VMM PRIMER

Writing Command-Layer Slave Transactors

Author(s):

Janick Bergeron

Version 0.4 / May 17, 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.

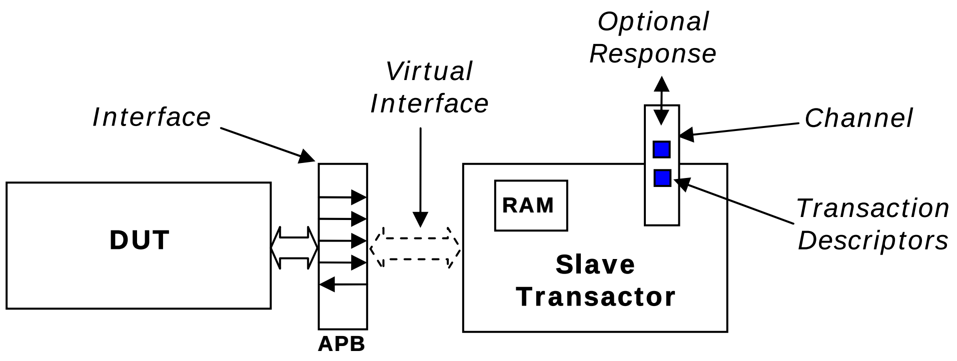

Figure 1

The complete source code for this primer can be found in the *Command_Slave_Xactor* directory. To run the example, simply type “make”.

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. In fact, it is possible to implement the APB bus using a single data bus (see section 5.6.4 of the AMBA™ Specification (Rev 2.0)). 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: 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: 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: 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: 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 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
...
`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 serial 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 (Rule 4-101).

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: 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: 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: 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: 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: 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 (Rule 4-156, 4-158) and after the transaction response has completed (Rule 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: 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 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
```

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: 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 return, the transaction descriptor contains the data to return to the DUT. 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: 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: 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: 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 useable 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 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)
`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: 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 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 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:

1) Physical interface declaration
2) Transaction Descriptor
3) 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.