Register This! Experiences Applying UVM Registers

By Sharon Rosenberg - Cadence Design Systems

Abstract

Controlling and monitoring registers and memories comprises a large part of typical functional verification projects. While this task can be managed manually for low-complexity subsystems, most projects have thousands or even hundreds of thousands of registers. Such environments demand automation and reuse and the Accellera UVM_REG register and memory package can provide it. The UVM_REG combines elements from multiple proprietary solutions (e.g. Synopsys RAL, and Cadence UVM_RGM) with new code from Mentor for tight alignment with the UVM BCL and methodology.

First released in March 2011, the UVM_REG has been used in production, which has led to several enhancements and enhanced use models. Since the use model a team chooses often depends on the circumstances, perspective, and history of the team, it is important to start with a common understanding of the library capabilities and some traditional and new use models. From that foundation, this paper compares and contrasts techniques and recommends a proven methodology with practical guidelines gleaned from real project experience. Among the topics and techniques that will be detailed are the following:

- Passive monitoring vs. checking in sequences
- Safe parallel register operations
- Vertical reuse and registers
- Who's afraid of IP-XACT? (Introduction to IP-XACT)
- What to expect from a register generator
- Debugging registers and memory
- Migrating to the UVM_REG from an existing solution

This paper provides useful information for users with different levels of experience and allows them to either validate their current register methodology or ask the right questions when moving to the UVM_REG. This paper includes code examples that are posted to the UVMWorld contribution enabling readers to implement the recommendations easily.

Keywords

UVM, Accellera Systems Initiative, SystemVerilog, VMM, RAL, IP-XACT, reuse, verification

1.0 The uvm_reg Usage

If the verification environment does not yet include register and memory package support, it is important to understand that when a register and memory package is introduced most of the interaction with the device's memory mapped registers is done via a protocol agnostic register API.

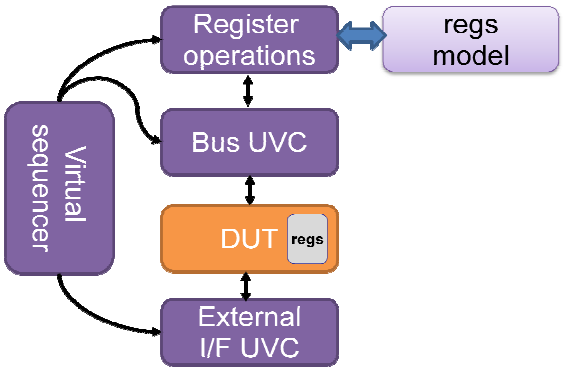

Figure 1: Interaction of Registers and the DUT

As illustrated in figure 1, bus operations can still be executed, but much of the interaction with the DUT is done at the register level saving the test writer the need to learn the protocol and implementation specific details. An example of this is when write operations are performed as opposed to an AHB write transfer to a certain address. The register write operation is later translated into specific bus operation(s) but the register operation-to-bus translation logic is done once per bus UVC and typically stored in the bus UVC package.

The configuration code abstracts away protocol specifics for test writers, and becomes protocol independent. The register operation logic can be layered on top of a certain protocol and then it can be swapped to a different protocol in the future while keeping the configuration sequences intact.

Registers and register files are an excellent vertical reuse (module-to-system) opportunity, as sub-systems configuration logic is valid and reusable at the system integration level. Designs can be packaged with their configuration sequences allowing the system integrator smooth operations without the need to learn all the sub-system configuration details.

The shadow register model is a hierarchal reference model for a specific DUT and captures the DUT memories and registers structure and attributes. It contains nested objects of register-blocks, registers and their field classes that are derived from the uvm_reg classes and specialized to the specifications at hand. The model allows the randomization of legal configuration values, checking of the DUT register values for correctness and collection of coverage. The register model is often automatically generated from a register specification using a code generator.

Coding the SystemVerilog (SV) model class manually is a labor intensive task that requires deep knowledge of the UVM_REG base classes, and is hard to maintain if the specification changes.

While this paper can't replace a full training class, there are a few access methods for memories, registers, and register sub-fields that will make the examples easier to understand:

- **write()/read()**: Write/read immediate value to DUT.
- **set()/get()** : Sets or gets desired value for the register model.
- **randomize()** **:** Using the randomize() method copies the randomized value in the uvm_reg_field::value property into the desired value of the mirror by the post_randomize() method.
- **update()** : Invokes the write() method if the desired value (previously modified using set() or randomize()) is different from the mirrored value.
- **mirror()** : Invokes the read() method to update the mirrored value

based on the read back value. mirror() can also compare the read back value with the current mirrored value before updating it.

Note that you can mirror() and randomize() full compound elements such as register files in a single operation.

The signature of the methods includes multiple parameters and typically uses a combination of binding by name, by position, and with default values to simplify the method calls.

An example of the signature of write is:

```
virtual task write(output
uvm_status_e status, input
uvm_reg_data_t value,
     input uvm_path_e
path=UVM_DEFAULT_PATH, input
uvm_reg_map map=null,
     input uvm_sequence_base parent
= null, input int prior = -1,
     input uvm_object extension =
null, input string fname = "",
     input int lineno = 0 )
```

and a use model can look like this:

```
  model.config_reg0.write(status,
'h34, UVM_BACKDOOR, .parent(this));
```

The first three arguments are bound by position and the parent field is bound by name. This provides some background on the hierarchical structure of the register model and it's API. These examples will be used again in this paper.

2.0 Considerations for Selecting a Code Generator

As described above, the first step to leverage the **uvm_reg** base classes involves transferring a register specification to a specialized uvm_reg register model. Unfortunately, the UVM reference library does not include a code generator which is a critical element in making the register logic scalable, reusable and vendor independent. Here are a few considerations for selecting an existing or creating the code generator you need.

1. Scalability - designs may include large numbers of registers and scalability is an important consideration. The customized code in uvm_reg (and this is derived from the uvm_reg implementation) is typically large and might create a bottleneck at compile time or even crash the compilation due to memory explosion. Run-time is typically secondary but should be observed as well.
2. Vendor independent generated code – Large part of the SV language and the UVM library promise involves the ability to run-code on all simulators. Although this wish is not feasible yet at the language level, it is still a strong requirement to ensure that the generated code is supported by all vendors and that the semantic of the used features is consistent across vendors.
3. Support for a standard input format – History shows that eventually standards overcome local initiatives and with a few exceptions, the industry progresses and moves forward in the right direction. Using proprietary input formats may disconnect your team from the overall progress and new upcoming solutions that are built on top of standards.
4. Ability to replace the generator as needed – In addition to the input format that we discussed above, make

sure that the generator does not create an extra API that is not part of the standard uvm_reg. Having the configuration logic and sequences use such added API will limit the ability to swap a commercial or homegrown generator with a different one.

5. Debug-ability and self-checking of the generated code – Typically, the generated code is not intended to be read by users. However, it is important that the generated code include construction time checkers to identify issues in the input specifications, as well as debug capabilities in case an issue is discovered during compile or run-time.
6. Productized and productive solution

Given that a generator is necessary, the choices are to create a proprietary generator or adopting a commercial one. The benefit of creating a proprietary generator is the flexibility to enhance and tune it to local needs. Process-oriented companies have proprietary specification formats and solutions that integrate designs, compose configuration logic, create documentation, and more. Creating a single automated process that results in multiple design and verification artifacts that are correct and consistent by construction is an important goal to pursue to streamline and shorten design integration process. This means that the legacy input format must be maintained as the UVM register package is being adopted. Obviously, adopting a commercial generator reduces the development effort, checking compliance with all simulators, maintaining the generator as the uvm_reg classes are enhanced, and provides access to novel capabilities requested from the larger verification community.

A middle-ground solution involves creating an adaptor from a proprietary input format into the standard format that is accepted as input to a commercial generator. This allows balanced the creation or continued use of a proprietary format while leveraging the maturity, low-maintenance and support of a commercial tool.

Which option is the right one for a given project? The solution can vary from one company to the other. Some companies do well with internal development. Others are stuck with inferior solutions that may have been the state of the art a few years ago, but have since become inferior, buggy and an overall burden on the company. If this is the case, the internal support team may want to review commercial options.

Since most project teams will use these criteria to choose a commercial generator, it is worth the time to explain how these features come together in Cadence's commercial generator as an example. Cadence iregGen is a native IP-XACT to UVM generator (no intermediate formats are being created that can impact debug). The front-end IP-XACT format has been used in production for more than three years to support the Cadence uvm_rgm package and more than a year with uvm_reg. The accumulated experience in making the generated code concise, scalable and vendor independent enables optimized code creation which can run on all major simulators. The tool supports optional IP-XACT standard extensions that are being evaluated for the next IP-XACT revision (a runtime argument allows identifying non-current IP-XACT standard usage).

iregGen supports registers as well as memories, wide range of registers such as immediate, fifo, shared and more, and automatically creates functional coverage. A unique capability of

iregGen is the ability to customize the front-end to support user-defined formats to the back-end code generator. This is important for companies that are using a proprietary specification format but do not want to constantly develop and maintain their own uvm_reg code generator.

3.0 The IP-XACT Standard Input Format

At first glance the Accellera standard IP-XACT input format seems verbose and complicated. The standard itself is wider in scope than just registers, and enables other block composition related technology. Experience shows that complexity is a minor issue for most teams once the work begins. Project teams that used IP-XACT can adopt one of the many existing technologies that standards cultivate and allow easier viewing and editing. For example, such editors allow compose-time checking against the IP-XACT schema, which means that an error will be detected as soon as you type it in the editor. We also discovered that many users are comfortable with using their favorite text editor and learn to natively read and write the IP-XACT format.

Here is a short example of a register definition in IP-XACT.

```
<spirit:register>     <!– CONFIG
REGISTER -->

<spirit:name>config_reg</spirit:n
ame>

<spirit:addressOffset>0x0010</spi
rit:addressOffset>

<spirit:size>8</spirit:size>
       <spirit:reset>
<spirit:value>0x00</spirit:value>

<spirit:mask>0xff</spirit:mask>
</spirit:reset>

       <spirit:field>    <!–
FIELD DEFINITIONS -->

<spirit:name>f1</spirit:name>

<spirit:bitOffset>0</spirit:bitOf
fset>

<spirit:bitWidth>1</spirit:bitWid
th>

<spirit:access>RW</spirit:access>
      </spirit:field>
     <spirit:field>
<spirit:name>f2</spirit:name>
<spirit:bitOffset>1</spirit:bitOf
fset>

<spirit:bitWidth>1</spirit:bitWid
th>
<spirit:access>RO</spirit:access>
     </spirit:field>
. . .
</spirit:register>
```

The standard also recognizes that it is impossible to conceive the multiple attributes that users may need and provides an extension scheme to the core standard. Examples of the extensions that users leverage in their register models are the ability to capture field dependencies with constraints or coverage directives.

```
<spirit:vendorExtensions>

    <vendorExtensions
type>ua_cr_c</vendorExtensions:type
>

    <vendorExtensions:constraint>c1
{tx_en!= value.rx_en;}


</vendorExtensions:constraint>

</spirit:vendorExtensions>
```

The Accellera Systems Initiative has a standardization effort to enhance IP-XACT to support more registers related attributes as part of the standard.

4.0 Setting-up uvm_reg Model in the Testbench

As described previously, the register operation logic is layered on top of the bus API. There are two ways to set up this layering. You can instantiate a register sequencer and execute configuration sequences on it or execute a register sequence on top of the bus sequencer. Both techniques require an adapter definition that translates between register operations and the specific protocols data items.

When executing a register sequence on top of a bus UVC (figure 2b) the user can combine both bus operations with register operations in the same sequence body. This simplifies the ability to embed non-register accesses with register operations. The local bus can also be easily grabbed as needed to serve interrupts.

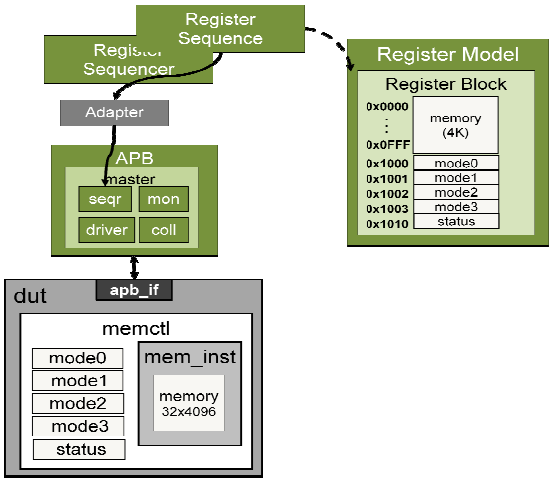

Figure 2a: Layering of uvm_reg Sequencer

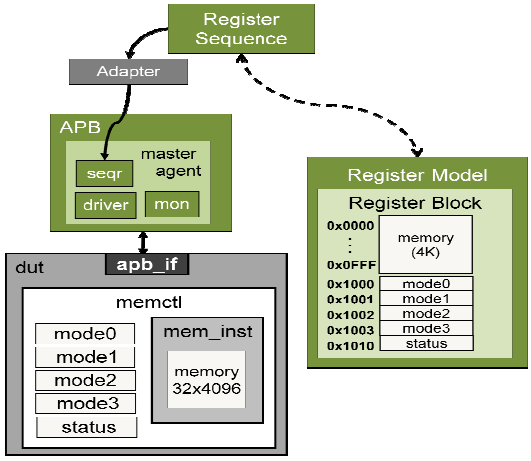

Figure 2b: Executing a Sequence on top of the Bus Sequencer

In the other layering setup (figure 2a), a dedicated sequencer for the register operation is layered on top of the bus sequencer.

5.0 Creating Configuration Sequences

It is possible to leverage uvm_reg without sequences. From user-defined tasks, it is possible to call the uvm_reg read and write functions, update, and compare register values as needed. As far as the author was able to research, this is the only documented methodology for RAL (the VMM register package) users, and thus might be natural for users with VMM background.

If the project is transitioning to UVM from VMM or another proprietary methodology, it is important to repeat the motivation of using sequences in the generic case and in the context of registers. In UVM, sequences are the main format to capture any kind of ordered stimuli. They introduce a lot of value such as late generation and the ability to react to the state of the DUT, they remove the test-writers need to learn and use the UVM factory as it is automatically called by the `do` operator, they define standard implementation for layering, priorities, handling exceptions and interrupts,

multi-channel system-level control and much more. Introduced more than a decade ago, and implemented early on in OVM, it is rare to find a project environment that doesn't leverage sequences for stimuli creation. Project teams are highly encouraged to use register sequences for their value and for easy adoption. Here is an example of a register configuration sequence.

```
class blk_seq extends
uvm_reg_sequence;
     my_rf model;
    virtual task body();
       uvm_status_e status;
       int data;

model.mode0_reg.write(status,
'h12, .parent(this));

model.config_reg.write(status,
'h34, UVM_BACKDOOR,
.parent(this));

model.config_reg.read(status,
data, UVM_BACKDOOR,
.parent(this));
       model.my_mem.write(status,
'h8, 'h1234_5678, .parent(this));
       void'(model.randomize());
   endtask : body

    `uvm_object_utils(blk_seq)
    function new ( string
name="blk_seq" );
       super.new(name);
    endfunction : new
  endclass : blk_seq
```

6.0 Checking for Correctness

Register checking and coverage is useful. Register field values map nicely into DUT operation modes and designers appreciate the ability to observe the combinations of configurations that were exercised. Consistency checking against mirror/reference can identify errors regardless of the testbench implementation or DUT complexity.

The first questions asked include the following: "Where do I place the monitoring logic?" and "Is it in the sequences or passively via passive monitor?" Separation of the injection and monitoring paths is one of the basic concepts of UVM, and many UVM users have strong negative emotions about mixing stimuli with checking and monitoring.

Monitoring and checking mixed with the stimuli is traditionally a directed testing approach in which updates and compares are done as part of the directed test or sequence body. The advantage is that it seems natural for directed test writers that the driving and the checking are done in the same scope thus easier to correlate to each other.

The downsides of this approach are many. A check that is done in a specific sequence is valid only for this specific operation and not in other scenarios or sequence variations that require the same check. Also the inevitable question is "what if the test is not driving the bus?" It could be other VIP or even a DUT block that drives the bus. This scenario is typical for vertical reuse in which an external bus becomes internal in a larger system and is no longer driven from the testbench. Other considerations are the parallel nature of register operations. While the sequence body() seems like a single continuous procedural block, it may run in parallel to other sequences. In such cases, assumptions on register values can grow stale quickly due to other parallel competing sequences and a check might fail. There are other advantages for passive monitoring but this should be a strong enough case to justify the recommendation to split the stimuli from the checking. Just to conclude the discussion, in most cases it is ok to

have a check in the sequence body but the main location for reusable and accurate checking and coverage should not be inside sequences.

7.0 Leveraging the Desired Value for Checking

A register field is an instance of uvm_reg_field with a similar data width (default 64 bits) and holds three copies of the value: mirrored (a reflection of what should be in the actual hardware), value (a value to be randomized), and desired (a desired value for the field for reference and comparison). The original intention was to leverage these fields for shadowing but this also requires a discussion.

The issue with the field usage is to support parallel register activity that can be achieved by multiple masters that can write to a single register or even with pipelined busses. For example, randomizing a register would already change the desired value. If the randomized value was not written to the bus (e.g. generating tasks for DMA to be scheduled to be executed later on, an incomplete bus transaction that may be randomized but never make it to the DUT register, pipelines that introduced a delay, etc.), an undesired comparison will take place. The package supports semaphores on registers that prevent parallel activity such as randomization or accesses. If a register operation is in flight on a certain register, a second operation will be blocked till the first one completes.

It is possible to try to leverage these capabilities to solve the shadowing of multiple accesses challenge. However, changing the stimuli to enable the checking is a dangerous act that can eliminate corner cases. For example, a project may want to check that the DUT can handle two register operations coming from two different interfaces at the same time. There might be hardware or software logic to make sense out of this corner case, and this scenario definitely should not be eliminated because of the shadow model limitations. So how can the safe monitoring of registers be handled? The values must be monitored independently in a passive way.

It is important to note that with respect to coverage; many times a complete configuration of DUT involves setting multiple registers. Make sure that that the coverage is not sampled before all the values are set, and that the final configuration is indeed exercised before marking it down as covered.

8.0 Debugging Registers Related Activities

While the concept of registers is simple and sound, the library data structure implementation is complex. A project team may want to review register hierarchy and properties, check the current values, set break-points on register upon modifications, access or other related events and dump values to the waveform.

The fact that a standard exists allows different vendors create various visualization and debugging facilities. The screen shot in figure 3 is taken from typical register debug window.

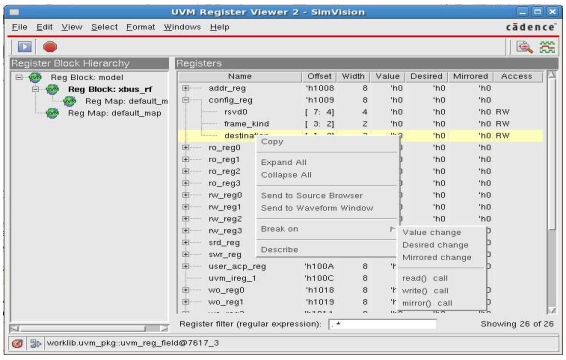

Figure 3: Register View in Debug

The viewer allows traversing the register model hierarchy while expanding and collapsing as needed. Users can review the register properties and set break points as desired.

9.0 Summary

UVM1.0 introduced the uvm_reg base classes that finally allows cross industry convergence on register descriptions and automation. Multiple project teams with different backgrounds and habits are adopting this solution. Any interested parties are highly encouraged to attend a training class that follows these UVM concepts to benefit your internal and intra-company verification projects.