PRIMER

Using the Register Layer in UVM

Author(s):

Janick Bergeron

Based on work by:

Janick Bergeron
John Choi
Brett Kobernat

Version 1.3 / Jan 13, 2011

Introduction

The UVM Register Layer is set of base classes 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. It may also be used to implement 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 register model of the registers and memories in a design, how to integrate this model in a UVM verification environment and how to verify the implementation of those registers and memories using the pre-defined tests. It will also show how the register 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 register model is used to implement additional functional tests.

This primer assumes that you are familiar with UVM. It also assumes that the register model is written manually: this will seem initially odd as register models were designed to be automatically generated and thus the coding style was not designed to be congenial as a register modeling style. The language used to specify the registers and memories is outside the scope of this primer.

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 register library. The DUT has enough features to show the steps needed to create a register model to verify the design.

This document is written in the same order you would develop a register 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 register model and use it to verify your design.

The source code for the primer can be found in the following directory:

```
$UVM_HOME/examples/registers/primer
```

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

0x0020	INDEX
0x0024	DATA
0x1000-0x1FFF	SESSION[256]
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.

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: INDEX Register

Field	Reserved	Index
Bits	31-8	7-0
Access	RO	RW
Reset	0x0000	0x0000

Table 4: DATA Register

The DATA register is an indirect register. It accesses one of the TABLE 32-bit registers, as specified by the value in the INDEX register. For example, reading the DATA register while the INDEX register contains the value “7”, the value of TABLE[7] is read, and writing the DATA register while the INDEX register contains the value “35”, the value of TABLE[35] is written.

Field	Data
Bits	31-0
Access	RW

Table 5: TABLE[256] Registers

The TABLE registers are not directly visible in the DUT address space. They are accessed indirectly via the INDEX and DATA registers.

Field	TABLE[INDEX]
Bits	31-0
Access	RW

Table 6: SESSION[256] Register File

The SESSION register file array contained 256 instances of the SRC and DST registers.

Address	Register
0x1000	SESSION[0].SRC
0x1008	SESSION[0].DST
0x1010	SESSION[1].SRC
0x1018	SESSION[1].DST
...	...
0x1FF0	SESSION[256].SRC
0x1FF8	SESSION[255].DST

Table 7: SRC and DST Registers

Field	PORT	IP
Bits	63-48	47-0
Access	RW	RW

Step 1: Register Model

IMPORTANT: this step is normally automatically performed by the code generator.

The first step is to create the abstract model of the registers in the DUT. This primer details the register model of the CHIP_ID register. The model for the other registers can be similarly constructed and are left as an exercise to the reader. Refer to Section 9.5 of the UVM User's Guide for more details.

A class, based on the *uvm_reg* class, models the register type. It contains an instance of the *uvm_reg_field* class for each field the register contains. The name of the register model class is technically arbitrary but it is recommended that it reflect the name or type of the register.

The name of each field instance handle is technically arbitrary but it is recommended that they be named according to the name of the respective field in the register specification. Furthermore, they are specified as public and with a *rand* attribute (where relevant) so their value may be later randomized and constrained.

The constructor must follow the standard *uvm_object* constructor style, with an instance name. It simply passes that argument and type-specific configuration parameters to the parent constructor.

A *build()* method must be implemented. Inside the method, each instance of the field class is created using the UVM class factory. Each field is then configured by calling its *configure()* method. This defines the field's access properties, relative position in the register, etc…

File: reg_slave.sv

```
class reg_slave_ID extends uvm_reg_reg;
   uvm_reg_field REVISION_ID;
   uvm_reg_field CHIP_ID;
   uvm_reg_field PRODUCT_ID;

   function new(string name);
      super.new(name, 32, UVM_NO_COVERAGE);
   endfunction

   function void build();
      REVISION_ID = uvm_reg_field::create("REVISION_ID",null,
                                          get_full_name());
      REVISION_ID.configure(…);
      CHIP_ID     = uvm_reg_field::create("CHIP_ID",null,
                                          get_full_name());
      CHIP_ID.configure(…);
      PRODUCT_ID  = uvm_reg_field::create("PRODUCT_ID" ",null,
                                          get_full_name());
      PRODUCT_ID.configure(…);
   endfunction

   function void configure();|
      super.configure
   endfunction
endclass : reg_slave_ID
```

Step 2: Memory Model

IMPORTANT: this step is normally automatically performed by the code generator.

The next step is to create the abstract model of the memories in the DUT.

A class, based on the *uvm_mem* class, models the memory type. The name of the memory model class is technically arbitrary but it is recommended that it reflect the name or type of the memory.

The constructor must follow the standard *uvm_object* constructor style, with an instance name. It simply passes that argument and type-configuration parameters to the parent constructor.

File: reg_slave.sv

```
class mem_slave_DMA_RAM extends uvm_reg_mem;

   function new(string name);
      super.new(name, 'h400, 32, "RW", UVM_NO_COVERAGE);
   endfunction

endclass: mem_slave_DMA_RAM
```

Step 3: DUT Model

IMPORTANT: this step is normally automatically performed by the code generator.

The next step is to create the abstract model of all the register in the DUT.

A class, based on the *uvm_reg_block* class, models the all the registers and memories in the DUT. The name of the register model class is technically arbitrary but it is recommended that it reflect the name or type of the model. The register model abstraction class contains a property for each register and memory that refers to an instance of the register or memory abstraction class of the appropriate type for that register. Registers arrays are modeled using arrays of abstraction classes.

The name of each register and memory instance handle is technically arbitrary but it is recommended that they be named according to the name of the respective register and memory in the address map specification. Furthermore, they are specified as public and with a *rand* attribute (where relevant) so their value may be later randomized and constrained. It may also optionally contain references to field instances. These allow fields to be accessed independently of the registers they are located in.

The constructor must follow the standard *uvm_object* constructor style, with an instance name. It simply passes that argument and type-specific configuration parameters to the parent constructor.

File: reg_slave.sv

```
class reg_block_slave extends uvm_reg_block;
   reg_slave_ID      ID;
   ...
```

```
   mem_slave_DMA_RAM DMA_RAM;

   function new(string name);
      super.new(name, UVM_NO_COVERAGE);
   endfunction
   ...
endclass: reg_block_slave
```

A *build()* method must be implemented. Inside the method, each instance of the register classes is created using the UVM class factory. Each register abstraction class is then configured by calling its *configure()* method then built by calling its *build()* method. This instantiates the register abstraction classes and configures them at the proper address offset, access policy, etc…

File: reg_slave.sv

```
class reg_block_slave extends uvm_reg_block;
   reg_slave_ID      ID;
   ...
   mem_slave_DMA_RAM DMA_RAM;

   function new(string name);
      super.new(name, UVM_NO_COVERAGE);
   endfunction

   function build();
      ID = reg_slave_ID::type_id::create("ID");
      ID.configure(this, null);
      ID.build();
      ...
      DMA_RAM = mem_slave_DMA_RAM::type_id::create("DMA_RAM");
      DMA_RAM.configure(this);
      DMA_RAM.build();
      ...
   endfunction
endclass: reg_block_slave
```

Step 4: Physical Interface

IMPORTANT: this step is performed by the integrator.

A register model is not aware of the physical interface used to access the registers and memories. It issues abstract register operations at specific addresses but these abstract operations need to be executed on whatever physical interface is provided by the DUT.

An adapter class must be provided to translate between the abstract register operation issued by the register model and the physical transaction executed by the bus sequencer. The adapter must be extended from the *uvm_reg_adapter* class and implement the *reg2bus()* conversion method.

File: ../common/apb/apb_rw.sv

```
...
class reg2apb_adapter extends uvm_reg_adapter;

  `uvm_object_utils(reg2apb_adapter)

   function new(string name = "reg2apb_adapter");
      super.new(name);
   endfunction

   virtual function uvm_sequence_item
             reg2bus(const ref uvm_reg_bus_op rw);

      apb_rw apb = apb_rw::type_id::create("apb_rw");

      apb.kind =
         (rw.kind == UVM_READ) ? apb_rw::READ : apb_rw::WRITE;
      apb.addr = rw.addr;
      apb.data = rw.data;

      return apb;
   endfunction

endclass
```

Step 5: Top-Level Module

IMPORTANT: this step is performed by the integrator.

The DUT must be instantiated in a top-level module and connected to protocol-specific interfaces corresponding to the agent that drives and monitors the DUT's signals. The DUT and the relevant interfaces are instantiated in a top-level *module*.

File: tb_top.sv

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

   slave dut(apb0, ...);
   ...
endmodule: tb_top
```

This top-level module also contains the clock generators and reset signal.

File: tb_top.sv

```
module tb_top;
   bit clk = 0;
   bit rst = 0;

   apb_if apb0(clk);
   slave dut(apb0, rst);

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

Step 6: Verification Environment

IMPORTANT: this step is performed by the integrator.

The register model is instantiated in the environment's *build()* method using the UVM class factory. To enable vertical reuse, the register model is only instantiated if it has not already been specified from a higher-level environment.

File: tb_env.sv

```
class tb_env extends uvm_component;

   `uvm_component_utils(tb_env)

   reg_block_slave model;
   apb_agent apb;

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
   endfunction

   virtual function void build();
      if (model == null) begin
         model = reg_block_slave::type_id::create("model",this);
         model.build();
         model.lock_model();
      end
      apb = apb_agent::type_id::create("apb", this);
   endfunction
   ...
endclass.
```

The next step is to associate the bus sequencer with the corresponding address map in the register model. This sequencer will provide access to the DUT's physical interfaces. The bus sequencer is associated, along with the required adapter class,

with its corresponding address map in the register model in the *connect()* method of the verification environment. This association is only performed if the register model is not embedded in a higher-level register model.

File: tb_env.sv

```
class tb_env extends uvm_component;

   `uvm_component_utils(tb_env)

   reg_block_slave model;
   apb_agent apb;

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
   endfunction

   virtual function void build();
      if (model == null) begin
         model = reg_block_slave::type_id::create("model",this);
         model.build();
         model.lock_model();
      end
      apb = apb_agent::type_id::create("apb", this);
   endfunction

   virtual function void connect();
      if (regmodel.get_parent() == null) begin
         reg2apb_adapter reg2apb = new;
         model.default_map.set_sequencer(apb.sqr, reg2apb);
         model.default_map.set_auto_predict(1);
      end
   endfunction

endclass.
```

Note how the implicit response prediction is enabled using the *set_auto_predict()* method. This causes the register model to predict the content of the registers based on the read and write operations it performs. It is the simplest prediction mode albeit not as reusable. The explicit prediction mode requires the additional integration of a monitor, which is currently outside the scope of this primer. Using a monitor updates the register model based on all observed read and write operations, not just those performed through the register model.

Step 7: The Test Runner

IMPORTANT: this step is performed by the integrator.

A *program* or *module* is required to instantiate the top-level environment, bind the virtual interfaces to the agents and run the actual test.

File: test.sv:

```
program test;

import uvm_pkg::*;
import apb_pkg::*;

`include "reg_model.sv"
`include "tb_env.sv"
`include "testlib.sv"
...
initial
begin
   static tb_env env = new("env");

   uvm_config_db#(apb_vif)::set(env, "apb", "vif",
                                $root.tb_top.apb0);
   run_test();
end
endprogram
```

The DUT needs to be reset before tests can proceed. A DUT reset sequence is defined in the test runner module or program. This allows the reset sequence to access and drive the appropriate signals in the top-level module.

File: test.sv:

```
program test;

import uvm_pkg::*;
import apb_pkg::*;

`include "reg_model.sv"
`include "tb_env.sv"
`include "testlib.sv"

class dut_reset_seq extends uvm_sequence;

   function new(string name = "dut_reset_seq");
      super.new(name);
   endfunction

   `uvm_object_utils(dut_reset_seq)

   virtual task body();
      tb_top.rst = 1;
      repeat (5) @(negedge tb_top.clk);
      tb_top.rst = 0;
   endtask
```

```
endclass

initial
begin
   static tb_env env = new("env");

   uvm_config_db#(apb_vif)::set(env, "apb", "vif",
$root.tb_top.apb0);
   run_test();
end
endprogram
```

Step 8: The Pre-Defined Sequences

IMPORTANT: this step is performed by the integrator.

A test that runs a pre-defined sequence specified on the command line is a simple way of managing the execution of the predefined register test sequences. After execution the DUT reset sequence and resetting the register model, a sequence whose type name is specified on the command-line using the *+UVM_REG_SEQ* command line option is created and executed.

File: cmdline_test:

```
class cmdline_test extends uvm_test;
   ...
   virtual task run();
      tb_env env;
      $cast(env, uvm_top.find("env"));

      begin
         uvm_sequence_base rst_seq;
         rst_seq = dut_reset_seq::type_id::create("rst_seq", this);
         rst_seq.start(null);
      end
      env.model.reset();

      begin
         uvm_cmdline_processor opts =
            uvm_cmdline_processor::get_inst();

         uvm_reg_sequence  seq;
         string            seq_name;

         opts.get_arg_value("+UVM_REG_SEQ=", seq_name);

         if (!$cast(seq, factory.create_object_by_name(seq_name,
                                                 get_full_name(),
                                                     "seq"))
             || seq == null) begin
```

```
            `uvm_fatal("TEST/CMD/BADSEQ", {"Sequence ", seq_name,
                                           " is not a known sequence"})
         end
         seq.model = env.model;
         seq.start(null);
      end
      global_stop_request();
   endtask : run
endclass : cmdline_test
```

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.

Command:

```
% <sim> +UVM_TESTNAME=cmdline_test \
        +UVM_REG_SEQ=uvm_reg_hw_reset_test
```

Step 9: User-Defined Sequences

IMPORTANT: this step is performed by the verification engineer(s).

Congratulations! You are now ready to execute your actual tests! The test sequence can access the registers and memories in the design by using the *read()* and *write()* methods in the register model. To be vertically reusable, a test sequence should extended from the *uvm_reg_sequence* class and obtain its register model from the *model* class property.

File: user_test.sv

```
class user_test_seq extends uvm_sequence;

   function new(string name="user_test_seq");
      super.new(name);
   endfunction : new

   rand bit   [31:0] addr;
   rand logic [31:0] data;

   `uvm_sequence_utils(user_test_seq , reg_block_slave_sequencer);

   virtual task body();
      reg_block_slave model;
      $cast(model, this.model);

      // Randomize the content of 10 random indexed registers
```

```
      repeat (10) begin
         bit [7:0]       idx = $urandom;
         uvm_reg_data_t  data = $urandom;
         uvm_status_e status;

         model.TABLES[idx].write(status, data, .parent(this));
      end

      // Find which indexed registers are non-zero
      foreach (model.TABLES[i]) begin
         uvm_reg_data_t  data;
         uvm_status_e    status;

         model.TABLES[i].read(status, data);
         if (data != 0)
            $write("TABLES[%0d] is 0x%h...\n", i, data);
      end
   endtask : body
endclass : user_test_seq
```

After adding the file to the *testlib.sv* file, the user-defined sequence can now be executed using the command-line test interface.

Command:

```
% <sim> +UVM_TESTNAME=cmdline_test \
        +UVM_REG_SEQ=user_test_seq
```

Step 10: Learn More

This primer does not demonstrate all of the capabilities of the UVM register library. Please consult Chatper 9 of the UVM User's Guide and the "Register Layer" section of the UVM Reference Manual.

They can respectively be found at:

`$UVM_HOME/uvm_ug.pdf`

`$UVM_HOME/uvm_ref.pdf or $UVM_HOME/docs/html`