VMM PRIMER

Using the Register Abstraction Layer

Author(s):

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

Tables 2 through 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

Table 5: COUNTER Registers

These registers are statistic counters that are incremented by the DUT under the appropriate circumstance. There are 256 such counters.

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 2 bits of the address bus. This effectively shifts the

address value left by 2 bits and creates a DWORD granularity, as illustrated in Table 8.

Table 8: Shifted 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 2 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 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's 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.peak(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.