Registering the standard: Migrating to the UVM_REG code base

Sachin Patel,
Broadcom, UK
sachinp@broadcom.com

Amit Sharma,
Synopsys, India
amits@synopsys.com

Adiel Khan,
Synopsys, UK
adiel@synopsys.com

ABSTRACT

In June 2010, the Accellera VIP TSC released the "UVM Early Adopter" version of the Universal Verification Methodology (UVM). Subsequently, UVM1.0 was released in the beginning of the year to coincide with DVCON 2011. The UVM1.0 release added a lot of anticipated and relevant functionalities, one of the most significant additions were the standardization of DUT register layer verification. This was the first time the three major verification vendors (Cadence, Mentor, Synopsys) along with multiple semiconductor organizations had aligned on a single IEEE1800-SystemVerilog base class library implementation with associated methodology. For the methodology to enable horizontal and vertical reuse it was imperative that the industry finally resolve on having one standard implementation and mechanism for verifying DUT registers and/or memories. Therefore, it was crucial that the standardization effort was extended to the abstraction level for DUT registers verification.

In this paper, taking the case study of a block level design under verification component MMU (Memory Management Unit), one will walk through the different steps in the migration of a legacy OVM RGM Register package to the standard UVM1.0 register model (UVM_REG). These steps will encompass the generation of the abstracted register model from the input specifications, along with connecting the UVM_REG to the verification environment encompassing the mapping and translation of different APIs to those available within the UVM_REG code base. The analysis and presentation of required techniques would be done so as to abstract away the divergences in the two different register-codebases so that similar techniques can be leveraged in the migration of verification environments using other legacy register packages and to ensure that engineers have the most efficient means of migrating their complete environment to the UVM standard and maximize the benefits of doing so.

Categories and Subject Descriptors

General Terms

Documentation, Performance, Design, Standardization, Languages, Theory, Legal Aspects, Verification.

Keywords

Testbench, Register Package, UVM, SystemVerilog, RAL

1. INTRODUCTION

Every design has host-accessible registers. These registers must be documented, implemented and verified. Since they have well-defined functionality, the process of documentation, implementation and verification can proceed from a common source. A standard register package on top of the infrastructure provided by a base class methodology can enable user testbenches to leverage the functionality already provided thus making them easier to implement correctly and more efficiently. With UVM-1.0, standardization effort was extended to the abstraction layer for DUT registers verification.

It has been noted that UVM quickly gained acceptance as organizations moved from OVM towards a UVM based flow. The move to UVM become more rewarding for OVM users, as they could migrate from their vendor or internal specific legacy register packages to the standard adopted by the UVM verification community. Thereby, they reaped the benefits of having a standard view of register abstraction layer verification. As users migrate from non-standard packages, it is important to understand the challenges that arise during such a migration process. By documenting the efforts undergone one by successful migrations, one can provide a proven path for future users wishing to migrate towards the UVM standard register model.

2. REGISTER PACKAGES

When a design contains a few dozens of registers, creating—and more importantly maintaining—the testcases and firmware emulation routines can be done manually. Using symbolic values for addresses and bit offsets can take care of a lot of maintenance effort. But when the number of registers is in the hundreds, even in the thousands, creating—yet alone maintaining—the register correctness testcases and the firmware emulation routines used by the other functional tests becomes overwhelming. The task gets even more daunting when it becomes necessary to migrate between block-level environment, firmware emulation or tests to a system-level environment: the physical interface used to access the register may no longer be available. The Register Abstraction Layer isolates the upper layers of a verification environment and the testcases running on top of it from the implementation details of accessing the registers and memories in the design. The abstraction model would have appropriate hooks so that it can easily be integrated in an existing verification environment. As seen below, most packages would use an adapter layer to convert generic register transactions to that understood by the User BFM (figure1). The adapter would also be responsible for translating the transactions collected from the Bus back to a generic register transaction that would be fed back to the abstraction layer.

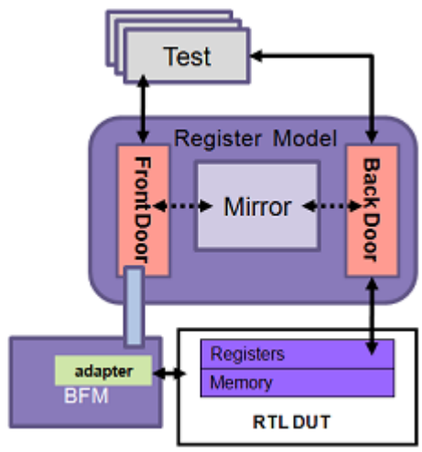

Figure 1: A Register Abstraction Layer for Register Verification

Typically the register packages come with an executable spec. A generator would generate the Abstraction Model from the executable specification. Hence, the authoring of the RAL model by hand is not required and any change in the specification is translated automatically to the model by the generator.

Not only does the RAL simplify the maintenance and verification of accessing registers and memories inside the design, it also simplifies writing the code that need to access them. And this simpler code is unaffected by any changes in physical interface, address, location or bit offset of a field as all of these non-functional implementation details are taken care of by the automatic generation of the design-specific RAL model based on any modification to the executable specification. This benefit is demonstrated in Example 1 and Example 2.

Example 1: Directly Accessing Fields in a Register

env.ahb.read(`MODE_CONFIG_REG_ADDR, val);
val[`PROTECTED_MODE] = 1'b0;
env.ahb.write(`MODE_CONFIG_REG_ADDR, vall);

Example 2: Accessing Fields in a Register through RAL

env.ral.protected_mode.write(0);

Thus, across verification teams, it has been accepted that adopting a Register Abstraction Layer can significantly improve productivity. The benefits extend from the automatic generating of the model from a specification to the abstracted mechanism for accessing and verifying the correct operation of the registers, as well as using other package specific capabilities to converge on the register verification tasks.

2.1 CREATION OF THE STANDARD

The inclusion of a register package was accepted as one of the key requirements for UVM-1.0 The Accellera Committee performed its due-diligence by studying the existing OVM and VMM register packages. The outcome of the study was a single solution for register-layer verification labeled as the UVM_REG. There were specific features and capabilities which were deemed to be integral by the committee and the verification community. It was also desired that the base package was significantly mature and leveraged successfully across multiple large projects. The base VMM Register Package suitably enhanced for UVM and with contributions from Mentor Graphics met all the requirements and was accepted as the UVM_REG package.

2.2 UVM REGISTER LIBRARY OVERVIEW

The UVM Register Layer is set of base classes that can be used to create a high-level, object oriented abstraction model of the registers and memories inside a design. The library of base classes can be leveraged to abstract the read/write operations to the registers and memories in a DUV. It also includes a sequence library which can be used to verify the correct implementation of the registers and memories in the design. It may also be used to implement a functional coverage model to ensure that every bit of every register has been exercised, as well as to verify the different combinations of values being driven to the DUT registers. One key aspect of the abstraction mechanism is to enable a smooth migration of verification environments and tests from block to system levels without requiring any source code modification.

The hierarchy blocks composing of the abstraction model correlates to the design hierarchy. The smallest unit that can be used to represent a design is the *block*. A block generally corresponds to a design component with its own host processor interface(s), address decoding, and memory-mapped registers and memories. Blocks can comprise of registers, memories and register files, as well as other blocks as shown below. Multiple fields comprise a register. The snippet in the left in figure 2a shows how UVM REG code will correlate to the actual design register hierarchy on the right. The access to the design register elements hence can use a very similar hierarchical path.

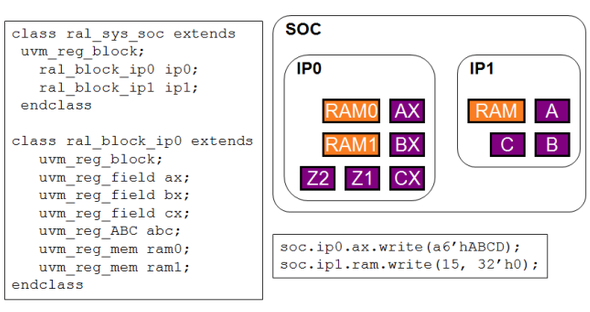

Figure 2a: UVM Reg Code correlated to Design Hierarchy

Figure 2b shows a pictorial representation of the abstraction model corresponding to the actual DUT registers in Figure 2a.

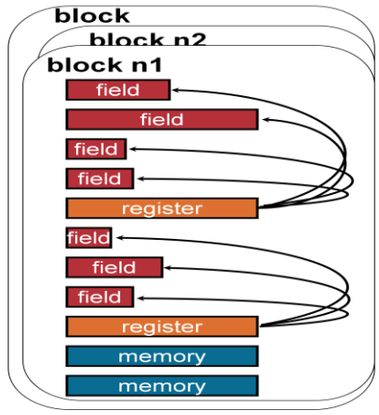

Figure: 2b : Pictorial representation of the abstraction model

The UVM Register model is typically generated from a input specification. Model generators work from a specification of the registers and memories in a design and thus are able to provide an up-to-date, correct-by-construction register model. For verifying thousands of registers as well as maintaining them, model generators are a must-have in any verification setup. Without them, the verification engineer will have to update the Register Model by hand to reflect any change in the input specifications. This can translate to a lot of redundant activity especially during the initial phases of a design ne verification environment bring-up. The model generators themselves as well as a standard input specification are outside the scope of the UVM library.

Once the model is generated, it must be integrated with the bus agents that perform and monitor the actual read and write operations. This is done though an adapter mechanism were the generic read write sequences resulting from RAL model accesses are converted to sequences on the host sequencer. The transformations are in the opposite direction for transactions on the bus which needs to go back to the register model. Once, integrated

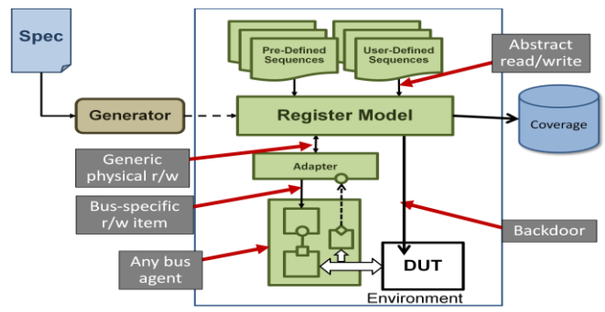

Figure 3: UVM REG Model integrated in a UVM environment

The UVM Register Library provides a set of useful pre-defined sequences that helps to automate a lot of desired functionality to verify the proper operation of the registers and memories in the DUV. Once integrated, the user can execute these set of pre-defined sequences and create their own user defined sequences that can create accesses at a high level of abstraction as shown below:

soc.ip0.ax.write(a6'hABCD);
soc.ip1.ram.write(15, 32'h0);

The Register Library provides the necessary API to control the instantiation and sampling of various coverage models though it doesn't include any as a part of the specification. The model generator would generally generate different kind of coverage models based on the input specifications.

3. Adoption of UVM REG package

Here is a quick overview of the steps required to generate a RAL model of the registers and memories in a design, integrate this model in a UVM verification environment and then to verify the implementation of those registers and memories.

3.1 Input specification and Model Generation

The first step is to create the abstract model of the registers in the DUT. Once the registers and memories have been specified in a one commonly use specification called the Register Abstraction Layer Format (RALF) or IPXACT, the UVM Register Model generator, *ralgen* [10] is used to generate the corresponding RAL model. There are tools from multiple vendors which can help generate the UVM Register Model from multiple other input specifications. The following command will generate a SystemVerilog UVM REG model of the *specified block* in the file *ral_blkname.sv*:

Command:

% ralgen –l uvm -u –t blkname blkname.ralf

The generated code is not designed to be read or subsequently manually modified. However, the structure of the generated RAL model demonstrates how it mirrors the register hierarchy in the DUV. The user can create the Register Model by hand but that is strictly not recommended.

3.2 Creating the Adapter Layer

The role of the integrator is to provide the adapter layer. This is one of the key steps that require user involvement when one seeks to leverage the UVM Register Library in a verification Environment. 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.

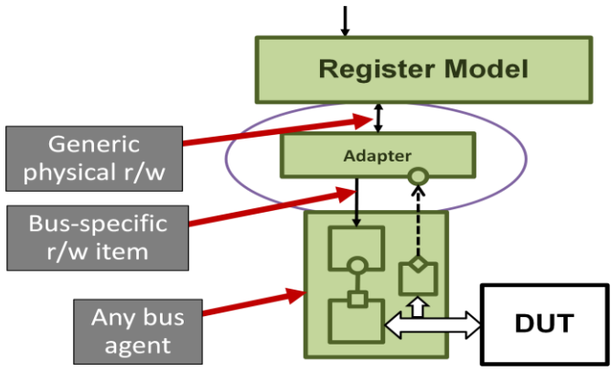

Figure 4: UVM REG Adapter for hooking up the REG Model

The *'uvm_reg_adapter'* class is provided for converting between *uvm_reg_bus_op* which is a generic Register sequence to one which can execute on the host sequencer. Extensions of this class must implement the reg2bus() and bus2reg() methods to convert a *uvm_reg_item* to the *uvm_sequence_item* subtype that defines the bus transaction and vice versa . The functionality provided in the adapter class is specific for different physical interfaces and hence has to be coded appropriately. Each abstracted register access function calls one of these methods implicitly when using the frontdoor mechanism.

3.3 Integration the Register Model

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.

```
ral_block_IP model;
...
model = ral_block_IP::type_id::create("model",this);
model.build();
 model.lock_model();
```

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

```
reg2host_adapter reg2hsot = new;
model.default_map.set_sequencer(host.sqr, reg2hst);
model.default_map.set_auto_predict(1);
```

This is where the 'Adapter' class created in the previous section is instantiated. The '*set_auto_predict()*' method is called to enable automatic update of the Register Model immediately after any bus read or write operation via this map. It is the simplest prediction mode albeit not as reusable. The explicit prediction mode requires the additional integration of a monitor. Using a monitor updates the register model based on all observed read and write operations, not just those performed through the register model.

3.4 Executing pre-defined functionality and creating user-defined sequences

The Functionality delivered by pre-defined sequences include: reading all the register in a block and check their value is the specified reset value, bit bashing all registers and checking results based on the field access policy specified for the field containing the target bit and other bashing all registers, doing a memory walk and other such generic functionality. The user can create a test that runs a pre-defined sequence specified on the command line as specifed here :

```
Command : simv+UVM_TESTNAME=cmdline_test \
      +UVM_REG_SEQ=uvm_reg_hw_reset_test
```

Some of the predefined test sequences require back-door access be available for registers or memories.

4. MIGRATING TO UVM REG FROM CADENCE RGM PACKAGE: A CASE STUDY

Here, we will taking the case study of a block level design under verification component MMU (Memory Management Unit), and walk through the different steps in the migration of a legacy OVM RGM Register package to the standard UVM register model.

4.1 THE DESIGN UNDER TEST FEATURES

The MMU DUT is responsible for the implementation of the virtual memory and translating virtual address to physical address before actually initiating read/write access of the physical memory. It has both a AXI Master and Slave Interfaces. The memory is connected via the MMU AXI master interface. The Memory contains the various Page table entries which the MMU DUT reads for TLB (Translation Lookaside Buffer) misses. The MMU Registers are programmed through an APB interfaces. The important registers are:

- MMU_control : Turns MMU on/off
- PAGE_TABLE: This is the base address register which is a pointer to base address of page table
- TLB_clear : Clear entries of TLB buffer
- TLB Hits/ TLB misses : Counter for number of hits and misses for TLB entries,

There are other registers for Debug and Stalls

4.2 OVERVIEW OF THE RGM VERIFICATION ENVIRONMENT

The Original MMU Verification Environment was based in OVM and the Cadence supplied Register and Memory Model (RGM) was the primary vehicle for register abstraction and verification. Figure 5 depicts the UVM verification Environment leveraging the UVM Register Package.

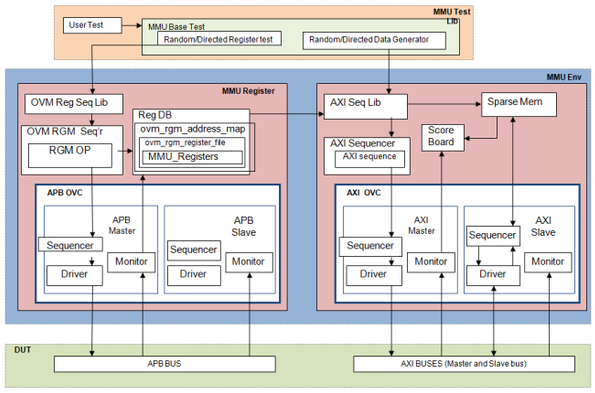

Figure 5: OVM MMU Verification Environment

The programming interface is through the APB driver in the APB OVC. User tests invoke would register write sequences using the OVM RGM abstraction model which will program MMU registers. AXI sequences will pre-program MMU page Table entries in sparse memory. The sequences also read in MMU registers to generate valid/invalid contains of MMU page table in memory. There is a scoreboard for comparing contents of memory against actual memory read write observed on AXI master bus.

The RGM package uses the OVM sequence mechanism to randomize and drive register and memory sequences. In a sequence, you can randomly select a register object from the RGM_DB, randomize it, set the access direction (read or write), and perform the operation. These user sequences are executed on the *uvm_rgm_sequencer*

4.2.1 THE BUS INTERFACE UVC

The RGM sequencer is layered on top of an existing bus master sequencer which in this case was the APB sequencer. As in other register packages, every read and write operation is translated to protocol-specific bus transactions. These register operations are isolated from the protocol-specific bus transactions. The RGM sequencer which is connected to the bus monitor runs the adapter sequence. The adapter sequence converts the register operations to protocol specific bus transactions through the *execute_op* task [10]

```
virtual task execute_op(uvm_rgm_reg_op op);
   read_byte_seq rbs;
   write_byte_seq wbs;
    cur_op = op;
    uvm_report_info("ADTRSEQ", $psprintf("Executing the following operation :\n%0s",
      op.sprint()), UVM_LOW);
    case (op.get_direction())
      uvm_rgm_pkg::OP_RD : begin
        `uvm_do_with(rbs, { start_addr == op.get_address(); } )
      end
      uvm_rgm_pkg::OP_WR : begin
        `uvm_do_with(wbs, { start_addr == op.get_address();
          write_data == op.get_reg_value(); } )
      end
    endcase
  endtask
```

Figure 6: Translation of register operations to bus specific ones

Similarly the adapter sequence has another method called the *process_responses()* which continuously monitors the response from the bus monitor, and passes it back to the RGM sequencer. This is the *process* for further comparison/operation.

```
virtual task execute_op(uvm_rgm_reg_op op);
virtual task process_responses();
   while (1) begin
     get_response(rsp);
      cur_op.set_reg_value(rsp.data[0]);
     p_sequencer.reg_rsp_port.write(cur_op); // Write the monitored data via the reg_sequencer's
port
   end
 endtask
```

Figure 7: Bus transactions to RGM response transaltion

4.2.2 THE INTERFACE UVC MONITOR AND THE MODULE UVC

The APB monitor is used to detect a transaction on the APB Bus. At that point, the transaction information is sent to the module UVC/OVC. The Module UVC takes the transactions collected by the interface UVC monitor and decides what action to execute on the register and memory model. In the case of the MMU environment, the TLM analysis port from the APB monitor accepts the APB transfers and pushes the information into the register DB. In the case of a write access to a register or memory location, the shadow register is updated. When a read access is detected on the bus, the 'implementation' accesses the RGM model and compares that with the read result from the DUT.

4.2.4 THE ENVIRONMENT

The following components are instantiated in the MMU environment in addition to the AXI and APB OVCs: The RGM DB, the Register sequencer and the TLM port for updating the register database with APB transfers. The Bus Monitor is connected to the REGMEM:TLM. The RGM_sequencer is connected to RGM_DB. The Response port of the APB sequencer is connected to the Response Export of the RGM sequencer

```
mmu_register_sequencer reg_sequencer;; //RGM sequence
mmu_ rdb;  // RGM Register Model
// REGMEM: TLM port for updating the register database with APB transfers
 `uvm_analysis_imp_decl(_rgm)
 uvm_analysis_imp_rgm #(apb_transfer, top_env) apb_transfer_imp;
...
//in build_phase//
rdb = mmu_rdb::type_id::create("rdb", this);
 reg_sequencer = mmu_register_sequencer::type_id::create(" reg_sequencer", this);
uvm_config_db#(uvm_object_wrapper)::set(this, "reg_sequencer.run_phase", "default_sequence",
null_rgm_sequence::type_id::get());

//in connect phase : making the connections
apb_rgm_master_sequencer p_apb_seq;
$cast(p_apb_seq,apb.master.sequencer);
reg_sequencer.set_container(rdb.mmu_map);
reg_sequencer.set_top_addr_map(rdb.mmu_map);
reg_sequencer.req_port.connect(p_apb_seq.reg_req_export);
p_apb_seq.reg_rsp_port.connect(reg_sequencer.rsp_export);
apb.bus_monitor.item_collected_port.connect(apb_transfer_imp);

// Implementation body for the TLM analysis port from the APB monitor
  // Accepts APB transfers and pushes the information into the register DB.
 function void write_rgm(apb_transfer trans);
 ...
 endfunction
```

Figure 8: RGM instantiations and connections

4.3 VERIFICATION ENVIRONMENT MIGRATION

The Verification environment was originally in OVM. It was converted to UVM-EA using the OVM_UVM_Rename.pl script which is a part of the Accellera UVM distribution as well the Synopsys UVM distribution. Simultaneously, Synopsys provided UVM-EA to UVM-1.0 script was used for converting 95% of the environment to UVM-1.0. Minimal hand modifications were required to complete the full migration to UVM-1.0. The next step was to migrate from the proprietary RGM flow to the UVM Reg standard.

4.3.1 THE INPUT SPECIFICATIONS

IPXACT based register representation was the general format used for RGM Model generation. If the IPXACT file is available, 'ralgen' from Synopsys can generate the UVM Reg model for the same. However, in this case, the IPXACT file was not available. There were specific challenges in adding constraints to the generated register model, as well as creating relevant functional coverage models for control register fields when using IPXACT as an input format. Hence, the hand modified register model was the

starting point for the migration. The RGM model also fairly maps the hierarchy of the DUT registers and hence for the number of registers in the MMU DUT, it was a trivial task to create the RALF file from the same. Once the RALF file is generated, the generation of the register model can be done with a flip of a switch. For larger environments, the expectation is that the input specification will be available. If it is IPXACT, there are no migration requirements of the same. Otherwise, some automation has to be created for the transformation if it not available already. Third Party tools now convert can convert from nearly all inout specification formats to either RALF or the UVM Reg Model.

4.3.2 MIGRATING THE ADAPTERS

Creating the Adapters in order to integrate Register Package is typically the role of the integrator. In the migration effort as well, it is crucial that this is done correctly. The following snippet shows how to adapter code will be in UVM Reg for the corresponding code in *execute_op* earlier

```
class reg2apb_adapter extends uvm_reg_adapter;
  ..
  virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
    apb_transfer apb = apb_transfer::type_id::create("apb_transfer");
    apb.direction = (rw.kind == UVM_READ) ? APB_READ : APB_WRITE;
    apb.addr = rw.addr;
    apb.data = rw.data;
    return apb;
  endfunction

  virtual function void bus2reg(uvm_sequence_item bus_item,
                                ref uvm_reg_bus_op rw);
    apb_transfer apb;
    if (!$cast(apb,bus_item)) begin
      `uvm_fatal("NOT_APB_TYPE","Provided bus_item is not of the correct type")
      return;
    end
    rw.kind = apb.direction == APB_READ ? UVM_READ : UVM_WRITE;
    rw.addr = apb.addr;
    rw.data = apb.data;
    rw.status = UVM_IS_OK;
  endfunction
endclass
```

Figure 8: Register to APB translation in UVM REG

4.3.3 MIGRATING THE MODEL INTEGRATIONS

Once, the adapter class is created, it is a matter of changing the instantiations and connections related to the Register Model.

```
ral_sys_mmu model; //Generated UVM REG Model
reg2apb_adapter reg2apb;
//uvm_reg_predictor#(apb_transfer) predict; //required if explcit mode is used, not used here

//build phase
model = ral_sys_mmu::type_id::create("model", this);
 model.build();
 model.lock_model();
//apb2reg_predictor = new("apb2reg_predictor", this);

//connect phase : Creating the adapetr and connecting the sequencer to the default map
reg2apb = reg2apb_adapter::type_id::create("reg2apb");
 model.default_map.set_sequencer(apb.master.sequencer,reg2apb);
  model.default_map.set_auto_predict(1); // implicit prediction
/* not used in this migratuon , but can be used if an explict prediction is required
apb2reg_predictor.map = regmodel.APB;
apb2reg_predictor.adapter = reg2apb;
regmodel.APB.set_auto_predict(0);
apb.monitor.ap.connect(apb2reg_predictor.bus_in); */

model.set_coverage(UVM_CVR_ALL);
```

Figure 9: UVM Reg model instantiations and connections

Thus, migrating the Register Model integration and connections is quite trivial. There are fewer connections and instantiations to be made. The code snippet above also has the instantiation and integration of the Predictor Model commented out as we felt the implicit mode was more relevant to our tests. Given that the fact that the bus monitor was available, the Predictor Model integration is relatively straightforward as seen above. The predictor model when used would accept bus transactions from the APB monitor and use the preconfigured adapter to obtain the canonical address and data from the bus operation. The map is used to lookup the register object associated with that address. The predict*()* method would be then used to update the mirror. The explicit mode maps directly to the update model of the RGM package.

4.3.4 MIGRATING THE REGISTER ACCESSES

Most of the APIs which are a part of the RGM package have their counterparts in the UVM REG model. The important aspect to keep in mind is the intent behind the use of different methods. Then it becomes easier to map the functionality used in the proprietary register packages to the ones available in UVM REG. Most of the Register Model access in RGM is through a set of macros which delivers the functionality. As the RGM model is not purely hierarchical, it is not possible to make hierarchical access especially in the context of fields and the granularity is only up to the register level. For example, an entire register is randomized and written to when a `rgm_write(REG_ARG) is called. Also, the REG_ARG in this case is not the hierarchical name. The corresponding operation in UVM REG would be to randomize the register separately calling reg.randomize() followed by a reg.update(); Though these are 2 methods being called, the UVM REG model will only write to fields which have changed and thus not waste redundant BFM cycles. For more details on other access functionality, one can go through the respective Reference Guides [2], [11] In fact if desired for large projects, automation can be brought in to map the functionality in the access APIs to the ones in the UVM REG Library

The RGM library has 6 user defined sequences and the UVM Reg Model has 14. Though a lot of functionality is similar, there are a couple of sequences like the *uvm_rgm_aliasing_seq* for which the UVM REG user can easily model similar functionality.

5. RESULTS AND CONCLUSION

The migration of the MMU block verification from using OVM_RGM to UVM_REG was successful. By migrating to UVM_REG the bulk of the verification was achieved by using the built-in test sequences then observing functional coverage. The test sequences used included hw_reset, bit-bashing and HDL path checking. Not all features of the UVM_REG were exploited as the MMU did not have a need for shared access registers, complex memories or virtually overlaid register spaces.

Other engineers within the team have further leveraged UVM_REG by using *ralgen* [10] to automatically generate the functional coverage of the DUV registers. This coverage model of the registers is then be reused to create hierarchal cross coverage to observe particular DUV functionality. Additionally there was a lacuna in generating coverage bins for interesting values of the Control Registers. With the granularity extended to individual fields registers, we could now created user defined coverage bins in the field specification. This was something which was not possible with the legacy specification. With the coverage models in place, it was possible to quickly converge on the interesting values through a mix of pre-defined sequences and user defined one. The other gap that was observed in the legacy package was in adding constraints to the generated register model due to specific limitations in the automation flow. This is also addressed now as using UVM REG and RALF allows us to add in the constraints in the specification as also inline additional ones in the model. There were other challenges included the validation of the power up values for various registers, the bootup sequence and bit bashing reserved fields of all registers. We are now in the process of addressing them with UVM REG.

To help all engineers within Broadcom there has been a flow developed combining the in-house register-automation with UVM_REG. All SW and HW register information is stored in a Broadcom specific format maintaining uniformity throughout the company. The Broadcom RDB format can is translated to RALF syntax using rdb2ralf script. By having such a flow we can maintain a single source entry for all register data and leverage the information for documentation, SW development, RTL coding and Verification purposes. The reverse flow of updating RDB files from RALF modifications could be developed by using RALF C++ API but as of yet that need has not arisen.

There is a current task of unifying the C and RTL verification by using UVM_REG with a C-API. This would allow SW engineers to reuse UVM verification environments whilst providing them with a simple C read and write API to access the DUV registers.

The move from OVM_RGM to the industry standard UVM_REG has been very fruitful for our projects. It has not only provided all the requirements for register verification but also provided enhanced automation. UVM_REG will continue to unify teams as we provide enhanced functionality to leverage the C API's and we look forward to EDA companies and the vibrant UVM ecosystem to provide us even more automation along with tools to promote UVM_REG usage.

6. REFERENCES

[1] UVM User Guide
[2] UVM Reference Manual
[3] UVM World http://www.uvmworld.org/
[4] VMM Martial Arts : www.vmmcentral.org
[5] Accellera Verification IP Technical Subcommittee (UVM Development Website); http://www.accellerea.org/apps/org/workgroup/vip
[6] UVM Register Abstraction Layer Generator User Guide
[7] UVM RAL Primer, Janick Bergeron
[8] VMM application packages: the next level of productivity, Janick Bergeron
[9] RGM User Guide
[10] UVM Register Abstraction Layer Generator User Guide
[11] RGM Reference Guide