Making RAL Jump, an Introspection

Jeremy Ridgeway
LSI Corporation
Ph: +1 408-433-4257
Email: Jeremy.Ridgeway@lsi.com

Karishma Dhruv
LSI Corporation
Ph: +1 408-433-8292
Email: Karishma.Dhruv@lsi.com

Manmohan Singh
LSI Corporation
Ph: +9 18-04197-7422
Email: Manmohan.Singh@lsi.com

***Abstract*—The register abstraction layer (RAL) in the universal verification methodology (UVM) library provides a valuable code base for easy re-use throughout the environment. In a recent PCI-Express transaction and link layer verification project, the RAL model was connected to the data path via back door access to primary DUT inputs and outputs. With field-level constraints, scoreboarding, and special back door access handling, our RAL model served as a full verification component. We present details on how to make RAL jump as well as some pros and cons to consider.**

I. Introduction

PCI-Express defines a standard set of configuration registers [8]. Our device under test (DUT) in a recent project encompassed PCI-Express transaction and data link layers. The internal configuration registers were accessible on three interfaces (refer to figure 1):

1) External device to physical interface (PHY) with a configuration data packet,
2) CPU bus interface (AHB) to application layer, or
3) Input/Output (IO) port to application layer.

A requirement was to test all register accesses in a universal verification methodology (UVM) register abstraction layer (RAL) model. Immediately, we were faced with a unique set of challenges.

For basic configuration DUT ingress flow, in figure 1, the bus functional model acted as an external device and transmitted a configuration data packet to the DUT. When received, the DUT consumed the packet and updated its internal configuration register. The new value was then presented to the application layer as a primary output, or could be read through a CPU read transaction on the AHB bus interface.

At issue was that the configuration flow exercised the data path but resided outside regular data path verification because the packet was consumed rather than passed. This prevented easy scoreboarding. Furthermore, the DUT had disjointed interfaces that required autonomous verification components to communicate with the RAL model. Instead, we opted to maximize reuse in the RAL model by connecting it directly to multiple interfaces and with global test bench access.

Configuration space testing was made possible with the RAL model as a full verification component. Referring to figure 1, the bus functional model (BFM) acting as external device, randomized a specific register field in the RAL model and transmitted the new value to the DUT. Active monitoring on the application layer primary outputs resulted in an update

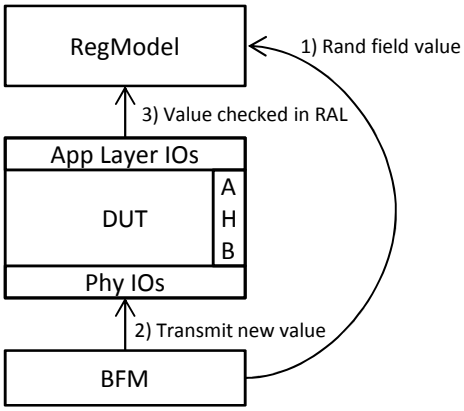

Fig. 1. Basic ingress configuration test flow.

and check on correctness in the RAL model via back door access. With the model acting as a scoreboard, per field, unsatisfied updates were flagged at simulation exit.

In this paper, we first discuss the base RAL model requirements and project-specific base classes implemented in section II. We delve into detail on DUT application layer IO connections in sections III and IV. An important point in register testing is the sheer volume of registers and fields that must be created. We employed automation through scripting as described in section V. Pros and cons to this approach are discussed in the conclusion, section VI.

II. Blocks, Registers and Fields

Our RAL model was connected to two DUT interfaces. The AHB interface was accessed via the RAL front door path. RAL model standard access is through front door transactions. The DUT primary input and output ports to the application layer were accessed via the RAL back door path. Additionally, these ports were actively monitored with automatic register field updates.

To enable the basic configuration test flow, as described in section I, the RAL model needed to take care for the following items:

- Constraints at all levels,
- Controlled randomization,
- Driving reset value,
- Scoreboarding, and
- Integrating back door access.

Refer to sections III and IV for details on the back door access integration.

A. Constraints

Our primary concern with register constraints were twofold: Tests had to randomize register fields separate from registers, and constraints on those fields had to support interdependencies. The UVM user's guide notes that the register field class is not usually required as fields may be constrained within the register [3]. However, constraints in the register scope have no bearing on field-specific randomization. Therefore, we implemented full register field classes.

In figure 2, the field class specifies a constraint (denoted with the bold **c**) for the valid range as well as references to its logical ancestry. The constraints were declared *without* the

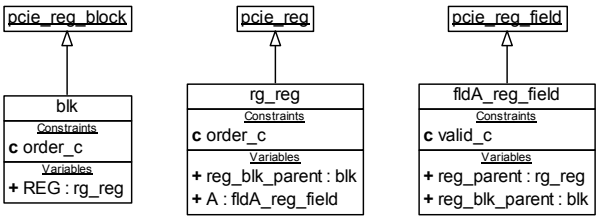

Fig. 2. Class structure for constraints.

extern attribute:

```
constraint valid_c;
```

According to SystemVerilog, this construct is implicitly declared extern. If the constraint is not defined anywhere the compiler treats it as empty and issues a warning only [6].[1] Empty constraints in all RAL related classes enabled constraint definition over time. The constraint in the register field specified the valid range for the field value. In contrast, the constraints in the register and register blocks primarily set solver order on fields (e.g., **solve** x **before** y).

Each implemented class declared reference to its ancestry. These references had to be defined within the implemented class rather than a base class (e.g., pcie_reg) to support proper cross-hierarchical binding at elaboration time.

```
constraint
reg_ecrc_check_enable_field::valid_c {
  (reg_block_parent.reg_A.
     ecrc_check_capable.value == 0) ->
      value == 0;
  (reg_block_parent.reg_A.
     ecrc_check_capable.value == 1) ->
      value inside {0,1};
}
```

In the example above, the value in the ecrc_check_enable field is constrained by a value in the neighboring register's ecrc_check_capable field, as depicted in figure 3. Then, the register block's order constraint indicated solving the ECRC

[1]As of this writing, implicit constraints are supported in VCS but not all compilers [11], [4].

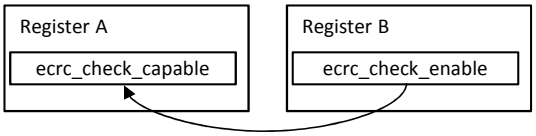

Fig. 3. Enable ECRC field constrained by capable field in neighboring register.

capability field before the ECRC enable field. These interdependences are supported by the compiler only when direct cross-hierarchy access can be resolved (i.e., polymorphism does not apply here).

B. Controlled Randomization

A subset of the PCI-Express configuration space required automatic controlled randomization. That is, some registers set the simulation mode (e.g., operating speed) and thus were randomized a single time only. Other registers were not randomized initially but were required to be randomized later during test sequences. We accommodated both requirements outside of register field access type in the project-specific register block base class, pcie_reg_block.

All fields in the RAL model were declared inside the respective register with the **rand** attribute. Then, each field, during configuration, specified its randomization mode in its block grandparent. The register block, refer to figure 4,

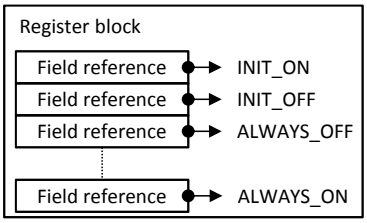

Fig. 4. Register block maintains table of field randomize modes.

maintained an associative array of randomization modes keyed by the register field reference. Overall, two classes of modes were supported: a persistent mode, ALWAYS, and an initial mode, INIT. Figure 5 shows the randomization mode transition after randomizing the register field. Once a field entered the

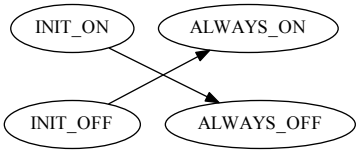

Fig. 5. Randomization mode transitions.

ALWAYS category, its mode persisted.

Implementation required some instrumentation in the register block and register field base classes, as shown in the class diagrams in figure 6. Note, the plus sign (+) indicates public access on a variable, function, or task; the hash mark (#) indicates protected access. The register block maintained

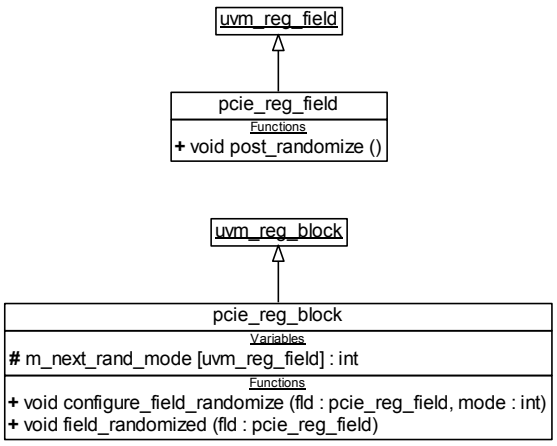

Fig. 6. Randomization mode implementation.

the associative array keyed by register field reference (scoping to uvm_reg_field was sufficient). During configuration, the field was enabled for randomization on ALWAYS_ON and INIT_ON modes, and disabled otherwise. Register configuration was executed once during RAL register block build.

During simulation, the register field post-randomize function triggered a randomization mode update in its register block grandparent via field_randomized(**this**). At that point, the register block updated the field's value rand mode again, if necessary and according to the transitions in figure 5. Test code could reconfigure any field randomization mode and, usually for debug purposes, had access to field.value.rand_mode().

C. Driving Reset

Resetting the register model means that each register field mirror value is set to the register field reset value [3]. In our PCI-Express verification environment, this was insufficient. A subset of register fields in the RAL model were bound to DUT primary inputs. And, reset() followed by update() did not always drive the mirror value to the DUT. For example, if a register field was set as non-volatile, then during update() the field would indicate that it didn't need to be updated in the DUT; no write transaction would occur. Therefore we implemented a drive reset scheme in our project-specific RAL base classes, as shown in figure 7.

The register update() function first queries each register field to determine if an update write to the DUT is actually required. We added a flag in the register field, m_drive_reset, that was asserted during the drive reset sequence. When the register queried the register field at that time, and via the overridden needs_update() function, the register field always returned true. This ensured the reset value was always driven to the DUT primary input.

After determining an update is internally required, the register reads the value as a final check before writing. If the read value is equivalent then a write does not occur. This was incomplete in our environment during the drive reset sequence. The back door read to the DUT IO was incorrect either because:

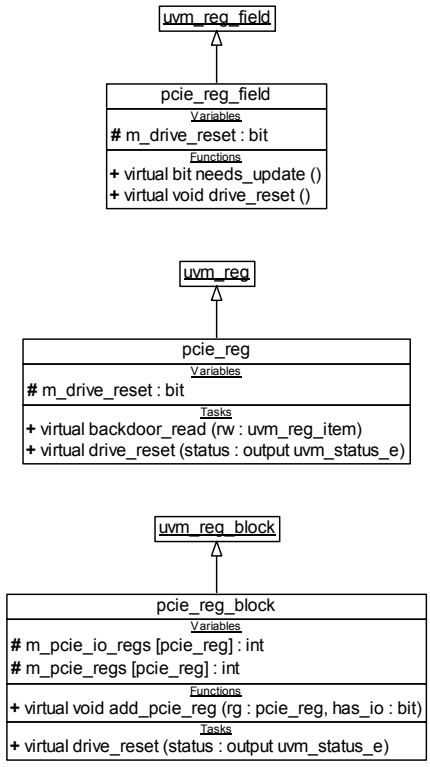

Fig. 7. Drive reset implementation.

- the read value was actually unknown, or
- the read cross-hierarchical path does not match the write cross-hierarchical path.

The back door read path in our RAL Model could differ from the back door write path on a single register field (see figures 8 and 9). Therefore, we also added a flag to the register, m_drive_reset, that was asserted during the drive reset sequence. In the backdoor_read function, when the flag was asserted back door reads were disabled.

The register block maintains two associative arrays of contained registers. A full array was a duplication of the register array in the UVM register block, **local int** regs[uvm_reg]. The local protection in the UVM class requires a copy for iteration in the extended class. The register block also maintains an associative array of contained registers *with* fields that are bound to cross-hierarchical paths (i.e., DUT primary IOs). This array is a subset of all registers in the model. During RAL register block build, each register at construction indicates that it contained at least one field with back door access (add_pcie_reg(.rg=**this**, .has_io=1)). If so, the register reference is added to both associative arrays.

Now, proceeding top-down, the register block drive_reset() task iterates over all registers, as shown in the following pseudo-code:

```
pcie_reg rg;
foreach(rg = m_pcie_reg[i]) begin
  rg.reset();
  if(exists(m_pcie_io_reg[rg]))
      rg.drive_reset(...);
```

```
end
```

The register drive_reset () task follows a similar process:

```
pcie_reg_field fld;
m_drive_reset = 1;
foreach(fld = m_pcie_reg_fld[i]) begin
  fld.drive_reset();
  update();
end
m_drive_reset = 0;
```

Note that the register's associative array of contained fields has protected access and thus a duplication is not strictly required. Also, the pseudo-code above is simplified by not showing casting steps.

Following this approach, for some field A in some register and register block, the drive_reset () task eventually initiates a back door write on to the DUT primary input, fldA_i, as shown in figure 9. After driving reset all RAL bound inputs had known values.

D. Scoreboard

The UVM User's Guide notes that the mirrored value in the register is not intended as a scoreboard. The mirror value can "accurately predict the contents of registers that are not updated by the design [but] it cannot determine if an updated value is correct" [3].

The RAL model in our project is part of the data path, as shown in figure 1. As such, it was the most viable test bench component for checking correctness as well as timeliness. Therefore, the RAL model acted as scoreboard for configuration path testing.

III. Fields to Ports

A device under test (DUT) register contains a set of non-overlapping contiguous bit fields. Bit field ranges are mutually exclusive. The bit field in a single register is considered contiguous even if the implementation is not. In the code example below, a two-bit register field, A, may be implemented as disjoint wires within a DUT module.

```
wire fldAb1, fldAb0;
fldAb0 = wrAp ? HWDATA[0] : fldAb0;
fldAb1 = wrAp ? HWDATA[1] : fldAb1;
HRDATA[1:0] = rdAp ? {fldAb1, fldAb0} : 2'b0;
```

Similarly, the field may be described in disjoint register fields in one or more registers. In the code example below, the DUT implementation is contiguous.

```
wire [1:0] fldA;
fldA[1] = wrRg4Ab1p ? HWDATA[1] : fldA[1];
fldA[0] = wrRg6Ab0p ? HWDATA[1] : fldA[0];
HRDATA[1] = rdRg4Ab1p ? fldA[1] : 1'b0;
HRDATA[1] = rdRg6Ab0p ? fldA[0] : 1'b0;
```

Of course, a combination of these may be used.

For our PCI-Express project, the internal register implementation was drawn out to the interface as contiguous fields within the same port. While the internal architecture was difficult to externally peek or poke because of disjoint implementation, the interface more closely matched the register documentation. Additionally, the interface was well documented and tended to change slowly over the course of the project. This consistency eased verification environment implementation and maintenance between RTL releases.

Read and write back door paths had to be implemented separately in our project. All of the DUT ports were either input or output, not both. This somewhat deviates from normal RAL back door expectation of peek and poke of some storage data type (e.g. reg). To illustrate, consider figure 8 where the DUT output port is connected to a specific register field. When

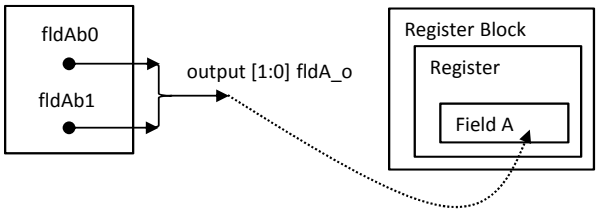

Fig. 8. Output port connected to internal DUT field.

test code reads the value, and assuming an update to the register is required, a back door read operation is initiated. When Direct Programming Interface (DPI) package is enabled in the UVM library (compiled when macro UVM_HDL_NO_DPI is undefined), then this action calls the UVM DPI function uvm_hdl_read. In figure 8, the back door read is indicated by the dotted line.

```
int uvm_hdl_read(string path,
                 output uvm_hdl_data_t value);
```

The register field is bound to a signal name in the test bench based on the string hierarchical path. This is a *soft bound* because the string may be composed by the test bench dynamically; it is not a bound made by the SystemVerilog compiler.

The UVM register block and UVM register each maintain an associative array mapping of path kinds to hierarchical paths. We utilized this mapping to distinguish writing to an input, as in figure 9, and reading from an output. At runtime,

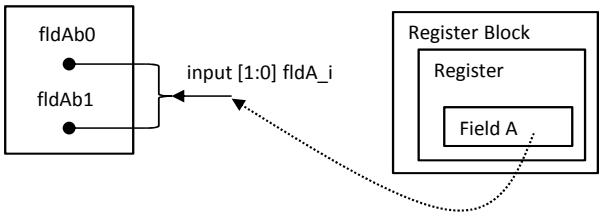

Fig. 9. Input port connected to internal DUT field.

the UVM register read or write task composes the hierarchical path to use in uvm_hdl_read based on the root path in one or more block parents and the register field signal. We split the mapping in both along access kind: "RTL_READ" to read from an output and "RTL_WRITE" to write to an input, as in figure 10.

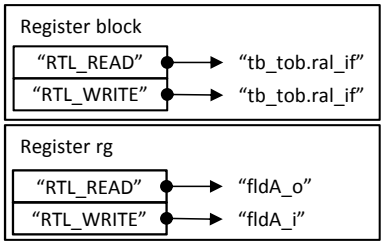

Fig. 10. String key indicates hierarchical path to use for back door access.

The signal paths were set in the register build function for each soft bound field. Refer to figure 10 for the example code below. Field A is a two-bit contiguous field in register rg starting at bit offset 1 (i.e. fldA = rg[2:1]).

```
rg.add_hdl_path_slice(.name("fldA_o[1:0]"),
                      .offset(1), .size(2),
                      .kind("RTL_READ"));
rg.add_hdl_path_slice(.name("fldA_i[1:0]"),
                      .offset(1), .size(2),
                      .kind("RTL_WRITE"));
```

The string path kind element is propagated up through the RAL model during composition. Therefore, the block root paths were set similarly.

```
blk.set_hdl_path_root(
        "tb_top.ral_if", "RTL_READ");
blk.set_hdl_path_root(
        "tb_top.ral_if", "RTL_WRITE");
```

We took the liberty of declaring all RAL-specific inputs and outputs within a single SytemVerilog interface instantiated alongside the DUT. Thus, for each register block in our RAL model, a single hierarchical root was required for each access kind.

Finally, we had to tie our path mapping together with default UVM register behavior so that back door access was handled without test code intervention. All register accesses execute with a reference to uvm_reg_item and through the Xcheck_accessX function. Furthermore, it is this function that checks the associative array mapping but only does so for the default path kind, "RTL."

```
function
bit Xcheck_accessX(uvm_reg_item rw, ...);
  if (rw.path == UVM_DEFAULT_PATH)
    rw.path = m_parent.get_default_path();
          // gotcha: locally protected!

   if (rw.path == UVM_BACKDOOR) begin
      if (get_backdoor() == null &&
          !has_hdl_path()) begin
          // gotcha: checks only "RTL"!
  ...
endfunction
```

It made sense to override this function in a new project-specific register base class. We created pcie_reg as extension to uvm_reg with a duplicate block parent reference and provided similar functionality.

```
function
void get_bd_kind(uvm_reg_item rw);
  if(rw.path == UVM_DEFAULT_PATH)
    rw.path =
        reg_block_parent.get_default_path();

  if(rw.path == UVM_BACKDOOR)
    case(rw.kind)
      UVM_READ, UVM_BURST_READ:
          rw.bd_kind = "RTL_READ";
      UVM_WRITE, UVM_BURST_WRITE:
          rw.bd_kind = "RTL_WRITE";
       default: rw.bd_kind = "";
    endcase
endfunction
```

Then we copied Xcheck_accessX into our class and connected our logic.

```
function
bit Xcheck_accessX(uvm_reg_item rw, ...);
  get_bd_kind(rw);
  if(get_backdoor() == null &&
     !has_hdl_path(rw.bd_kind))
                  // okay now
    ...
endfunction
```

Because the function was *copied* (read: kludged), for each new release of UVM we had to ensure compatibility. There was no issue between UVM-1.1c and UVM-1.1d, the two releases used in our project.

IV. Ports to Fields

Register fields with back door access may implement active monitoring of the bound signal to automatically mirror the DUT actual value in the register field value. The UVM library base class uvm_reg_backdoor provides the necessary hooks to implement active monitoring [3].

A few notes regarding this approach should be described. First, active monitoring in a register specific uvm_reg_backdoor extension hard binds the register field to the hierarchical path. The task below implements active monitoring for field, A, in some register (refer to figures 8 and 9).

```
class rg_backdoor extends uvm_reg_backdoor;
  task wait_for_change();
    @($root.tb_top.ral_if.fldA_o);
  endtask
endclass
```

The path to fldA_o is to the task by the compiler. This class is structurally associated only with the register containing field A. Note, too, that for each register with fields that have back door paths, a new class must be:

- defined,
- instantiated separate from the register,
- set as the register's back door access, and
- active monitoring thread started.

That is a lot to manage in addition to the regular RAL model.

Second, the back door access class for these registers are decoupled from the built-in UVM register back door behavior. The user's back door access methods are preferred, as in the register read execution path example below.

```
uvm_reg_backdoor bkdr = get_backdoor();
if (bkdr != null)
  bkdr.read(rw);
else
  backdoor_read(rw);
```

Thus, to enable active monitoring within the RAL scope the implementation must recreate *all* back door access. Together, these points lead us to employ a different approach.

Near the DUT, we implemented simple **always** @(..) blocks for actively monitoring read path signals. All RAL-specific inputs and outputs were declared in an interface instantiated alongside the DUT. For each "RTL_READ" path signal, a monitor was implemented directly in the interface. Back door register reads were initiated for any field value change.

```
bit monitor_master_enablep; // global

logic [1:0] fldA_o; // interface signal
bit rg_read_enablep, rg_request_readp;
bit rg_request_readp;

always @(fldA_o)
  if(monitor_master_enablep &&
     rg_read_enablep)
    rg_request_readp <= 1'b1;

always @(posedge rg_request_readp)
    pcie_cfg::DUT_RAL.blk.rg.read(...);
```

A single read request was implemented per register. Each field's read path signal, on value change, asserted the register's read request. This was a non-blocking action to ensure only one register read even when multiple fields asserted the read request in the same simulator time slice.

Fig. 11. Read events in active monitoring at the interface.

Each register read request was protected by two enables. A master enable turned on or off all active monitoring. This was critical to handle time zero signal change from unknown to known DUT output value, as illustrated in figure 11. The transition to a known signal value created a simulator event that caused a register read *prior* to RAL model instantiation. If not filtered then this first read reliably caused segmentation fault and core dump (of course we could have just existentially tested the RAL model reference). Also, we globally disabled active monitoring during RAL model reset to ensure no spurious register reads (refer to section II regarding driving reset). The master enable, itself, was mapped to a RAL register with back door access.

Finally, a register specific enable allowed for fine-grain active monitoring control. This enable was not mapped to any RAL register and was only provided for debug assist.

V. RAL Generation

The UVM RAL base code is generic enough to support many different types of projects. Because the implementation is in SystemVerilog and, importantly, public, it is straightforward to extend the base classes [1].

The divergence in methodology seems to be in the automatic generation of the project-specific RAL implementation. Several commercial tools exist such as Synopsys' ralgen, Semifore's CSRCompiler and Cadence's blueprint [12], [9], [5].[2] At base, some sort of specification must be translated into UVM-specific SystemVerilog code and instantiation in a test bench. Specification description varies as does the translation into source code.

We chose our register specification as the document to be delivered with our DUT to customers. This was to address time sensitivity in our schedule. Prior to the start of our project, the PCI-Express configuration register space was already specified in a Microsoft Excel spreadsheet (XLSX format), see figure 12. As with the automation tools, several proprietary languages

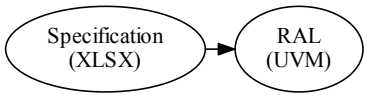

Fig. 12. Project required customer specification to translate to source.

exist to specify the RAL model, such as IEEE SystemRDL, Synopsys' RALF, and Semifore's CSR [10], [12], [9] . In all these cases, we would have had to introduce, in figure 12, an additional translation step between XLSX specification and UVM implementation. We did explore this avenue and found that we had to still manipulate the generated UVM source code to meet our requirements. Additionally, we found that direct translation from spreadsheet to source code was within our reach. To that end, we built scripts in Perl 5 utilizing a freely available XLSX reader to parse the spreadsheet and generate code [13], [7].

All registers in our DUT were specified in the format as shown in table I. We set up each worksheet in the XLSX file to represent a single UVM register block. Each register and field within a single worksheet would instantiate within that register block. Multiple register blocks could be instantiated in simulation (e.g., an array of blocks).

Our scripts operated in two passes. First all names and details of each register and register field were collected from the spreadsheet. Then, those details were output separately into:

- UVM RAL source code with Naturaldocs comments,
- Interface signals declaration, and
- Active monitor implementation.

A. UVM RAL Source Code

UVM RAL source code generation operated in the second pass of scripting. In the first pass, the spreadsheet was parsed

[2]Blueprint seems to have an uncertain future.

TABLE I
BASIC XLSX REGISTER SPECIFICATION.

Name	Acronym	Offset	Field	MSB	LSB	Access	Volatile	Rand Mode	IO	Read IO	Description
Example Reg	REG	0x010									Simple example
			A	1	0	RW	1	init_on	fldA_o	fldA_i	Field A primary input/output

to collect all details. Then, all class and function definitions were output to UVM SystemVerilog source code according to algorithm 1. This step was the most time consuming to implement with the algorithm worked out as required. Note that the UVM register field was constructed through the UVM factory. This allowed the constraints defined in the field to be overridden through normal UVM procedure.

B. NaturalDocs Comments

During UVM source code generation, the scripts inserted NaturalDocs-styled comments [2]. While the spreadsheet register specification offered one view of the register set, Naturaldocs comments provided a linked dynamic perspective with a focus on RAL model structure.

Referring to table I, a register class definition provided insight into RAL implementation. For example, the register class itself:

```
// Class: REG_pcie_reg
// Simple example.
// Offset: 0x010.
// Refer to: <blk_pcie_reg_block>
```

Then, for register field members within:

```
// Variable: A
// Field A primary input/output.
// Bits [1:0], randomized.
// Refer to:
//  - <REG_pcie_reg>
//  - <REG_A_pcie_reg_field>
```

This reference guide proved useful actual RAL model identifiers to use while coding.

C. Interface Signals

The SystemVerilog interface source code generation operated in the second pass of scripting. The spreadsheet was parsed in the first pass. Then, all signals listed in the IO and Read IO columns, see table I, were output to a file, according to algorithm 2. Some register fields considered differing ranges on the same interface signal. The scripts accounted for this by collecting all signals merging all ranges to determine the actual SystemVerilog interface signal. Ranges that overlapped indicated an error in the register specification (i.e., spreadsheet). Ranges that were not included were legal but indicated a warning.

D. Active Monitors

The SystemVerilog active monitor source code generation operated in the second pass of scripting. All details were collected from the spreadsheet in the first pass. Then, for each signal listed in the Read IO column, see table I, were output to

Algorithm 1: UVM Source code generation.

```
foreach register do
    foreach field do
        define register field class :
        begin
            declare valid constraint
            declare UVM automation
            define function new
            define function configure
        end
    end
    define register class :
    begin
        declare order constraint
        define function new :
        begin
            add self to block parent
            foreach field do
                configure field rand mode
            end
        end
        define function build :
        begin
            construct via UVM factory configure
        end
    end
end
define register block class :
begin
    declare order constraint
    define function new
    define function build :
    begin
        construct default map
        foreach register do
            construct
            build
            configure
            foreach field do
                add read IO path add write IO path
            end
        end
    end
end
```

Algorithm 2: Interface code generation.

```
foreach register do
    foreach field do
        if signal exists then merge signal range ;
        else add new signal and range ;
    end
end
foreach signal do
    if signal.size > 0 then
        logic [signal.msb : signal.lsb] signal.name ;
    else
        logic signal.name ;
    end
end
```

Algorithm 3: Active monitor generation.

```
foreach register do
    bit {reg.acronym + "_read_enp"} ;
    bit {reg.acronym + "_read_reqp"} ;
    always @( posedge {reg.acronym + "_read_reqp"} )
    begin
        reg.read(.path(UVM_BACKDOOR));
        {reg.acronym + "_read_reqp"} ← 0 ;
    end
    foreach field do
        always @( fld.RTL_READ_path )
            {reg.acronym + "_read_reqp"} ← 1 ;
    end
end
```

a file, according to algorithm 3. Refer to section IV regarding active monitoring.

VI. Conclusion

We presented an approach for UVM RAL model back door access to DUT primary inputs and outputs. Several challenges to this approach lead us to back door path mapping, driving reset to IOs, and active monitoring as solutions. Our implementation was simple *enough* to integrate within the existing RAL model flow with some extensions. The base test was aware of these extensions and required some special care to build the model and drive the reset values, but test code was oblivious.

The sheer volume of registers requires automation. There is a trade-off here between custom automation and third-party tool. Unless it is a company requirement to distribute one of the popular register formats, then translation into RALF, CSRSpec or SystemRDL is a schedule burden. Furthermore, if field level randomization and back door path mapping is not supported by the tool vendor for these formats, then customization should be considered. Nonetheless, it may be simpler for a one-off or short-lifespan project to randomize at the register and build components to interact with a generated RAL model. This was not the case for our project.

References

[1] Universal verification methodology reference implementation. http://www.accellera.org/downloads/standards/uvm, October 2010.

[2] Natural docs. http://www.naturaldocs.org//, 2011.

[3] Acellera. *Universal Verification Methodology (UVM) 1.1 User's Guide*, 2011.

[4] Cadence Design Systems, Inc. *SystemVerilog Reference*, 11.1 edition, March 2012.

[5] Denali Software, Inc. *Blueprint Compiler Users Guide*, 3.7.4 edition, June 2009.

[6] IEEE Computer Society. *SystemVerilog–Unified Hardware Design, Specification, and Verification Language*, 1800-2012 edition, 2012.

[7] D. Ovsyanko. Spreadsheet::xlsx - perl extension for reading ms excel 2007 files. http://search.cpan.org/~dmow/Spreadsheet-XLSX-0.13-withoutworldwriteables/lib/Spreadsheet/XLSX.pm, 2008.

[8] PCI-SIG. *PCI Express Base Specification Revision 3.0*, November 2010.

[9] Semifore, Inc. *Register Management with CSRCompiler, Reference Manual*, 2013.03 edition, February 2013.

[10] The SPIRIT Consortium. *SystemRDL v1.0: A specification for a Register Description Language*, 1.0 edition, 2009.

[11] Synopsys, Inc. *VCS MX/VCS MXi User Guide*, g-2012.09 edition, September 2012.

[12] Synopsys, Inc. *VMM Register Abstraction Layer User Guide*, 1.12 edition, March 2013.

[13] L. Wall. The perl programming language. http://www.perl.org/, 1994.