             this.sigs.sck.paddr > this.cfg.end_addr);
      $cast(tr, this.tr_factory.allocate());

      tr.notify.indicate(vmm_data::STARTED);
      tr.kind = (this.sigs.sck.pwrite) ?
                   apb_rw::WRITE : apb_rw::READ;
```

```
tr.addr = this.sigs.sck.paddr;

`vmm_trace(log, {"Responding to transaction...\n",
                 tr.psdisplay("   ")});
if (tr.kind == apb_rw::READ) begin
   if (this.resp_chan == null) begin
     if (!this.ram.exists(tr.addr)) tr.data = 'x;
      else tr.data = this.ram[tr.addr];
   end
   else begin
      bit abort = 0;
      fork
         begin
            fork
              begin
                 @ (this.sigs.sck);
       `vmm_error(this.log, "No response in time");
                 abort = 1;
              end
              this.resp_chan.put(tr);
            join_any
            disable fork;
         end
      join
      if (abort) continue;
   end

  `vmm_callback(apb_slave_cbs, pre_response(this,
                tr));

   this.sigs.sck.prdata <= tr.data;
   @ (this.sigs.sck);
end
else begin
   @ (this.sigs.sck);
   tr.data = this.sigs.sck.pwdata;

  `vmm_callback(apb_slave_cbs, pre_response(this,
                tr));

   if (this.resp_chan == null)
      this.ram[tr.addr] = tr.data;
   else this.resp_chan.sneak(tr);
end
if (this.sigs.sck.penable !== 1'b1) begin
  `vmm_error(this.log, "APB protocol violation:\n
```

```
                    SETUP cycle not followed by ENABLE cycle");
            end
            tr.notify.indicate(vmm_data::ENDED);

            `vmm_callback(apb_slave_cbs, post_response(this,
                          tr));

            `vmm_trace(log, {"Responded to transaction...\n",
                             tr.psdisplay("   ")});
            this.notify.indicate(RESPONSE, tr);
      end
      endtask: main
endclass: apb_slave

`endif
```

With the transaction descriptors allocated using a factory pattern and the transaction descriptor accessible in a callback method before a response is returned, a test may now add user-defined information to a transaction descriptor that may be used to determine the transaction response.

File: Command_Slave_Xactor/test_annotate.sv

```
`include "tb_env.sv"

program annotate_test;

class annotated_apb_rw extends apb_rw;
   string note;

   virtual function vmm_data allocate();
      annotated_apb_rw tr = new;
      return tr;
   endfunction

   virtual function vmm_data copy(vmm_data to = null);
      annotated_apb_rw tr;

      if (to == null) tr = new;
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot copy to a \n
                     non-annotated_apb_rw  instance");
```

```
         return null;
      end

      super.copy(tr);
      tr.note = this.note;

      return tr;
   endfunction

   virtual function string psdisplay(string prefix = "");
     psdisplay = {super.psdisplay(prefix), " (", this.note,
                  ")"};
   endfunction
endclass

class annotate_tr extends apb_slave_cbs;

   local int    seq = 0;
   local string format = "[-%0d-]";

   function new(string format = "");
      if (format != "") this.format = format;
   endfunction

   virtual function void pre_response(apb_slave xactor,
                                      apb_rw    cycle);
      annotated_apb_rw tr;
      if (!$cast(tr, cycle)) begin
         `vmm_error(xactor.log, "Transaction descriptor \n
                    is not a annotated_apb_rw");
         return;
      end

      $sformat(tr.note, this.format, this.seq++);
   endfunction: pre_response
endclass


vmm_log log = new("Test", "Annotate");
tb_env env = new;
initial begin
   env.stop_after = 5;

   env.build();
   begin
      annotated_apb_rw tr = new;
```

```
      annotate_tr      cb = new;

      env.slv.tr_factory = tr;
      env.slv.append_callback(cb);
   end

   env.run();

   $finish();
end

endprogram
```

Step 15: Top-Level File

To help users include all necessary files without having to know the detailed filenames and file structure of the transactor, interface and transaction descriptor, it is a good idea to create a top-level file that will automatically include all source files that make up the verification IP for a protocol.

File: apb/apb.sv

```
`ifndef APB__SV
`define APB__SV

`include "vmm.sv"
`include "apb_if.sv"
`include "apb_rw.sv"
`include "apb_slave.sv"

`endif
```

In this example, we implemented only a slave transactor; but a complete VIP for a protocol would also include a master transactor and a passive monitor. All of these transactors would be included in the top-level file.

Step 16: Congratulations!

You have now completed the creation of a VMM-compliant command-layer slave transactor!

Upon reading this primer, you probably realized that there is much code that is similar across different monitors. Wouldn't be nice if you could simply cut-and-paste from an existing VMM-compliant monitor and only modify what is unique or different for your protocol? That can easily be done using the "vmmgen" tool provided with VCS 2006.06-6. Based on a few simple question and answers, it will create a template for various components of a VMM-compliant monitor.

Command

```
% vmmgen -l sv
```

The relevant templates for writing command-layer slave transactors are:

- Physical interface declaration
- Transaction Descriptor
- Reactive Driver, Physical-level, Half-duplex

Note that the menu number used to select the appropriate template may differ from the number in the above list.

You may consider reading other publications in this series to learn how to write VMM-compliant command-layer master transactors, command-layer monitors, functional-layer transactors or verification environments.

VMM Primer:
Writing Command-Layer Monitors

Author:
Janick Bergeron

Version 0.3 / Dec 01, 2006

Introduction

The *Verification Methodology Manual for SystemVerilog* book was never written as a training book. It was designed to be a reference document that defines what is—and is not—compliant with the methodology described within it.

This primer is designed to learn how to write VMM-compliant command-layer monitors—transactors that observe pin wiggling on one side and report the observed transactions on a transaction-level interface on the other side. Other primers will eventually cover other aspects of developing VMM-compliant verification assets, such as master and slave transactors, functional-layer transactors, generators, assertions and verification environments.

The protocol used in this primer was selected for its simplicity. Because of its simplicity, it does not require the use of many elements of the VMM standard library. It is sufficient to achieve the goal of demonstrating, step-by-step, how to create a simple VMM-compliant master transactor.

This document is written in the same order you would implement a command-layer monitor. As such, you should read it in a sequential fashion. You can use the same sequence to create your own specific transactor.

A word of caution however: it may be tempting to stop reading this primer half way through, as soon as a functional monitor is available. VMM compliance is a matter of degree. Once a certain minimum level of functionality is met, a monitor may be declared VMM compliant. But additional VMM functionality—such as callbacks—will make it much easier to use in different verification environments. Therefore, you should read—and apply—this primer in its entirety.

This primer will show how to apply the various VMM guidelines, not attempt to justify them or present alternatives. If you are interested in learning more about the justifications of various techniques and approaches used in this primer, you should refer to the VMM book under the relevant quoted rules and recommendations.

The Protocol

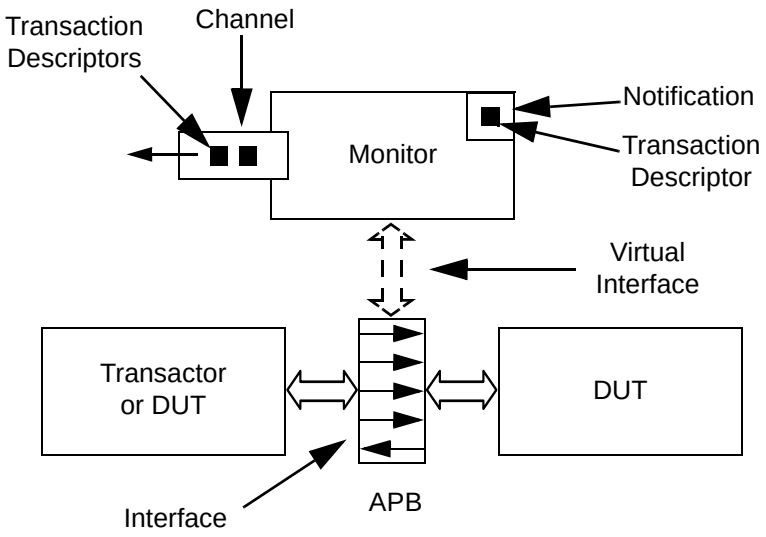
The protocol used in this primer is the AMBA™ Peripheral Bus (APB) protocol. It is a simple single-master address-based parallel bus providing atomic individual read and write cycles. The protocol specification can be found in the AMBA™ Specification (Rev 2.0) available from ARM (http://arm.com).

When writing a reusable monitor, you have to think about all possible applications it may be used in, not just the device you are using it for the first time. Therefore, even though the device in this primer only supports 8 address bits and 16 data bits, the APB monitor should be written for the entire 32-bit of address and data information.

The Verification Components

Figure 1 illustrates the various components that will be created throughout this primer. A command-layer monitor interfaces directly to the DUT signals and reports all observed transactions on a transaction-level interface.

Figure 1 Components Used in this Primer



Step 1: The Interface

The first step is to define the physical signals used by the protocol to exchange information between a master and a slave. A single exchange of information (a READ or a WRITE operation) is called a *transaction*. There may be multiple slaves on an APB bus but there can only be one master. Slaves are differentiated by responding to different address ranges.

The signals are declared inside an *interface* (Rule 4-4). The name of the interface is prefixed with “apb_” to identify that it belongs to the APB protocol (Rule 4-5). The entire content of the file declaring the **`interface`** is embedded in an **`` `ifndef/`define/`endif ``** construct. This is an old C trick that allows the file to be included multiple times, whenever required, without causing multiple-definition errors.

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if;
   ...
endinterface: apb_if

`endif
```

The signals, listed in the AMBA™ Specification in Section 2.4, are declared as **`wires`** (Rule 4-6) inside the **`interface`**.

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if(input bit pclk);
   wire [31:0] paddr;
   wire        psel;
   wire        penable;
   wire        pwrite;
   wire [31:0] prdata;
   wire [31:0] pwdata;
   ...
endinterface: apb_if

`endif
```

Because this is a synchronous protocol, **`clocking`** blocks are used to define the direction and sampling of the signals (Rule 4-7, 4-11).

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if(input bit pclk);
   wire [31:0] paddr;
   wire        psel;
   wire        penable;
```

```
   wire        pwrite;
   wire [31:0] prdata;
   wire [31:0] pwdata;

   clocking pck @(posedge pclk);
      input paddr, psel, penable, pwrite, prdata, pwdata;
   endclocking: pck
   ...
endinterface: apb_if

`endif
```

The **`clocking`** block defining the synchronous signals is specified in the **`modport`** for the APB monitor (Rule 4-9, 4-11, 4-12). The clock signal need not be specified as it is implicit in the **`clocking`** block.

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if(input bit pclk);
   wire [31:0] paddr;
   wire        psel;
   wire        penable;
   wire        pwrite;
   wire [31:0] prdata;
   wire [31:0] pwdata;

   clocking pck @(posedge pclk);
      input paddr, psel, penable, pwrite, prdata, pwdata;
   endclocking: pck

   modport passive(clocking pck);

endinterface: apb_if

`endif
```

The **interface** declaration is now sufficient for writing a passive APB monitor. To be fully compliant, it should eventually include a **modport** for a master and a slave monitor transactor (Rule 4-9). These can be added later, when these transactors will be written.

Step 2: Connecting to the DUT

The interface may now be connected to the DUT. It is instantiated in a top-level **module**, alongside of the DUT instantiation (Rule 4-13). The connection to the DUT pins are specified using a hierarchical reference to the **wires** in the **interface** instance.

File: Command_Monitor_Xactor/tb_top.sv

```
module tb_top;
   ...
   apb_if apb0(...);
   ...
   master_ip dut_mst(...,
                     .apb_addr   (apb0.paddr[7:0]  ),
                     .apb_sel    (apb0.psel        ),
                     .apb_enable (apb0.penable     ),
                     .apb_write  (apb0.pwrite      ),
                     .apb_rdata  (apb0.prdata[15:0]),
                     .apb_wdata  (apb0.pwdata[15:0]),
                     ...);
   slave_ip dut_slv(...,
                    .apb_addr   (apb0.paddr[7:0]  ),
                    .apb_sel    (apb0.psel        ),
                    .apb_enable (apb0.penable     ),
                    .apb_write  (apb0.pwrite      ),
                    .apb_rdata  (apb0.prdata[15:0]),
                    .apb_wdata  (apb0.pwdata[15:0]),
                    ...);

endmodule: tb_top
```

This top-level module also contains the clock generators (Rule 4-15), using the **bit** type (Rule 4-17) and ensuring that no clock edges will occur at time zero (Rule 4-16).

File: Command_Monitor_Xactor/tb_top.sv

```
module tb_top;
   bit clk = 0;
   apb_if apb0(clk);
   ...

   my_design dut(...,
                 .apb_addr   (apb0.paddr[7:0]  ),
                 .apb_sel    (apb0.psel        ),
                 .apb_enable (apb0.penable     ),
                 .apb_write  (apb0.pwrite      ),
                 .apb_rdata  (apb0.prdata[15:0]),
                 .apb_wdata  (apb0.pwdata[15:0]),
                 ...
                 .clk        (clk));

   always #10 clk = ~clk;
endmodule: tb_top
```

Step 3: The Transaction Descriptor

The next step is to define the APB transaction descriptor (Rule 4-54). This descriptor is a **class** (Rule 4-53) extended from the **vmm_data** class (Rule 4-55), containing a public property enumerating the various transactions that can be observed by the monitor (Rule 4-60, 4-62) and public properties for each parameter or value in the transaction (Rule 4-59, 4-62). It also needs a **static vmm_log** property instance used to issue messages from the transaction descriptor. This instance of the message service interface is passed to the **vmm_data** constructor (Rule 4-58).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   enum {READ, WRITE} kind;
   bit   [31:0] addr;
   logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new
   ...
endclass: apb_rw
...
`endif
```

A single property is used for `data`, despite the fact that the APB bus has separate read and write data buses. Because the APB does not support concurrent read/write transactions, there can only be one data value valid at any given time. The `data` class property is interpreted differently depending on the transaction kind (Rule 4-71). In a WRITE transaction, it is interpreted as the data to be written. In a READ transaction, it is the data value that was read. The type for the `data` property is **`logic`** as it will allow the description of READ cycles to reflect unknown results.

Although the transaction descriptor is not yet VMM-compliant, it is has the minimum functionality to be used by a monitor. A transaction-level interface will be required to transfer transaction descriptors to a transactor to be executed. This is done using the **`vmm_channel`** macro (Recommendation 4-56).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   enum {READ, WRITE} kind;
   bit   [31:0] addr;
   logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new
   ...
endclass: apb_rw

`vmm_channel(apb_rw)
...
`endif
```

Step 4: The Monitor

The monitor transactor can now be started. It is a class (Rule 4-91) derived from the **`vmm_xactor`** base class (4-92).

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV
...
class apb_monitor extends vmm_xactor;
   ...
endclass: apb_monitor

`endif
```

The transactor needs a physical-level interface to observe transactions. The physical-level interface is done using a **`virtual modport`** passed to the transactor as a constructor argument (Rule 4-108) and is saved in a public property (Rule 4-109).

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_monitor extends vmm_xactor;
   virtual apb_if.passive sigs;
   ...
   function new(string                  name,
                int unsigned            stream_id,
                virtual apb_if.passive sigs,
                ...);
      super.new("APB Monitor", name, stream_id);
      this.sigs = sigs;
      ...
   endfunction: new
   ...
endclass: apb_monitor

`endif
```

The transaction descriptors are filled in from observations on the physical interface in the `main()` task (Rule 4-93). The observation of READ and WRITE transactions is coded exactly as it would be if good old Verilog was used. It is a simple matter of sampling input signals at the right point in time. The only difference is that the physical signals are accessed through the **`clocking`** block of the **`virtual modport`** instead of pins on a module. The active clock edge is defined by waiting on the **`clocking`** block itself, not an edge of an input signal (Rule 4-7, 4-12).

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_monitor extends vmm_xactor;
   virtual apb_if.passive sigs;
   ...
   function new(string                  name,
                int unsigned            stream_id,
                virtual apb_if.passive sigs,
                ...);
      super.new("APB Monitor", name, stream_id);
      this.sigs = sigs;
      ...
   endfunction: new
   ...
   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         ...
         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
         tr = new;
         ...
         tr.kind = (this.sigs.pck.pwrite) ?
                      apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.pck.paddr;

         @ (this.sigs.pck);
         if (this.sigs.pck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation:
                 SETUP cycle not followed by ENABLE cycle");
         end
         tr.data = (tr.kind == apb_rw::READ) ?
                      this.sigs.pck.prdata :
                      this.sigs.pck.pwdata;
         ...
   end
   endtask: main
endclass: apb_monitor
```

```
`endif
```

Once a monitor recognizes the start of a transaction, it must not be stopped until the end of the transaction is observed and the entire transaction is reported. Otherwise, if the monitor is stopped in the middle of a transaction stream, it will report a transaction error at best or transactions composed of information from two different transactions. Therefore, the **`vmm_xactor::wait_if_stopped()`** method is called only before the beginning of transaction.

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_monitor extends vmm_xactor;
   virtual apb_if.master sigs;
   ...
   function new(string                  name,
                int unsigned            stream_id,
                virtual apb_if.passive sigs,
                ...);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      ...
   endfunction: new
   ...
   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         this.wait_if_stopped();

         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
         tr = new;
         ...
```

```
            tr.kind = (this.sigs.pck.pwrite) ?
                        apb_rw::WRITE : apb_rw::READ;
            tr.addr = this.sigs.pck.paddr;

            @ (this.sigs.pck);
            if (this.sigs.pck.penable !== 1'b1) begin
               `vmm_error(this.log, "APB protocol violation:
                    SETUP cycle not followed by ENABLE cycle");
            end
            tr.data = (tr.kind == apb_rw::READ) ?
                        this.sigs.pck.prdata :
                        this.sigs.pck.pwdata;
            ...
      end
      endtask: main
endclass: apb_monitor

`endif
```

Step 5: Reporting Transactions

The monitor must have the ability to inform the testbench that a specific transaction has been observed. Simply displaying the observed transactions is not very useful as it will require that the results be manually checked every time. Observed transactions can be reported by indicating a notification in the **`vmm_xactor::notify`** property. The observed transaction, being derived from **`vmm_data`**, is attached to a newly defined **`OBSERVED`** notification and can be recovered by all interested parties waiting on the notification.

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
```

```
class apb_monitor extends vmm_xactor;
   virtual apb_if.master sigs;
   ...
   typedef enum {OBSERVED} notifications_e;
   ...
   function new(string                  name,
                int unsigned            stream_id,
                virtual apb_if.passive sigs,
                ...);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      ...
      this.notify.configure(OBSERVED);
   endfunction: new
   ...
   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         this.wait_if_stopped();

         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
         tr = new;
         ...
         tr.kind = (this.sigs.pck.pwrite) ?
                      apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.pck.paddr;

         @ (this.sigs.pck);
         if (this.sigs.pck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation:
                 SETUP cycle not followed by ENABLE cycle");
         end
         tr.data = (tr.kind == apb_rw::READ) ?
                      this.sigs.pck.prdata :
                      this.sigs.pck.pwdata;
         ...
         this.notify.indicate(OBSERVED, tr);
         ...
   end
   endtask: main
endclass: apb_monitor
```

```
`endif
```

Step 6: The First Test

Although not completely VMM-compliant, the monitor can now be used to monitor the activity on an APB bus. It is instantiated in a verification environment class, extended from the **`vmm_env`** base class. The monitor is constructed in the extension of the **`vmm_env::build()`** method (Rule 4-34, 4-35) and started in the extension of the **`vmm_env::start()`** method (Rule 4-41). The reference to the **`interface`** encapsulating the APB physical signals is made using a hierarchical reference in the extension of the **`vmm_env::build()`** method.

File: Command_Monitor_Xactor/tb_env.sv

```
`ifndef TB_ENV__SV
`define TB_ENV__SV

`include "vmm.sv"
`include "apb_monitor.sv"

class tb_env extends vmm_env;
   ...
   apb_monitor mon;

   virtual function void build();
      super.build();
      ...
      this.mon = new("0", 0, tb_top.apb0);
   endfunction: build

   virtual task start();
      super.start();
      ...
      this.mon.start_xactor();
      ...
   endtask: start
   ...
```

```
endclass: tb_env

`endif
```

As the simulation progress, the monitor can report the observed transactions. The monitor can also be used to determine when it is time to end the test by reporting when enough APB transactions have been observed.

```
`ifndef TB_ENV__SV
`define TB_ENV__SV

`include "vmm.sv"
`include "apb_monitor.sv"

class tb_env extends vmm_env;
   ...
   apb_monitor mon;

   int stop_after = 10;

   virtual function void build();
      super.build();
      ...
      this.mon = new("0", 0, tb_top.apb0);
   endfunction: build

   virtual task start();
      super.start();
      ...
      this.mon.start_xactor();
      ...
      fork
         ...
         forever begin
            apb_rw tr;
          this.mon.notify.wait_for(apb_monitor::OBSERVED);
            this.stop_after--;
            if (this.stop_after <= 0) -> this.end_test;
            $cast(tr,
           this.mon.notify.status(apb_monitor::OBSERVED));
            tr.display("Notified: ");
         end
      join_none
```

```
   endtask: start

   virtual task wait_for_end();
      super.wait_for_end();
      @ (this.end_test);
   endtask: wait_for_end
   ...
endclass: tb_env

`endif
```

A test can control how long the simulation should run by simply setting the value of the **`tb_env::stop_after`** property. The test is written in a **`program`** (Rule 4-27) that instantiates the verification environment (Rule 4-28).

File: Command_Monitor_Xactor/test_simple.sv

```
`include "tb_env.sv"

program simple_test;

vmm_log log = new("Test", "Simple");
tb_env env = new;
initial begin
   env.stop_after = 5;
   env.run();

   $finish();
end

endprogram
```

Step 7: Standard Methods

The transactions reported by the monitor are not very useful as they do not show the content of the observed transactions. That's because the **`psdisplay()`** method, defined in the **`vmm_data`** base class does not know about the content of the APB transaction

descriptor. For that method to display the information that is relevant for the APB transaction, it is necessary to overload this method in the transaction descriptor class (Rule 4-76).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   enum {READ, WRITE} kind;
   bit   [31:0] addr;
   logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new
   ...
   virtual function string psdisplay(string prefix = "");
      $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
               this.kind.name(), this.addr, this.data);
   endfunction: psdisplay

   ...
endclass: apb_rw

`vmm_channel(apb_rw)
...
`endif
```

Re-running the test now yields useful and meaningful transaction monitoring. The **`vmm_data::psdisplay()`** method is one of the pre-defined methods in the **`vmm_data`** base class that users expect to be provided to simplify and abstract common operations on transaction descriptors. These common operations include creating transaction descriptors (**`vmm_data::allocate()`**), copying transaction descriptors (**`vmm_data::copy()`**), comparing

transaction descriptors (**`vmm_data::compare()`**) and checking that the content of transaction descriptors is valid (**`vmm_data::is_valid()`**) (Rule 4-76).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   enum {READ, WRITE} kind;
   bit   [31:0] addr;
   logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new

   virtual function vmm_data allocate();
      apb_rw tr = new;
      return tr;
   endfunction: allocate

   virtual function vmm_data copy(vmm_data to = null);
      apb_rw tr;

      if (to == null) tr = new;
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot copy into non-apb_rw
                     instance");
         return null;
      end

      super.copy_data(tr);
      tr.kind = this.kind;
      tr.addr = this.addr;
      tr.data = this.data;

      return tr;
   endfunction: copy
```

```
    virtual function string psdisplay(string prefix = "");
       $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
                this.kind.name(), this.addr, this.data);
    endfunction: psdisplay

    virtual function bit is_valid(bit silent = 1,
                                  int kind   = -1);
       return 1;
    endfunction: is_valid

    virtual function bit compare(input  vmm_data to,
                                 output string   diff,
                                input  int      kind = -1);
       apb_rw tr;

       if (to == null) begin
        `vmm_fatal(log, "Cannot compare to NULL reference");
         return 0;
       end
       else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot compare against non-apb_rw
                    instance");
         return 0;
       end

       if (this.kind != tr.kind) begin
        $sformat(diff, "Kind %s !== %s", this.kind, tr.kind);
         return 0;
       end

       if (this.addr !== tr.addr) begin
          $sformat(diff, "Addr 0x%h !== 0x%h", this.addr,
                   tr.addr);
          return 0;
       end

       if (this.data !== tr.data) begin
          $sformat(diff, "Data 0x%h !== 0x%h", this.data,
             tr.data);
          return 0;
       end

       return 1;
    endfunction: compare

endclass: apb_rw
```

```
`vmm_channel(apb_rw)
...
`endif
```

Three other standard methods, **`vmm_data:byte_size()`**, **`vmm_data::byte_pack()`** and **`vmm_data::byte_unpack()`** should also be overloaded for packet-oriented transactions, where the content of the transaction is transmitted over a physical interface (Recommendation 4-77).

Step 8: More on Transactions

The **`vmm_notify`** notification service interface can notify an arbitrary number of threads or transactors but it can store only one transaction at a time. Should a higher-level thread or transactor have the potential to block, it is possible that transactions will be missed. The **`vmm_channel`** transaction-level interface is used to provide a buffering reporting mechanism, where the transaction descriptors are kept until they are explicitly consumed. Transactions reported via a **`vmm_notify`** can be consumed by an arbitrary number of consumers but must be consumed in zero time. Transactions reported via a **`vmm_channel`** can only be consumed by a single consumer, but that consumer can only consume transactions at its own pace. Should there be no consumers, a **`vmm_notify`** mechanism will not accumulate transaction descriptors. However, a channel with no consumer will accumulate transaction descriptors, unless it is sunk using the **`vmm_channel::sink()`** method. The output channel, passed to the transactor as constructor arguments (Recommendation 4-113), is saved in a public property (Rule 4-112). It is sunk by default to avoid accidental memory leakage. Transaction

descriptors are added to the output channel using the **`vmm_channel::sneak()`** method (Rule 4-140) to avoid the monitor from blocking on a full channel and missing transactions.

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_monitor extends vmm_xactor;
   virtual apb_if.master sigs;
   apb_rw_channel        out_chan;

   typedef enum {OBSERVED} notifications_e;
   ...
   function new(string                  name,
                int unsigned            stream_id,
                virtual apb_if.passive sigs,
                apb_rw_channel         out_chan = null
                ...);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;

      if (out_chan == null) begin
        out_chan = new("APB Monitor Output Channel", name);
         out_chan.sink();
      end
      this.out_chan = out_chan;
      ...
      this.notify.configure(OBSERVED);
   endfunction: new
   ...
   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         this.wait_if_stopped();

         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
```

```
            tr = new;
            ...
            tr.kind = (this.sigs.pck.pwrite) ?
                        apb_rw::WRITE : apb_rw::READ;
            tr.addr = this.sigs.pck.paddr;

            @ (this.sigs.pck);
            if (this.sigs.pck.penable !== 1'b1) begin
               `vmm_error(this.log, "APB protocol violation:
                    SETUP cycle not followed by ENABLE cycle");
            end
            tr.data = (tr.kind == apb_rw::READ) ?
                        this.sigs.pck.prdata :
                        this.sigs.pck.pwdata;
            ...
            this.notify.indicate(OBSERVED, tr);
            this.out_chan.sneak(tr);
      end
      endtask: main
endclass: apb_monitor

`endif
```

When the transactor is reset, the output channel must be flushed. This is accomplished in the extension of the **`vmm_xactor::reset_xactor()`** method (Table A-8).

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_monitor extends vmm_xactor;
   virtual apb_if.master sigs;
   apb_rw_channel        out_chan;

   typedef enum {OBSERVED} notifications_e;
   ...
   function new(string                  name,
                int unsigned            stream_id,
                virtual apb_if.passive sigs,
```

```
            apb_rw_channel         out_chan = null
            ...);
   super.new("APB Master", name, stream_id);
   this.sigs = sigs;

   if (out_chan == null) begin
     out_chan = new("APB Monitor Output Channel", name);
      out_chan.sink();
   end
   this.out_chan = out_chan;
   ...
   this.notify.configure(OBSERVED);
endfunction: new

virtual function void reset_xactor(reset_e rst_typ =
                                   SOFT_RST);
   super.reset_xactor(rst_typ);
   this.out_chan.flush();
endfunction: reset_xactor

virtual protected task main();
   super.main();
   forever begin
      apb_rw tr;
      this.wait_if_stopped();

      // Wait for a SETUP cycle
      do @ (this.sigs.pck);
      while (this.sigs.pck.psel !== 1'b1 ||
             this.sigs.pck.penable !== 1'b0);
      tr = new;
      ...
      tr.kind = (this.sigs.pck.pwrite) ?
                   apb_rw::WRITE : apb_rw::READ;
      tr.addr = this.sigs.pck.paddr;

      @ (this.sigs.pck);
      if (this.sigs.pck.penable !== 1'b1) begin
         `vmm_error(this.log, "APB protocol violation:
              SETUP cycle not followed by ENABLE cycle");
      end
      tr.data = (tr.kind == apb_rw::READ) ?
                   this.sigs.pck.prdata :
                   this.sigs.pck.pwdata;
      ...
      this.notify.indicate(OBSERVED, tr);
```

```
         this.out_chan.sneak(tr);
   end
   endtask: main
endclass: apb_monitor

`endif
```

It may be useful for the higher-level functions using the transactions reported by the monitor to know when the transaction was started and when it ended. These transaction endpoints are recorded in the transaction descriptor itself by indicating the **`vmm_data::STARTED`** and **`vmm_data::ENDED`** notifications (Rule 4-142).

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_monitor extends vmm_xactor;
   virtual apb_if.master sigs;
   apb_rw_channel        out_chan;

   typedef enum {OBSERVED} notifications_e;
   ...
   function new(string                  name,
                int unsigned            stream_id,
                virtual apb_if.passive sigs,
                apb_rw_channel         out_chan = null
                ...);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;

      if (out_chan == null) begin
        out_chan = new("APB Monitor Output Channel", name);
         out_chan.sink();
      end
      this.out_chan = out_chan;
      ...
      this.notify.configure(OBSERVED);
   endfunction: new
```

```
   virtual function void reset_xactor(reset_e rst_typ =
                                      SOFT_RST);
      super.reset_xactor(rst_typ);
      this.out_chan.flush();
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         this.wait_if_stopped();

         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
         tr = new;
         tr.notify.indicate(vmm_data::STARTED);

         tr.kind = (this.sigs.pck.pwrite) ?
                      apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.pck.paddr;

         @ (this.sigs.pck);
         if (this.sigs.pck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation:
                 SETUP cycle not followed by ENABLE cycle");
         end
         tr.data = (tr.kind == apb_rw::READ) ?
                      this.sigs.pck.prdata :
                      this.sigs.pck.pwdata;
         tr.notify.indicate(vmm_data::ENDED);
         ...
         this.notify.indicate(OBSERVED, tr);
         this.out_chan.sneak(tr);
   end
   endtask: main
endclass: apb_monitor

`endif
```

Step 9: Debug Messages

To be truly reusable, it should be possible to understand what the monitor does and debug its operation without having to inspect the source code. This capability may even be a basic requirement if you plan on shipping encrypted or compiled code.

Debug messages should be added at judicious points to indicate what the monitor is about to do, is doing or has done. These debug messages are inserted using the **`vmm_trace()**, **`vmm_debug()** or **`vmm_verbose()** macros (Recommendation 4-51).

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_monitor extends vmm_xactor;
   virtual apb_if.master sigs;
   apb_rw_channel        out_chan;

   typedef enum {OBSERVED} notifications_e;
   ...
   function new(string                  name,
                int unsigned            stream_id,
                virtual apb_if.passive sigs,
                apb_rw_channel         out_chan = null
                ...);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;

      if (out_chan == null) begin
        out_chan = new("APB Monitor Output Channel", name);
         out_chan.sink();
      end
      this.out_chan = out_chan;
      ...
      this.notify.configure(OBSERVED);
```

```
   endfunction: new

   virtual function void reset_xactor(reset_e rst_typ =
                                      SOFT_RST);
      super.reset_xactor(rst_typ);
      this.out_chan.flush();
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         this.wait_if_stopped();

     `vmm_trace(log, "Waiting for start of transaction...");

         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
         tr = new;
         tr.notify.indicate(vmm_data::STARTED);

         tr.kind = (this.sigs.pck.pwrite) ?
                      apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.pck.paddr;

         @ (this.sigs.pck);
         if (this.sigs.pck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation:
                 SETUP cycle not followed by ENABLE cycle");
         end
         tr.data = (tr.kind == apb_rw::READ) ?
                      this.sigs.pck.prdata :
                      this.sigs.pck.pwdata;
         tr.notify.indicate(vmm_data::ENDED);
         ...
         `vmm_trace(log, {"Observed transaction...\n",
                          tr.psdisplay("   ")});
         this.notify.indicate(OBSERVED, tr);
         this.out_chan.sneak(tr);
   end
   endtask: main
endclass: apb_monitor

`endif
```

You can now run the “simple test” with the latest version of the monitor and increase the message verbosity to see debug messages displayed as the monitor observes the various transactions.

File: Command_Monitor_Xactor/Makefile

```
% vcs -sverilog -ntb_opts rmm +vmm_log_default=trace ...
```

Step 10: Extension Points

A callback method is a third mechanism for reporting observed transactions. A callback method should be provided after a transaction has been observed (Recommendation 4-155). This callback method allows the observed transaction to be recorded in a functional coverage model or checked against an expected response.

The callback method is first defined as a **`virtual void function`** (Rule 4-160) in a callback façade class extended from the **`vmm_xactor_callbacks`** base class (Rule 4-159).

Next, the callback method needs to be invoked at the appropriate point in the execution of the monitor, using the **`` `vmm_callback() ``** macro (Rule 4-163).

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include "apb_if.sv"
`include "apb_rw.sv"

typedef class apb_monitor;
class apb_monitor_cbs extends vmm_xactor_callbacks;
```

```
   virtual function void post_cycle(apb_monitor xactor,
                                    apb_rw      cycle);
   endfunction: post_cycle
endclass: apb_monitor_cbs

class apb_monitor extends vmm_xactor;
   virtual apb_if.master sigs;
   apb_rw_channel         out_chan;

   typedef enum {OBSERVED} notifications_e;
   ...
   function new(string                  name,
                int unsigned            stream_id,
                virtual apb_if.passive sigs,
                apb_rw_channel         out_chan   = null,
                ...);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      if (out_chan == null) begin
        out_chan = new("APB Monitor Output Channel", name);
         out_chan.sink();
      end
      this.out_chan = out_chan;
      ...
      this.notify.configure(OBSERVED);
   endfunction: new

   virtual function void reset_xactor(reset_e rst_typ =
                                      SOFT_RST);
      super.reset_xactor(rst_typ);
      this.out_chan.flush();
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         this.wait_if_stopped();

     `vmm_trace(log, "Waiting for start of transaction...");

         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
```

```
            tr = new;
            tr.notify.indicate(vmm_data::STARTED);

            tr.kind = (this.sigs.pck.pwrite) ?
                        apb_rw::WRITE : apb_rw::READ;
            tr.addr = this.sigs.pck.paddr;

            @ (this.sigs.pck);
            if (this.sigs.pck.penable !== 1'b1) begin
               `vmm_error(this.log, "APB protocol violation:
                     SETUP cycle not followed by ENABLE cycle");
            end
            tr.data = (tr.kind == apb_rw::READ) ?
                        this.sigs.pck.prdata :
                        this.sigs.pck.pwdata;
            tr.notify.indicate(vmm_data::ENDED);

         `vmm_callback(apb_monitor_cbs, post_cycle(this, tr));

            `vmm_trace(log, {"Observed transaction...\n",
                             tr.psdisplay("   ")});

            this.notify.indicate(OBSERVED, tr);
            this.out_chan.sneak(tr);
      end
      endtask: main
endclass: apb_monitor

`endif
```

The transaction descriptor created by the monitor is always of type `apb_rw` because of the following statement:

```
tr = new;
```

This allocates a new instance of an object of the same type as the `tr` variable. However, a user may wish to annotate the transaction descriptor with additional information inside an extension of the “post_cycle” callback method. Unfortunately, the `apw_rw` transaction descriptor cannot be written to meet the unpredictable needs of users for annotating it with arbitrary information.

Using a factory pattern, a user can cause the monitor to instantiate an extension of the `apb_rw` class (Rule 4-115) that will then be filled in by the monitor but can also provide additional properties and methods for user-specified annotations.

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include "apb_if.sv"
`include "apb_rw.sv"

typedef class apb_monitor;
class apb_monitor_cbs extends vmm_xactor_callbacks;
   virtual function void post_cycle(apb_monitor xactor,
                                    apb_rw     cycle);
   endfunction: post_cycle
endclass: apb_monitor_cbs

class apb_monitor extends vmm_xactor;
   virtual apb_if.master sigs;
   apb_rw_channel         out_chan;

   typedef enum {OBSERVED} notifications_e;

   apb_rw tr_factory;

   function new(string                  name,
                int unsigned            stream_id,
                virtual apb_if.passive sigs,
                apb_rw_channel          out_chan   = null,
                apb_rw                  tr_factory = null);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      if (out_chan == null) begin
        out_chan = new("APB Monitor Output Channel", name);
         out_chan.sink();
      end
      this.out_chan = out_chan;

      if (tr_factory == null) tr_factory = new;
      this.tr_factory = tr_factory;

      this.notify.configure(OBSERVED);
```

```
endfunction: new

virtual function void reset_xactor(reset_e rst_typ =
                                   SOFT_RST);
   super.reset_xactor(rst_typ);
   this.out_chan.flush();
endfunction: reset_xactor

virtual protected task main();
   super.main();
   forever begin
      apb_rw tr;
      this.wait_if_stopped();

  `vmm_trace(log, "Waiting for start of transaction...");

      // Wait for a SETUP cycle
      do @ (this.sigs.pck);
      while (this.sigs.pck.psel !== 1'b1 ||
             this.sigs.pck.penable !== 1'b0);

      $cast(tr, this.tr_factory.allocate());
      tr.notify.indicate(vmm_data::STARTED);

      tr.kind = (this.sigs.pck.pwrite) ?
                   apb_rw::WRITE : apb_rw::READ;
      tr.addr = this.sigs.pck.paddr;

      @ (this.sigs.pck);
      if (this.sigs.pck.penable !== 1'b1) begin
         `vmm_error(this.log, "APB protocol violation:
              SETUP cycle not followed by ENABLE cycle");
      end
      tr.data = (tr.kind == apb_rw::READ) ?
                   this.sigs.pck.prdata :
                   this.sigs.pck.pwdata;
      tr.notify.indicate(vmm_data::ENDED);

  `vmm_callback(apb_monitor_cbs, post_cycle(this, tr));

      `vmm_trace(log, {"Observed transaction...\n",
                       tr.psdisplay("   ")});

      this.notify.indicate(OBSERVED, tr);
      this.out_chan.sneak(tr);
```

```
   end
   endtask: main
endclass: apb_monitor

`endif
```

With the transaction descriptors allocated using a factory pattern and the transaction descriptor accessible in a callback method before it is reported to other transactors, a test may now add user-defined information to a transaction descriptor.

File: Command_Monitor_Xactor/test_annotate.sv

```
`include "tb_env.sv"

program annotate_test;

class annotated_apb_rw extends apb_rw;
   string note;

   virtual function vmm_data allocate();
      annotated_apb_rw tr = new;
      return tr;
   endfunction

   virtual function vmm_data copy(vmm_data to = null);
      annotated_apb_rw tr;

      if (to == null) tr = new;
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot copy to a non- \n
                    annotated_apb_rw instance");
         return null;
      end

      super.copy(tr);
      tr.note = this.note;

      return tr;
   endfunction

   virtual function string psdisplay(string prefix = "");
     psdisplay = {super.psdisplay(prefix), " (", this.note,
                  ")"};
```

```
   endfunction
endclass

class annotate_tr extends apb_monitor_cbs;

   local int    seq = 0;
   local string format = "[-%0d-]";

   function new(string format = "");
      if (format != "") this.format = format;
   endfunction

   virtual function void post_cycle(apb_monitor xactor,
                                    apb_rw     cycle);
      annotated_apb_rw tr;
      if (!$cast(tr, cycle)) begin
         `vmm_error(xactor.log, "Transaction descriptor \n
                    is not a annotated_apb_rw");
         return;
      end

      $sformat(tr.note, this.format, this.seq++);
   endfunction: post_cycle
endclass


vmm_log log = new("Test", "Annotate");
tb_env env = new;
initial begin
   env.stop_after = 5;

   env.build();
   begin
      annotated_apb_rw tr = new;
      annotate_tr      cb = new;

      env.mon.tr_factory = tr;
      env.mon.append_callback(cb);
   end

   env.run();

   $finish();
end

endprogram
```

Step 11: Random Transactions

To promote the use of random stimulus and make the same transaction descriptor usable in transaction generators, all public properties in a transaction descriptor should be declared as **rand** (Rules 4-59, 4-60, 4-62). It is also a good idea to pre-define random transaction generators whenever transaction descriptors are defined. It is a simple matter of using the **`vmm_atomic_gen()** and **`vmm_scenario_gen()** macros (Recommendation 5-23, 5-24) in the transaction descriptor file.

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   rand enum {READ, WRITE} kind;
   rand bit   [31:0] addr;
   rand logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new

   virtual function vmm_data allocate();
      apb_rw tr = new;
      return tr;
   endfunction: allocate

   virtual function vmm_data copy(vmm_data to = null);
      apb_rw tr;

      if (to == null) tr = new;
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot copy into non-apb_rw \n
                     instance");
```

```
      return null;
   end

   super.copy_data(tr);
   tr.kind = this.kind;
   tr.addr = this.addr;
   tr.data = this.data;

   return tr;
endfunction: copy

virtual function string psdisplay(string prefix = "");
  $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
           this.kind.name(), this.addr, this.data);
endfunction: psdisplay

virtual function bit is_valid(bit silent = 1,
                              int kind   = -1);
   return 1;
endfunction: is_valid

virtual function bit compare(input  vmm_data to,
                             output string   diff,
                            input  int      kind = -1);
   apb_rw tr;

   if (to == null) begin
    `vmm_fatal(log, "Cannot compare to NULL reference");
      return 0;
   end
   else if (!$cast(tr, to)) begin
      `vmm_fatal(log, "Cannot compare against non- \n
                 apb_rw instance");
      return 0;
   end

   if (this.kind != tr.kind) begin
    $sformat(diff, "Kind %s !== %s", this.kind, tr.kind);
      return 0;
   end

   if (this.addr !== tr.addr) begin
      $sformat(diff, "Addr 0x%h !== 0x%h", this.addr,
               tr.addr);
      return 0;
   end
```

```
      if (this.data !== tr.data) begin
         $sformat(diff, "Data 0x%h !== 0x%h", this.data,
                  tr.data);
         return 0;
      end

      return 1;
   endfunction: compare

endclass: apb_rw

`vmm_channel(apb_rw)
`vmm_atomic_gen(apb_rw, "APB Bus Cycle")
`vmm_scenario_gen(apb_rw, "APB Bus Cycle")

`endif
```

Step 12: Top-Level File

To help users include all necessary files without having to know the detailed filenames and file structure of the transactor, interface and transaction descriptor, it is a good idea to create a top-level file that will automatically include all source files that make up the verification IP for a protocol.

File: apb/apb.sv

```
`ifndef APB__SV
`define APB__SV

`include "vmm.sv"
`include "apb_if.sv"
`include "apb_rw.sv"
`include "apb_monitor.sv"

`endif
```

In this example, we implemented only a monitor; but a complete VIP for a protocol would also include a master transactor and a slave transactor. All of these transactors would be included in the top-level file.

Step 13: Congratulations!

You have now completed the creation of a VMM-compliant command-layer monitor transactor!

Upon reading this primer, you probably realized that there is much code that is similar across different monitors. Wouldn't be nice if you could simply cut-and-paste from an existing VMM-compliant monitor and only modify what is unique or different for your protocol? That can easily be done using the "vmmgen" tool provided with VMM open source ($VMM_HOME/shared/bin/vmmgen) and in VCS. Based on a few simple question and answers, it will create a template for various components of a VMM-compliant monitor.

Command

```
% vmmgen –l sv
```

The relevant templates for writing command-layer monitors are:

- Physical interface declaration
- Transaction Descriptor
- Monitor, Physical-level, Half-duplex

Note that the menu number used to select the appropriate template may differ from the number in the above list.

You may consider reading other publications in this series to learn how to write VMM-compliant command-layer master transactors, command-layer slave transactors, functional-layer transactors or verification environments.

VMM Primer:
Using the Data Stream Scoreboard

Author:
Janick Bergeron

Version 1.1 / May 17, 2006

Introduction

The VMM Data Stream Scoreboard is an application package that can be used to simply the creation of a self-checking structure of “data mover” designs. Data mover designs are design that take data on one side, transform it, and then produce data on the other side. Routers, modems, codec, DSP functions, bridges and busses are all data mover designs.

This primer is designed to learn how to create a scoreboard based on the Data Stream Scoreboard foundation classes, and how to integrate this scoreboard in a verification environment. Other primers cover other aspects of developing VMM-compliant verification assets, such as transactors, generators, assertions and verification environments.

This primer assumes that you are familiar with VMM. If you need to ramp-up on VMM itself, you should read the other primers in this series, such as “Writing a Command-Layer Command Transactor”.

The DUT used in this primer was selected for its simplicity. Because of its simplicity, it does not require the use of many elements of the VMM Data Stream Scoreboard package. It is sufficient to achieve the goal of demonstrating, step by step, how to use create a scoreboard and use it to verify a design.

This document is written in the same order you would develop a scoreboard, integrate it in a verification environment and verify your design. As such, you should read it in a sequential fashion. You can use the same sequence to create your own scoreboard and use it to verify your design.

The Verification Environment

The Design Under Test used in this primer is an AMBA™ Peripheral Bus (APB), with a single master port and three slave ports. It is a simple address decoding device with the following address map:

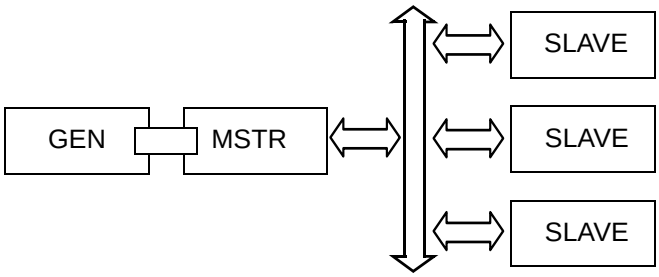
Table 1 Address Map

paddr[9:8]	**Slave**
2'b00	Slave #0, paddr[7:0]
2'b01	Slave #1, paddr[7:0]
2'b10	Slave #2, paddr[7:0]

The simplicity of the APB bus does not require the use of a full-fledge scoreboard to verify the correctness of its operation. Since it is not pipelined and does not support out-of-order execution, a simple global variable would suffice. However, the purpose of this primer is to show how to use the Data Stream Scoreboard foundation classes, not verify a complex functionality. This simple design serves the purpose quite well. Let's ignore the fact that the design is a trivial bus and assume that it is a complex pipelined bus that can execute multiple transactions simultaneously.

As shown in Figure 1, the design is exercised by an APB master transactor and the responses are provided by three APB slave transactors. These are the same transactors that were created in the primers on writing command-layer transactors. Transactions are created by an atomic generator. As is, this verification environment is not self-checking.

Figure 1 Verification Environment



Beside its simplicity, there is another reason for selecting this design as the DUT for this primer. The name “Data Stream Scoreboard” seems to imply that it is suited only for data networking applications. The documentation and method arguments themselves refer to the data to be verified as “packets”. But “packet” and “data stream” are abstract notions. They are used to identify transactions and flows of transaction and do not imply any particular technology or application. Using a bus as the DUT give you the opportunity to break any fallacious mental association between the Data Stream Scoreboard package and data networking applications.

In this example, a “packet” is an APB transaction and a stream is a transaction executed between the master and one of the slaves. There are thus two types of packets in the design: READ and WRITE cycles. And there are three streams, one for each slave.

Step 1: The Self-Checking Strategy

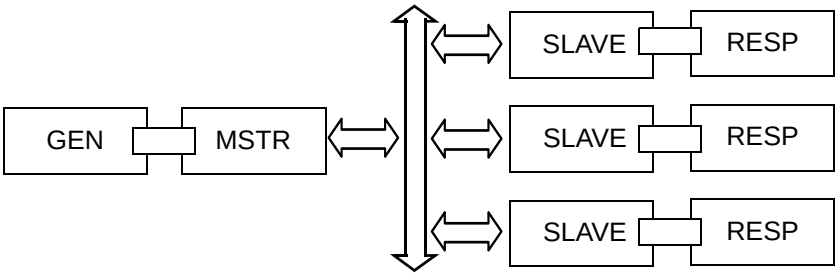
A scoreboard is only an implementation medium. It must be properly used and fit in an overall self-checking strategy to be effective at verifying the response of a design.

One possible strategy would be to use the default RAM-like behavior of each slave and perform a series of WRITE-READ cycles. The correctness would be verified by checking that the data that was read back was indeed the data that was written. But given the nature of the design, this is a rather poor self-checking strategy. It would fail to uncover several classes of functional bugs, such as misconnected address and data busses and incorrect address decoding. It also requires that a READ cycle targets an address that was previously written to, making it impossible to verify the correct operation of two back-to-back WRITE cycles to the same address.

A better strategy is to use the optional response channel in the slave transactor. When present, it allows a higher-level transactor—in this case a response generator—to provide an arbitrary response to any READ cycle. A RAM behavior is only one of the possible responses, one that happens to be built in the slave transactor.

Figure 2 shows the structure of the verification environment with slave response generators attached to each slave. These response generators do not react to WRITE cycles but provide a random response to READ cycles.

Figure 2 Random Slave Response Environment



The slave response generator can be implemented as a VMM compliant transactor and be reusable. Because the intent of this primer is not to show how to write VMM-compliant random response

generator, a simple *non-reusable* and *unconstrainable* response generator will be used. It can be coded directly in the environment as a forked-off thread.

File: DateStream_DB/tb_env.sv

```
...
class tb_env extends vmm_env;
   ...
   apb_slave slv[3];
   apb_rw_channel resp_chan[3];
   ...
   virtual function build();
      ...
      this.resp_chan[0] = new("Response", "0");
      this.resp_chan[1] = new("Response", "1");
      this.resp_chan[2] = new("Response", "2");
      this.slv[0] = new("Slave", 0, tb_top.s0, ,
                        this.resp_chan[0]);
      this.slv[1] = new("Slave", 1, tb_top.s1, ,
                        this.resp_chan[1]);
      this.slv[2] = new("Slave", 2, tb_top.s2, ,
                        this.resp_chan[2]);
   endfunction: build

   virtual function start();
      ...
      foreach (this.resp_chan[i]) begin
         int j = i;
         fork
            forever begin
               apb_rw tr;
               this.resp_chan[j].peek(tr);
               ...
               if (tr.kind == apb_rw::READ) begin
                  tr.data = $random(); // Poor random
                                       // strategy!
                  ...
               end
               this.resp_chan[j].get(tr);
            end
```

```
         join_none
      end
   endfunction: start
   ...
endclass: tb_env
```

With random responses, it is no longer possible to predict the expected response strictly from the master's interface. It is necessary to carry the expected response from the master to the slave (to verify that the right slave is targeted with the right address and write data) and from the slave to the master (to verify the read data). This requires two scoreboards: one in the master-to-slaves direction, the other in the slaves-to-master direction.

Step 2: Master-to-Slaves

The master-to-slaves scoreboard requires one queue of input transaction and three queues of expected transaction, one per slave. This is implemented using the Data Stream Scoreboard foundation class by defining one input stream and three expected streams. A unique stream identifier must be assigned to each stream. The stream identifier is chosen to facilitate the association of a slave with its corresponding stream during integration. In this case, the stream identifier corresponds to the value of 'j' in the response generator thread.

File: DataStream_SB/tb_env.sv

```
...
`include "vmm_sb.sv"

class m2s_sb extends vmm_sb_ds;
   function new();
      super.new("Master->Slave");
```

```
      this.define_stream(0, "Master",  INPUT);
      this.define_stream(0, "Slave 0", EXPECT);
      this.define_stream(1, "Slave 1", EXPECT);
      this.define_stream(2, "Slave 2", EXPECT);
   endfunction: new
   ...
endclass: m2s_sb
...
```

The expected transaction, as observed by the slaves, will be different from the one injected by the master. Because each slave only has 256 addressable locations, bits [31:8] of the address will always be zero. Any bits set by the master will be masked. It is thus necessary to perform the same transformation in the scoreboard.

File: DataStream_SB/tb_env.sv

```
...
`include "vmm_sb.sv"

class m2s_sb extends vmm_sb_ds;
   function new();
      super.new("Master->Slave");

      this.define_stream(0, "Master",  INPUT);
      this.define_stream(0, "Slave 0", EXPECT);
      this.define_stream(1, "Slave 1", EXPECT);
      this.define_stream(2, "Slave 2", EXPECT);
   endfunction: new

   virtual function bit transform(input  vmm_data in_pkt,
                                  output vmm_data out_pkts[]);
      apb_rw tr;

      $cast(tr, in_pkt.copy());
      tr.paddr[31:8] = `0;

      out_pkts = new [1];
      out_pkts[0] = tr;
```

```
   endfunction: transform
   ...
endclass: m2s_sb
...
```

By default, the comparison function use by the Data Stream Scoreboard foundation class is the **`vmm_data::compare()`** method defined for the transaction descriptor. However, when verifying the response of the master-to-slave path, the data value can only be compared against an expected value for WRITE cycles. It is thus necessary to specify a custom comparisons function in the master-to-slave scoreboard. This is done by overloading the **`vmm_sb_ds::compare()`** method.

File: DataStream_SB/tb_env.sv

```
...
`include "vmm_sb.sv"

class m2s_sb extends vmm_sb_ds;
   function new();
      super.new("Master->Slave");

      this.define_stream(0, "Master",  INPUT);
      this.define_stream(0, "Slave 0", EXPECT);
      this.define_stream(1, "Slave 1", EXPECT);
      this.define_stream(2, "Slave 2", EXPECT);
   endfunction: new

   virtual function bit transform(input  vmm_data in_pkt,
                                  output vmm_data out_pkts[]);
      apb_rw tr;

      $cast(tr, in_pkt.copy());
      tr.addr[31:8] = `0;

      out_pkts = new [1];
      out_pkts[0] = tr;
   endfunction: transform
```

```
   virtual function bit compare(vmm_data actual,
                                vmm_data expected);
      apb_rw act, exp;
      $cast(act, actual);
      $cast(exp, expected);

      if (act.kind == apb_rw::WRITE) begin
         string diff;
         return act.compare(exp, diff);
      end

      return (act.kind == exp.kind) &&
             (act.addr == exp.addr);
   endfunction: compare
endclass: m2s_sb
...
```

Step 3: Integration

Once the functionality of the scoreboard is defined, it must then be integrated with the rest of the verification environment.

The stimulus transaction executed by the master can be extracted using either the `pre_cycle()` or `post_cycle()` callback method. Because the `post_cycle()` method will only be invoked when the transaction will have completed—and hence after the slave has responded—it will be too late to check the transaction that the slave sees against what the master is executing. The `pre_cycle()` method is thus the proper integration point. The fact that the read-back data is not valid when this callback method is invoked is a non-issue since it is not compared against expected values.

File: DataStream_SB/tb_env.sv

```
...
class apb_master_to_sb extends apb_master_cbs;
   m2s_sb m2s;
   ...
   function new(m2s_sb m2s, ...);
      this.m2s = m2s;
      ...
   endfunction: new

   virtual task pre_cycle(apb_master xactor,
                          apb_rw cycle,
                          ref bit drop);
      this.m2s.insert(cycle, vmm_sb_ds::INPUT);
   endtask: pre_cycle
   ...
endclass: apb_master_to_sb

class tb_env extends vmm_env;
   m2s_sb m2s = new;
   ...
   virtual function build();
      ...
      this.mst = new("Master", 0, tb_top.m,
                     this.gen.out_chan);
      begin
         apb_master_to_sb cbs = new(this.m2s, ...);
         this.mst.append_callback(cbs);
      end
      ...
   endfunction: build
   ...
endclass: tb_env
```

Integrating the checking part is easier since the response generator is implemented directly in the environment. It is only necessary to invoke the proper checking function, identifying the stream this response has been observed on.

File: DataStream_SB/tb_env.sv

```
...
class tb_env extends vmm_env;
   ...
   virtual function start();
      ...
      foreach (this.resp_chan[i]) begin
         int j = i;
         fork
            forever begin
               apb_rw tr;
               this.resp_chan[j].peek(tr);
               this.m2s.expect_in_order(tr, j);
               if (tr.kind == apb_rw::READ) begin
                  tr.data = $random(); // Poor random
                                       // strategy!
                  ...
               end
               this.resp_chan[j].get(tr);
            end
         join_none
      end
   endfunction: start
   ...
endclass: tb_env
```

Step 4: Slaves-to-Master

The slaves-to-master scoreboard also requires three queues of input transactions, one per slave. This is implemented using the Data Stream Scoreboard foundation class by defining three input streams. A unique stream identifier must be assigned to each stream. The stream identifier is chosen to facilitate the association of a slave with its corresponding stream during integration. In this case, the stream identifier corresponds to the value of ‘j’ in the response generator thread.

File: DataStream_SB/tb_env.sv

```
...
`include "vmm_sb.sv"

class s2m_sb extends vmm_sb_ds;
   function new();
      super.new("Slave->Master");

      this.define_stream(0, "Slave 0", INPUT);
      this.define_stream(1, "Slave 1", INPUT);
      this.define_stream(2, "Slave 2", INPUT);
      this.define_stream(0, "Master",  EXPECT);
   endfunction: new
   ...
endclass: s2m_sb
...
```

The completed transactions, as reported by the master, will be different from the one replied by the slaves. The only information that is transferred from a slave to the master is the read back data. All of the remaining information is unmodified. The simplest approach is to leave the response transaction as-is and only compare the data value of READ cycles.

File: DataStream_SB/tb_env.sv

```
...
`include "vmm_sb.sv"

class s2m_sb extends vmm_sb_ds;
   function new();
      super.new("Master->Slave");

      this.define_stream(0, "Slave 0", EXPECT);
      this.define_stream(1, "Slave 1", EXPECT);
      this.define_stream(2, "Slave 2", EXPECT);
      this.define_stream(0, "Master",  EXPECT);
   endfunction: new
```

```
   virtual function bit compare(vmm_data actual,
                                vmm_data expected);
      apb_rw act, exp;
      $cast(act, actual);
      $cast(exp, expected);

      if (act.kind == apb_rw::WRITE) return 1;

      return (act.data == exp.data);
   endfunction: compare
endclass: s2m_sb
...
```

Step 5: Integration

Once the functionality of the scoreboard is defined, it must then be integrated with the rest of the verification environment.

The completed transaction observed by the master can be extracted using the **`post_cycle()`** callback method and used to compare against expected transaction if a response was expected.

File: DataStream_SB/tb_env.sv

```
...
class apb_master_to_sb extends apb_master_cbs;
   m2s_sb m2s;
   s2m_sb s2m;

   function new(m2s_sb m2s, s2m_sb s2m);
      this.m2s = m2s;
      this.s2m = s2m;
   endfunction: new

   virtual task pre_cycle(apb_master xactor,
                          apb_rw cycle,
                          ref bit drop);
      this.m2s.insert(cycle, vmm_sb_ds::INPUT);
```

```
   endtask: pre_cycle

   virtual task post_cycle(apb_master xactor,
                           apb_rw cycle);
      if (cycle.addr[9:8] == 2'b11) return;
      if (cycle.kind == apb_rw::WRITE) return;
      this.s2m.expect_in_order(cycle,
                          .inp_stream_id(cycle.addr[9:8]));
   endtask: post_cycle
endclass: apb_master_to_sb

class tb_env extends vmm_env;
   m2s_sb m2s = new;
   s2m_sb s2m = new;
   ...
   virtual function build();
      ...
      this.mst = new("Master", 0, tb_top.m,
                     this.gen.out_chan);
      begin
         apb_master_to_sb cbs = new(this.m2s, this.s2m);
         this.mst.append_callback(cbs);
      end
      ...
   endfunction: build
   ...
endclass: tb_env
```

Integrating the checking part is again easier since the response generator is implemented directly in the environment. It is only necessary to invoke the insertion function, identifying the stream this response has been observed on.

File: DataStream_SB/tb_env.sv

```
...
class tb_env extends vmm_env;
   ...
   virtual function start();
      ...
```

```
      foreach (this.resp_chan[i]) begin
         int j = i;
         fork
            forever begin
               apb_rw tr;
               this.resp_chan[j].peek(tr);
               this.m2s.expect_in_order(tr, j);
               if (tr.kind == apb_rw::READ) begin
                  tr.data = $random(); // Poor random
                                       // strategy!
                  this.s2m.insert(tr, vmm_sb_ds::INPUT,
                                  .inp_stream_id(j));
               end
               this.resp_chan[j].get(tr);
            end
         join_none
      end
   endfunction: start
   ...
endclass: tb_env
```

Step 6: Congratulations!

You have now completed the development and integration of not one but two scoreboards using the VMM Data Stream Scoreboard application package. You can verify the correct operation of the design by simulating the now self-checking verification environment.

You may consider reading other publications in this series to learn how to write VMM-compliant command-layer transactors, verification environments or integrate a Register Abstraction Layer model.

VMM Primer: Using the Register Abstraction Layer

Author:
Janick Bergeron

Updated By:
John Choi
Brett Kobernat

Version 1.4 / March 27, 2008

Introduction

The VMM Register Abstraction Layer is an application package that can be used to automate the creation of an object-oriented abstract model of the registers and memories inside a design. It also includes pre-defined tests to verify the correct implementation of the registers and memories, as specified as well as a functional coverage model to ensure that every bit of every register has been exercised.

This primer is designed to teach how to create a RAL model of the registers and memories in a design, how to integrate this model in a verification environment and how to verify the implementation of those registers and memories using the pre-defined tests. It will also show how the RAL model can be used to model the configuration and DUT driver code so it can be reusable in a system-level environment. Finally, it shows how the RAL model is used to implement additional functional tests. Other primers cover other aspects of developing VMM-compliant verification assets, such as transactors, generators, assertions and verification environments.

This primer assumes that you are familiar with VMM. If you need to ramp-up on VMM itself, you should read the other primers in this series, such as “Writing a Command-Layer Command Transactor”.

The DUT used in this primer was selected for its simplicity. As a result, it does not require the use of many elements of the VMM Register Abstraction Layer application package. The DUT has enough features to show the steps needed to create a RAL model to verify the design.

This document is written in the same order you would develop a RAL model, integrate it in a verification environment and verify your design. As such, you should read it in a sequential fashion. You can use the same sequence to create your own RAL model and use it to verify your design.

The DUT

The Design Under Test used in this primer is an AMBA™ Peripheral Bus (APB) slave device. It is a simple single-master device with a few registers and a memory, as described in Table 1. The data bus is 32-bit wide.

Table 1 Address Map

Address	Name
0x0000	CHIP_ID
0x0010	STATUS
0x0014	MASK
0x1000-0x13FF	COUNTERS
0x2000-0x2FFF	DMA RAM

Table 2 through Table 5 define the various fields found in each registers. “RW” indicates a field that can be read and written by the firmware. “RO” indicates a field that can be read but not written by the firmware. “W1C” indicates a field that can be read and written by the firmware, but writing a ‘0’ has no effect and writing a ‘1’ clears the corresponding bit if it is set.

Table 2 CHIP_ID Register

Field	Reserved	PRODUCT_ID	CHIP_ID	REVISION_ID
Bits	31-28	27-16	15-8	7-0
Access	RO	RO	RO	RO
Reset	0x0	0x176	0x5A	0x03

Table 3 STATUS Register

Field	Reserved	READY	Reserved	MODE	TXEN	BUSY
Bits	31-17	16	15-5	4-2	1	0
Access	RO	W1C	RO	RW	RW	RO
Reset	0x0000	0x0	0x0000	0x0	0x0	0x0

Table 4 MASK Register

Field	Reserved	READY	Reserved
Bits	31-17	16	15-0
Access	RO	RW	RO
Reset	0x0000	0x0	0x0000

These registers are statistic counters that are incremented by the DUT under the appropriate circumstance. There are 256 such counters.

Table 5 COUNTER Registers

Field	COUNT
Bits	31-0
Access	RO
Reset	0x0000

Address Granularity

It is important to understand the address granularity of the DUT. The address granularity refers to the minimum number of bytes that can be uniquely addressed. Consider the address of the STATUS and MASK registers, as defined in Table 1. There are two possibilities for interpreting these two addresses.

If the address is specified using a BYTE granularity, the address space of the DUT would look like Table 6, assuming it is LITTLE_ENDIAN.

Table 6 BYTE Address Granularity

Address	Data (32 bits)
0x0010	STATUS[31:0]
0x0011	MASK[7:0], STATUS[31:8]
0x0012	MASK[15:0], STATUS[31:16]
0x0013	MASK[23:0], STATUS[31:24]
0x0014	MASK[31:0]
0x0015	8'h00, MASK[31:8]
0x0016	16'h0000, MASK[31:16]
0x0017	24'h000000, MASK[31:24]

If the address is specified using DWORD (32 bits) granularity, the address space of the DUT would look like Table 7.

Table 7 DWORD Address Granularity

Address	Data (32 bits)
0x0010	STATUS[31:0]
0x0011	Unspecified
0x0012	Unspecified
0x0013	Unspecified
0x0014	MASK[31:0]
0x0015	Unspecified
0x0016	Unspecified
0x0017	Unspecified

Because the data bus is 32 bits, a design with BYTE granularity will often not implement the least significant two bits of the address bus. This effectively shifts the address value left by two bits and creates a DWORD granularity, as illustrated in Table 8.

Table 8 Shifter BYTE Address Granularity

Address[15:2]	Data (32 bits)
0x0004	STATUS[31:0]
0x0005	MASK[31:0]

The DUT used in this primer uses a BYTE granularity but does not implement the two least significant bits. Its address space is thus illustrated by Table 8 and effectively implements a DWORD granularity.

Step 1: The RALF File

The Register Abstraction Layer Format (RALF) file is a specification of the host-accessible registers and memories available in your design. It can be captured by hand from the specification above or it could be automatically generated from a suitably formatted specification document, such as an Excel spreadsheet.

The smallest unit that can be used to represent a design in a RALF description is the *block*. The name of the block should be relevant and somewhat unique. This will allow the RALF description of the block to be included in a RALF description of the system that instantiates it. **Do not** name your block “DUT”—that is simply begging to collide with the name of another block or system similarly badly named.

The **`bytes`** attribute defines the width of the physical data path when accessing registers and memories in the block. RAL assumes that the address granularity is equal to the width of the data path. Since the data path of the DUT is 32-bits, a DWORD address granularity will be assumed.

File: RAL/slave.ralf

```
block slave {
   bytes 4;
   ...
}
```

The registers are declared in the block using a **`register`** description. The COUNTERS registers, being identical and located at consecutive addresses can be specified using a register array. The address offset of each register within the blocks is also specified at the same time.

File: RAL/slave.ralf

```
block slave {
   bytes 4;
   register CHIP_ID @'h0000 {
      ...
   }
   register STATUS @'h0010 {
      ...
   }
   register MASK @'h0014 {
      ...
   }
   register COUNTERS[256] @'h1000 {
      ...
   }
   ...
}
```

If no address offset is specified for a register, it is assumed to be incremented by **one** from the previous register address offset. This creates a problem for the **COUNTERS** register array as the address of each subsequent register in the array is assumed to be incremented by one. Because the block is assumed to have a DWORD granularity, the address offset increment refers to the shifted address value, not the documented BYTE granularity address value. To avoid this problem, the DWORD granularity address values (i.e. the shifted values) should be specified. In some cases, the shifting may have to be undone in the translation transactor (see Step 4).

File: RAL/slave.ralf

```
block slave {
   bytes 4;
   register CHIP_ID @'h0000 {
      ...
   }
   register STATUS @'h0004 {
      ...
   }
   register MASK @'h0005 {
      ...
```

```
    }
    register COUNTERS[256] @'h0400 {
       ...
    }
    ...
}
```

The fields inside each register are then specified using a *field* description. A field is the smallest of information and describes a set of consecutive bits with identical behavior. It is not necessary to specify unused or reserved bits if they are read-only and read as zeroes. Fields are assumed to be contiguous, and justified in the least significant bits. Fields can be positioned at a specific bit offset within a register by specifying the bit number in the register that corresponds to the least significant bit of the field. The last remaining element to be specified is the DMA RAM.

File: RAL/slave.ralf

```
block slave {
   bytes 4;
   register CHIP_ID @'h0000 {
      field REVISION_ID {
         bits 8;
         access ro;
         reset 'h03;
      }
      field CHIP_ID {
         bits 8;
         access ro;
         reset 'h5A;
      }
      field PRODUCT_ID {
         bits 10;
         access ro;
         reset 'h176;
      }
   }
   register STATUS @'h0004 {
      field BUSY (BUSY) {
         bits 1;
         access ro;
```

```
            reset 'h0;
         }
         field TXEN (TXEN) {
            bits 1;
            access rw;
            reset 'h0;
         }
         field MODE (MODE) {
            bits 3;
            access rw;
            reset 3'h0;
         }
         field READY (RDY) @16 {
            bits 1;
            access w1c;
            reset 'h0;
         }
      }
      register MASK @'h0005 {
         field READY (RDY_MSK) @16 {
            bits 1;
            access rw;
            reset 'h0;
         }
      }
      register COUNTERS[256] @'h0400 {
         field value {
            bits 32;
            access ru;
            reset 'h0;
         }
      }
      memory DMA_RAM (DMA) @'h0800 {
         size 1k;
         bits 32;
         access rw;
      }
}
```

When **(HDL_PATH)** is specified for registers and memories, backdoor access code for those registers and memories can be automatically added to the RAL model. The backdoor access code allows the model to directly access registers and memories without

going through the physical interface bus functional model. Therefore, the registers and the memories can be read or written without penalty of simulation cycles.

File: RAL/slave.ralf

```
block slave {
   bytes 4;
   register CHIP_ID @'h0000 {
    ...
   }
   register STATUS @'h0004 {
      field BUSY (BUSY) {
         ...
      }
      field TXEN (TXEN) {
         ...
      }
      field MODE (MODE) {
         ...
      }
      field READY (RDY) @16 {
         ...
      }
   }
   register MASK @'h0005 {
      field READY (RDY_MSK) @16 {
         ...
      }
   }
      ...
   memory DMA_RAM (DMA) @'h0800 {
      ...
   }
}
```

Step 2: Model Generation

Once the registers and memories have been specified in a RALF file, the **`ralgen`** script is used to generate the corresponding RAL model. The following command will generate a SystemVerilog RAL model of the **`slave`** block in the file **`ral_slave.sv`**:

Command:

```
% ralgen -b -l sv -t slave slave.ralf
```

The generated code is not designed to be read or subsequently manually modified. However, the structure of the generated RAL model will be outlined to demonstrate how it mirrors the RALF description. The following generated RAL model corresponds to the documented output. All other lines not shown are not explicitly documented and should not be relied upon.

RAL/ral_slave.sv

```
class ral_reg_slave_CHIP_ID extends vmm_ral_reg;
   rand vmm_ral_field REVISION_ID;
   rand vmm_ral_field CHIP_ID;
   rand vmm_ral_field PRODUCT_ID;
   ...
endclass : ral_reg_slave_CHIP_ID
...
class ral_reg_slave_STATUS extends vmm_ral_reg;
   rand vmm_ral_field BUSY;
   rand vmm_ral_field TXEN;
   rand vmm_ral_field MODE;
   rand vmm_ral_field READY;
   ...
endclass : ral_reg_slave_STATUS
...
class ral_reg_slave_MASK extends vmm_ral_reg;
   rand vmm_ral_field READY;
   ...
```

```
endclass : ral_reg_slave_MASK

class ral_reg_slave_COUNTERS extends vmm_ral_reg;
   rand vmm_ral_field value;
   ...
endclass: ral_reg_slave_COUNTERS

class ral_block_slave extends vmm_ral_block;
   rand ral_reg_slave_CHIP_ID CHIP_ID;
   rand vmm_ral_field REVISION_ID,CHIP_ID_REVISION_ID;
   rand vmm_ral_field CHIP_ID_CHIP_ID;
   rand vmm_ral_field PRODUCT_ID,CHIP_ID_PRODUCT_ID;

   rand ral_reg_slave_STATUS STATUS;
   rand vmm_ral_field BUSY,STATUS_BUSY;
   rand vmm_ral_field TXEN,STATUS_TXEN;
   rand vmm_ral_field MODE,STATUS_MODE;
   rand vmm_ral_field STATUS_READY;

   rand ral_reg_slave_MASK MASK;
   rand vmm_ral_field MASK_READY;

   rand ral_reg_slave_COUNTERS COUNTERS[256];
   rand vmm_ral_field value[256],COUNTERS_value[256];

   rand ral_mem_slave_DMA_RAM DMA_RAM;

   function new(int cover_on = vmm_ral::NO_COVERAGE, ... );
      ...
   endfunction: new
endclass: ral_block_slave
```

The first thing to notice about the RAL model is the abstraction class that corresponds to the block. The block abstraction class contains a property for each register in the block that refers to an abstraction class for that register. The register array is modeled using an array of abstraction classes.

RAL/ral_slave.sv

```
...
class ral_block_slave extends vmm_ral_block;
   rand ral_reg_slave_CHIP_ID CHIP_ID;
```

```
   ...
   rand ral_reg_slave_STATUS STATUS;
   ...
   rand ral_reg_slave_MASK MASK;
   ...
   rand ral_reg_slave_COUNTERS COUNTERS[256];
   ...
   rand ral_mem_slave_DMA_RAM DMA_RAM;
   ...
endclass: ral_block_slave
```

Similarly, the register abstraction class for a register contains a property for each field it contains. There is also a property for each field in a register in the block abstraction class. This allows fields to be referenced without regards to their location in a specific register, thus allowing them to be relocated without having to modify the code that use them. However, this requires that the field name be unique within the block. For example, because the field named `CHIP_ID` in the register named `CHIP_ID` conflicts with the register of the same name, there is no class property named `CHIP_ID` for the field in the block abstraction class. Similarly, because there are two fields named `READY` in different registers, there are no class properties of that name in the block abstraction class.

RAL/ral_slave.sv

```
class ral_reg_slave_CHIP_ID extends vmm_ral_reg;
   rand vmm_ral_field REVISION_ID;
   rand vmm_ral_field CHIP_ID;
   rand vmm_ral_field PRODUCT_ID;
   ...
endclass : ral_reg_slave_CHIP_ID
...
class ral_block_slave extends vmm_ral_block;
   rand ral_reg_slave_CHIP_ID CHIP_ID;
   rand vmm_ral_field REVISION_ID,CHIP_ID_REVISION_ID;
   rand vmm_ral_field CHIP_ID_CHIP_ID;
   rand vmm_ral_field PRODUCT_ID,CHIP_ID_PRODUCT_ID;
   ...
   endclass: ral_block_slave
```

Every class property in the abstraction classes has the *rand* attribute. This allows the content of a RAL model to be randomized. However, this attribute is turned off by default in all fields unless a constraint—even an empty one—has been specified for that field.

File: RAL/slave.ralf

```
block slave {
   ...
   register STATUS @'h0004 {
      ...
      field TXEN (TXEN) {
         bits 1;
         access rw;
         reset 'h0;
         constraint valid {}
      }
      field MODE (MODE) {
         bits 1;
         access rw;
         reset 3'h0;
         constraint valid {
         value < 3'h6;
         }
      }
      ...
   constraint status_reg_valid {
         (MODE.value == 3'h5) -> TXEN.value != 1'b1;
   }
   }
   ...
}
```

For every register and memory with *(*`HDL_PATH`*)* specified, a backdoor access class is automatically generated. Each class contains **`virtual task read()`** and **`virtual task write()`** for backdoor read and write access. Both tasks utilize the compiler directive **`` `SLAVE_TOP_PATH ``** to define the hierarchical path to registers in the DUT.

File: RAL/ral_slave.sv

```
...
class ral_reg_slave_STATUS_bkdr extends
vmm_ral_reg_backdoor;
   vitual task read(output vmm_rw::status_e status, ...);
    begin
      data = 'VMM_RAL_DATA_WIDTH'h0;
      data[0:0] = 'SLAVE_TOP_PATH.BUSY;
      data[1:1] = 'SLAVE_TOP_PATH.TXEN;
      data[4:2] = 'SLAVE_TOP_PATH.MODE;
      data[16:16] = 'SLAVE_TOP_PATH.RDY;
    end
    status = vmm_rw::IS_OK;
   endtask

   vitual task write(output vmm_rw::status_e status, ...);
    begin
      'SLAVE_TOP_PATH.TXEN = data[1:1];
      'SLAVE_TOP_PATH.MODE = data[4:2];
      'SLAVE_TOP_PATH.RDY = data[16:16];
    end
    status = vmm_rw::IS_OK;
   endtask
endclass
...
```

Lastly, the constructor for the block abstraction class has a default argument **`vmm_ral::NO_COVERAGE`**. By default, no functional coverage model is included.

File: RAL/ral_slave.sv

```
...
class ral_block_slave extends vmm_ral_block;
   ...
   function new(int cover_on = vmm_ral::NO_COVERAGE, ...);
      ...
   endfunction: new
endclass: ral_block_slave
```

Note that the constructors for the other abstraction classes are not shown. That is because they are not documented and not intended to be used directly. Only block and system abstraction classes are intended as end-user RAL models.

Step 3: Top-Level Module

The DUT must be instantiated in a top-level module and connected to protocol-specific interfaces corresponding to the command-level transactors that drive and monitor the DUT's signals. The DUT and the relevant interfaces are instantiated in a top-level **`module`** (Rule 4-13). The connection to the DUT pins are specified using a hierarchical reference to the **`wires`** in the **`interface`** instance. Notice how the two least significant bits of the address are not used by the DUT to implement the BYTE granularity with a DWORD data bus.

File: RAL/tb_top.sv

```
module tb_top;
   ...
   apb_if apb0(...);

   slave_ip dut_slv(.apb_addr   (apb0.paddr[15:2] ),
                    .apb_sel    (apb0.psel        ),
                    .apb_enable (apb0.penable     ),
                    .apb_write  (apb0.pwrite      ),
                    .apb_rdata  (apb0.prdata[31:0]),
                    .apb_wdata  (apb0.pwdata[31:0]),
                    ...);
   ...
endmodule: tb_top
```

This top-level module also contains the clock generators (Rule 4-15) and reset signal, using the **`bit`** type (Rule 4-17). The clock generator ensures that no clock edges will occur at time zero (Rule 4-16).

File: RAL/tb_top.sv

```
module tb_top;
   bit clk = 0;
   bit rst = 0;

   apb_if apb0(clk);
   ...

   slave_ip dut_slv(.apb_addr   (apb0.paddr[15:2] ),
                    .apb_sel    (apb0.psel        ),
                    .apb_enable (apb0.penable     ),
                    .apb_write  (apb0.pwrite      ),
                    .apb_rdata  (apb0.prdata[31:0]),
                    .apb_wdata  (apb0.pwdata[31:0]),
                    .clk        (clk),
                    .rst        (rst));

   always #10 clk = ~clk;
endmodule: tb_top
```

Step 4: Physical Interface

A RAL model is not aware of the physical interface used to access the registers and memories. It issues generic read and write transaction requests at specific addresses but these generic transactions need to be executed on whatever physical interface is provided by the DUT.

The translation must be accomplished in a user-defined extension of the **`vmm_rw_xactor::execute_single()`** task in a transactor extended from the **`vmm_rw_xactor`** base class—which is itself based on the **`vmm_xactor`** base class. This task can use any transactor to execute the requested generic transactions.

File: RAL/apb_rw_xlate.sv

```
...
class apb_rw_xlate extends vmm_rw_xactor;
```

```
   ...
   virtual task execute_single(vmm_rw_access tr);
      ...
   endtask: execute_single
endclass: apb_rw_xlate
```

The first thing that is needed is a command-level transactor to execute the read and write transactions. Pass a reference to an instance of a suitable command-level transactor via the constructor argument. To ensure that the command-level transactor is started when the translation transactor is started, its **`start_xactor()`** method must be called in the extension of the translation transactor's **`start_xactor()`** own method.

File: RAL/apb_rw_xlate.sv

```
`include "apb_master.sv"
...
class apb_rw_xlate extends vmm_rw_xactor;
   apb_master bfm;

   function new(string        inst,
                int unsigned  stream_id,
                apb_master          bfm);
      super.new("APB RAL Master", inst, stream_id);
      this.bfm = bfm;
   endfunction: new

   virtual function void start_xactor();
      super.start_xactor();
      this.bfm.start_xactor();
   endfunction
   ...
   virtual task execute_single(vmm_rw_access tr);
      ...
   endtask: execute_single
endclass: apb_rw_xlate
```

The generic transaction is then translated into an equivalent transaction suitable for the command-level transactor used. Note that it may be necessary to adjust the address specified by the RAL

model in the generic transaction to the physical address used by the physical protocol. In our case, because paddr[1:0] is not used by the DUT (because it uses BYTE granularity addressing with a DWORD data bus), you must shift the specified address into paddr[31:2]. Once the transaction has been executed according to the translator execution model, the status and the read-back data (if applicable) is annotated onto the generic transaction before the **`execute_single()`** method is allowed to return.

File: RAL/apb_rw_xlate.sv

```
`include "apb_master.sv"
`include "vmm_ral.sv"

class apb_rw_xlate extends vmm_rw_xactor;
   apb_master bfm;

   function new(string        inst,
                int unsigned  stream_id,
                apb_master          bfm);
      super.new("APB RAL Master", inst, stream_id);
      this.bfm = bfm;
   endfunction: new

   virtual function void start_xactor();
      super.start_xactor();
      this.bfm.start_xactor();
   endfunction

   virtual task execute_single(vmm_rw_access tr);
      apb_rw cyc = new;

      // DUT uses BYTE granularity addresses
      // but with a DWORD datapath
      cyc.addr = {tr.addr, 2'b00};
      if (tr.kind == vmm_rw::WRITE) begin
         // Write cycle
         cyc.kind = apb_rw::WRITE;
         cyc.data = tr.data;
      end
      else begin
         // Read cycle
         cyc.kind = apb_rw::READ;
```

```
      end

      this.bfm.in_chan.put(cyc);

      if (tr.kind == vmm_rw::READ) begin
         tr.data = cyc.data;
      assert(!($isunknown(tr.data)))
         tr.status = vmm_rw::IS_OK;
      else begin
         `vmm_error(log, "Data Contains X Value");
         tr.status = vmm_rw::ERROR;
      end
   endtask: execute_single
endclass: apb_rw_xlate
```

The creation of the translation transactor can be simplified by using the template provided by the **vmmgen** tool.

Command

```
% vmmgen -l sv
```

The relevant templates for writing translation transactors are:

- RAL physical access BFM, single domain
- RAL physical access BFM, multiplexed domains

Note that the menu number used to select the appropriate template may differ from the number in the above list. The style used to implement the translation transactor shown in this primer is provided by the “multiplexed domains” template.

Step 5: Verification Environment

A RAL model must be used with a verification environment class extended from the **`vmm_ral_env`** base class. The RAL model is instantiated in the environment constructor then registered with the base class using the **`ral.set_model()`** method. The RAL model is instantiated in the constructor so it can be used to generate a suitable configuration in the **`gen_cfg()`** step.

File: RAL/tb_env.sv

```
`ifndef TB_ENV__SV
`define TB_ENV__SV

`include "vmm.sv"
`include "vmm_ral.sv"
`include "apb.sv"
`include "ral_slave.sv"
...

class tb_env extends vmm_ral_env;
   ral_block_slave ral_model;
   ...
   function new();
      ...
      ral_model = new(vmm_ral::NO_COVERAGE);
      ...
      super.ral.set_model(this.ral_model);
   endfunction: new
   ...
endclass: tb_env

`endif
```

Via constraint definitions in the RALF description, RAL model randomization has been enabled. Therefore, DUT configuration can be randomized in the **`gen_cfg()`** step.

File: RAL/tb_env.sv

```
...
class tb_env extends vmm_ral_env;
   ...
   function void gen_cfg();
      ...
      if (!(this.ral_model.randomize() ))
       `vmm_error(this.log, "ral_model could not be \n
                  randomized");
    endfunction: gen_cfg
...
endclass: tb_env
```

The translation transactor and its required command-layer transactor are constructed in the extension of the **`vmm_ral_env::build()`** method (Rule 4-34, 4-35). The translation transactor must then be registered using the **`ral.add_xactor()`** method. The transactors will be automatically started by RAL but it does not hurt to start them again in the extension of the **`vmm_env::start()`** method (Rule 4-41).

File: RAL/tb_env.sv

```
`ifndef TB_ENV__SV
`define TB_ENV__SV

`include "vmm.sv"
`include "vmm_ral.sv"
`include "apb.sv"
`include "ral_slave.sv"
`include "apb_rw_xlate.sv"

class tb_env extends vmm_ral_env;
   ral_block_slave ral_model;

   apb_master   mst;
   apb_rw_xlate ral2apb;

   function new();
      ...
      ral_model = new(vmm_ral::NO_COVERAGE);
      ...
```

```
      super.ral.set_model(this.ral_model);
   endfunction: new

   virtual function void build();
      super.build();

      this.mst = new("APB", 0, tb_top.apb0);
      this.ral2apb = new("APB", 0, this.mst);
      this.ral.add_xactor(this.ral2apb);
   endfunction: build
   ...
endclass: tb_env

`endif
```

Instead of specifying how to reset the DUT in the extension of the **`vmm_ral_env::reset_dut()`** method, it is specified in the extension of the **`vmm_ral_env::hw_reset()`** task. This task will be called by the default implementation of the **`vmm_ral_env::reset_dut()`** method, thus satisfying Rule 4-30.

File: RAL/tb_env.sv

```
`ifndef TB_ENV__SV
`define TB_ENV__SV

`include "vmm.sv"
`include "vmm_ral.sv"
`include "apb.sv"
`include "ral_slave.sv"
`include "apb_rw_xlate.sv"

class tb_env extends vmm_ral_env;
   ral_block_slave ral_model;


   apb_master   mst;
   apb_rw_xlate ral2apb;

   function new();
      ...
      ral_model = new(vmm_ral::NO_COVERAGE);
      ...
```

```
      super.ral.set_model(this.ral_model);
   endfunction: new

   virtual function void build();
      super.build();

      this.mst = new("APB", 0, tb_top.apb0);
      this.ral2apb = new("APB", 0, this.mst);
      this.ral.add_xactor(this.ral2apb);
   endfunction: build

   virtual task hw_reset();
      tb_top.rst <= 1'b1;
      repeat (3) @ (negedge tb_top.clk);
      tb_top.rst <= 1'b0;
      repeat (3) @ (negedge tb_top.clk);
   endtask: hw_reset
   ...
endclass: tb_env

`endif
```

Finally, in the **`cfg_dut()`** step, update the DUT to reflect the randomized RAL model values. Recall that the RAL model was randomized in the **`gen_cfg()`** step.

File: RAL/tb_env.sv

```
...
class tb_env extends vmm_ral_env;
   ...
   virtual task cfg_dut();
      ...
      ral_model.update(status, vmm_ral::BACKDOOR);
    endfunction: cfg_dut
...
endclass: tb_env
```

Step 6: The Pre-Defined Tests

Before you can run one of the pre-defined tests, you must specify which files must be included in the simulation first and what is the name of the verification environment class. This is done by the file `ral_env.svh` in the current working directory and defining the **`RAL_TB_ENV** macro respectively.

File: RAL/ral_env.svh

```
`define RAL_TB_ENV tb_env

`include "tb_env.sv"
```

You are now ready to execute any of the pre-defined tests! It is best to start with the simplest test: applying hardware reset then reading all of the registers to verify their reset values. Many of the problems with the DUT, the RAL model, or the integration of the two will be identified by this simple test. Notice the compiler directive **SLAVE_TOP_PATH** which will be needed for backdoor access codes used in **mem_walk**, **mem_access**, etc.

Command

```
% vcs -R -sverilog -ntb_opts rvm+dtm \
     +incdir+../apb +define+SLAVE_TOP_PATH=tb_top.dut
tb_top.sv \
     $VCS_HOME/etc/vmm/sv/RAL/tests/hw_reset.sv
```

Other tests are provided with RAL. They can all be found in the `$VCS_HOME/etc/vmm/sv/RAL/tests` directory. The *RAL User Guide* details the functionality of each test. Note that the pre-defined tests are available as unencrypted source code. Thus they can be modified to meet to particular needs of your design or they can be used as a source of inspiration for writing other RAL-based tests.

Step 7: Coverage Model

A functional coverage model can be added to the RAL model by using an option in the *ralgen* script.

Command

```
% ralgen -c b -b -l sv -t slave slave.ralf
```

The generated coverage model can be large (4 bins per bit in every field). When functional coverage is added it should be pruned to improve memory usage and run-time performance once the register implementation has been verified and the coverage model filled to satisfaction.

File: RAL/ral_slave.sv

```
...
class ral_cvr_reg_slave_STATUS;
...
  TXEN: coverpoint {data[1:1], is_read} {
    wildcard bins bit_0_wr_as_0 = {2'b00};
    wildcard bins bit_0_wr_as_1 = {2'b10};
    wildcard bins bit_0_rd_as_0 = {2'b01};
    wildcard bins bit_0_rd_as_1 = {2'b11};
    option.weight = 4;
  }
...
endclass
...
```

The coverage model can be enabled by passing the appropriate argument to the RAL model constructor. This will enable VCS to create and collect a coverage database. A report can then be generated using utilities such as the Unified Report Generator.

File: RAL/tb_env.sv

```
class tb_env extends vmm_ral_env;
...
  function new();
...
    ral_model = new(vmm_ral::REG_BITS);
    super.ral.set_model(this.ral_model);
    ...
  endfunction: new
...
endclass
...
```

Step 8: Congratulations!

You have now completed the integration of a RAL model with a design and were able to verify the correct implementation of all registers and memories!

A specific DUT configuration may be desirable to complete functional coverage. The RAL model provides easy access to write the appropriate values in the DUT registers through functions and tasks. This can be implemented by creating a user defined test to achieve the desired DUT configuration.

File: RAL/user_test.sv

```
program user_test;
...
env.cfg_dut();
   ...
//   writing registers with frontdoor access
      env.ral_model.MODE.write(status, 3'h3);
      env.ral_model.TXEN.write(status, 1'h1);
      env.ral_model.MASK_READY.write(status, 1'h1);
   ...
// reading registers with backdoor access
      env.ral_model.MODE.peek(status, reg_value);
      `vmm_note(log, $psprintf("... ", reg_value));
```

```
      env.ral_model.TXEN.peek(status, reg_value);
      `vmm_note(log, $psprintf("... ", reg_value));
      env.ral_model.MASK_READY.peek(status, reg_value);
      `vmm_note(log, $psprintf("... ", reg_value));
...
endprogram
```

There are many other VMM Primers that cover topics such as how to write VMM-compliant command-layer transactors, functional-layer transactors or verification environments.