A Fully Reusable Register/Memory Access Solution: Using VMM RAL

Paul Lungu,
Bo Zhu

Nortel, Ottawa, Canada

ABSTRACT

Register structure and memory modeling is a very complex task of any verification methodology. Building a zero time mirror to check the correct functionality of every field from every register and memory is usually a very time-consuming process which needs to be repeated for every design.
This paper describes a new approach in resolving this issue by using the VMM Register Abstraction Layer (RAL). This is a fully reusable solution which allows automatic creation of a high-level object oriented abstraction layer for registers and memories in a DUT. RAL works with any transaction generator capable of performing basic writes and reads to and from the registers or memories. Both front-door and automatic/custom backdoor access solutions are analyzed in this paper as well as a memory modeling approach using register array structures from RAL.
A complete solution from spec definition to simulation using RAL is presented.
An example of the RAL application is provided in the form of a fully reusable register and memory access testcase using random constraints and coverage analysis in a real verification environment.

Table of Contents

Table of Figures

1. Introduction

Most of today's modern designs have hundreds if not thousands of registers and lots of on chip memory. Continuously increasing number of registers in a design makes register/memory verification a growing challenge for verification engineers writing manual register/memory test cases and test environments. Moreover, changing specs during the cycle of a design may result in unnecessary churn in documentation, verification environment and/or register/memory test cases. In turn, this significantly impacts the productivity, and increases the probability of introducing human errors in the process. Hence the need for a streamlined methodology and a tool which ties the spec to the verification environment and makes register/memory testing an automatic process. VMM Register Abstraction Layer (RAL) from Synopsys is a tool that addresses the above concerns and it was used in conducting register/memory test in an existing VMM environment. A fixed sequence of events and commands was defined in order to create the methodology around RAL.

1.1. RAL – Description and Main Features

VMM RAL is a VMM application package which helps create an object oriented abstraction layer to model registers and memories from a DUT. A RAL model is created from fields which are grouped in registers. Register and memories are grouped into blocks which are static entities with their own registers and memories and sometimes with their own verification. Blocks may be further grouped in systems. Thus, RAL is trying to mimic exact design hierarchy and naming of different design entities. For each element in a RAL model – field, register, block or system, there is a class which abstracts the read and write accesses to that element.

Users are encouraged to read [1] for more details about RAL structure. Also, an interesting paper [3] on RVM RAL has been published and the user may be interested to read it in order to have a complete picture of VMM RAL. It is not the goal of this paper to go into details previously published on RAL.

RAL has a number of features which help in DUT verification for complex designs that contain a large number of registers and/or memories. Some of them are presented below. However, a more comprehensive list is can be found in [1, 2, 3].

- **Abstract physical location of the registers and the fields**. – The user is not required to know the exact address of a register in the memory map or the exact position of a field in a register. The user has to know only the name of the register or field which are already part of the spec. RAL will identify the correct address and field position in the register based on the name provided by the user.

- **Flexibility and transparency in switching from frontdoor to backdoor accesses**. – This feature provides the user with a valuable tool for DUT initialization or an alternative to time consuming frontdoor accesses after complete equivalency between frontdoor and backdoor has been separately proven. This feature also has the potential to expose incorrect bus connections inside the DUT or between the DUT and verification environment. Potential issues may be triggered when using frontdoor and backdoor accesses in an alternative fashion for a specific access at a random address.
- **The content of registers is mirrored in an external data structure**. – the RAL abstraction model maintains a mirror with what it thinks it is the most up to date value of the registers inside the DUT. Register values can be accessed in zero time after a frontdoor access or DUT can be updated with the values from the mirror. RAL is also able to skip some updates of the DUT if the same value is required to be loaded into DUT from the mirror when using particular RAL commands which are specified later in this section. The advantage of this feature is the saving of unnecessary clock cycles usually wasted with updating the register or memory location with same value when this is not really intended. Resetting the abstraction model sets the mirror to the resets value specified in the spec.
- **Independence of tests and register specifications**. - Register fields can be moved between physical registers without changing the test. It is recommended to have different names for fields in a block. This is particularly useful when a register is moved from one block to another or when fields are reshuffled in within a register during project development. The test should not be affected by changes in the spec since it is independent of the exact location of the register in the memory map or the field in the register.

During the project development some other RAL features were found useful in building an automatic testbench:

- **Predefined register types.** – The user has the option to use some of the predefined register types as: `ro` (read only), `rw` (read/write), `w1c` (write 1 to clear), `wo` (write only), `rc` (read to clear) and much more, all described in detail in [1].
- **Access tasks.** – These are methods of accessing either DUT or RAL mirror. Some of the tasks proven to be most useful in this project are listed below. However, for a complete list refer to [1]. Moreover, [3] shows a partial list of access tasks which are among the most used ones.

    - **`read`** – This task gives the user the register value from the DUT. The register's mirror in RAL is automatically updated following the **read** task. This access automatically creates a read command on the physical bus into the DUT.
    - **`write`** – This task writes the data provided by the user into the register inside DUT. The register's mirror in RAL is automatically updated following the write. The task automatically creates a write command on the physical bus into the DUT.

- `update` – This task updates the values from the RAL mirror into the register inside the DUT. It can be applied at block level in which case it automatically updates all registers from that block or system level in which case it updates all blocks from the system. The command skips updating the registers from the DUT which have no mirror changes. This saves clock cycles for registers with no mirror update from the reset value.
- `mirror` – This task updates RAL mirror with the value of the register from inside DUT. It can be used to automatically check if the value retrieved from DUT compared correctly with the value from the mirror. This command will always perform a DUT physical access since RAL doesn't know what value will be read from the DUT. It exists at block level and if called it runs thru all registers from a block and updates their mirror accordingly. This task can also be called from system level in which case it loops thru all the blocks in a system.
- `get` – This task provides the value from RAL mirror (it can be done at register or field level). It does not create a physical access into the DUT. This is a zero time access task.
- `predict` – This task forces the value from the argument list into the RAL mirror. It does not create a physical access into the DUT. This is a zero time access task.

All above RAL features and some others specified in [1, 2, 3] make writing register and memory test cases an automatic process. The user will find the test is already updated once the spec has been changed and mirror recreated if some basic rules are followed in building the test case, or if an existing test case provided along with RAL is used as an example. There is no need to manually introduce additional code in the test if all registers or memories have a well predefined behaviour from RAL.

1.2.VMM, RAL and VCS release

All results published in this paper were obtained using System Verilog language, VMM release 1.3.13, RAL release 1.7.2 and VCS release 2006.06SP1-15.

2. RAL implementation in a real verification environment

RAL has been deployed in an existing VMM verification environment created around a DUT representing a traffic management device as shown in Figure 1**.**

It is known that RAL has most productivity impact for complex designs with at least 100 registers. At the time RAL was deployed on our design the spec was in progress and little was known about the total number of registers. The total number of registers ended up at around 160, which complied with the "100 register rule of thumb" for RAL. The memory used in the DUT had to be modeled as register arrays and tested as such. Memory field access was a requirement for the project, so a simple solution to access a memory field without too much manual decoding from the user was needed. With the DUT implementing 10 different logical blocks a similar approach has been followed in defining RAL hierarchy since each block in the DUT had its own set of registers and most of the time its own set of memories. The DUT implemented memories in 6 out of 10 of its blocks. The size of the each memory ranged from 1 to 64 Kbytes. The access size into DUT registers/memories was 32 bits (4 bytes). All registers had 32 bit access while the memory was organized into rows with data ranging from 4 to 64 bytes per row. Because of this variety of access sizes `VMM_RAL_DATA_WIDTH` parameter from RAL had to be defined in the `Makefile` to be 512 (bits) in order for the RAL mirror to be properly generated later (i.e. `+define+VMM_RAL_DATA_WIDTH=512`). RAL end users should read [1] for more details about the usage of this parameter including other ways of defining it in an existing VMM environment.

Based on the complexity of memory sizes and seeing the necessity to decode such a variety of accesses into DUT for every individual memory, RAL has proven to be a good choice in the end. Based on the requirements of the project a couple of RAL predefined register access types were implemented in the DUT as follows: `rw` (read/write) , `rc` (read to clear), `ro` (read only), `w1c` (write 1 to clear), `wo` (write only). Some registers with a different behaviour than RAL predefined types were assigned `user0` or `user1` types and they were treated as exceptions in the automatic test case or tested separately under a special setup based on the name of the register or by the type if all ended up in the same type category. These registers usually determined unexpected behaviour from DUT when some bits were written under normal operation. In order to simplify the verification process and debug later on in the development of the project, only one access type per register was allowed for each register or memory modeled as register array. If a new access type was needed in a register another separate register was created in the block.

As shown in Figure 1 software access into the DUT was performed via PCI Express bus. The verification environment implemented a PCI Express transactor capable of creating reads and writes from/to DUT on the bus. Only a single domain access was used in the verification environment. The RAL model will send read/write transactions to a 3rd party PCI Express bus functional model (BFM) which will access the registers and memories in the DUT. The RAL model also provides the verification environment with backdoor access to the registers and memories modeled as register arrays which will by-pass the physical interface in order to speed up simulations.

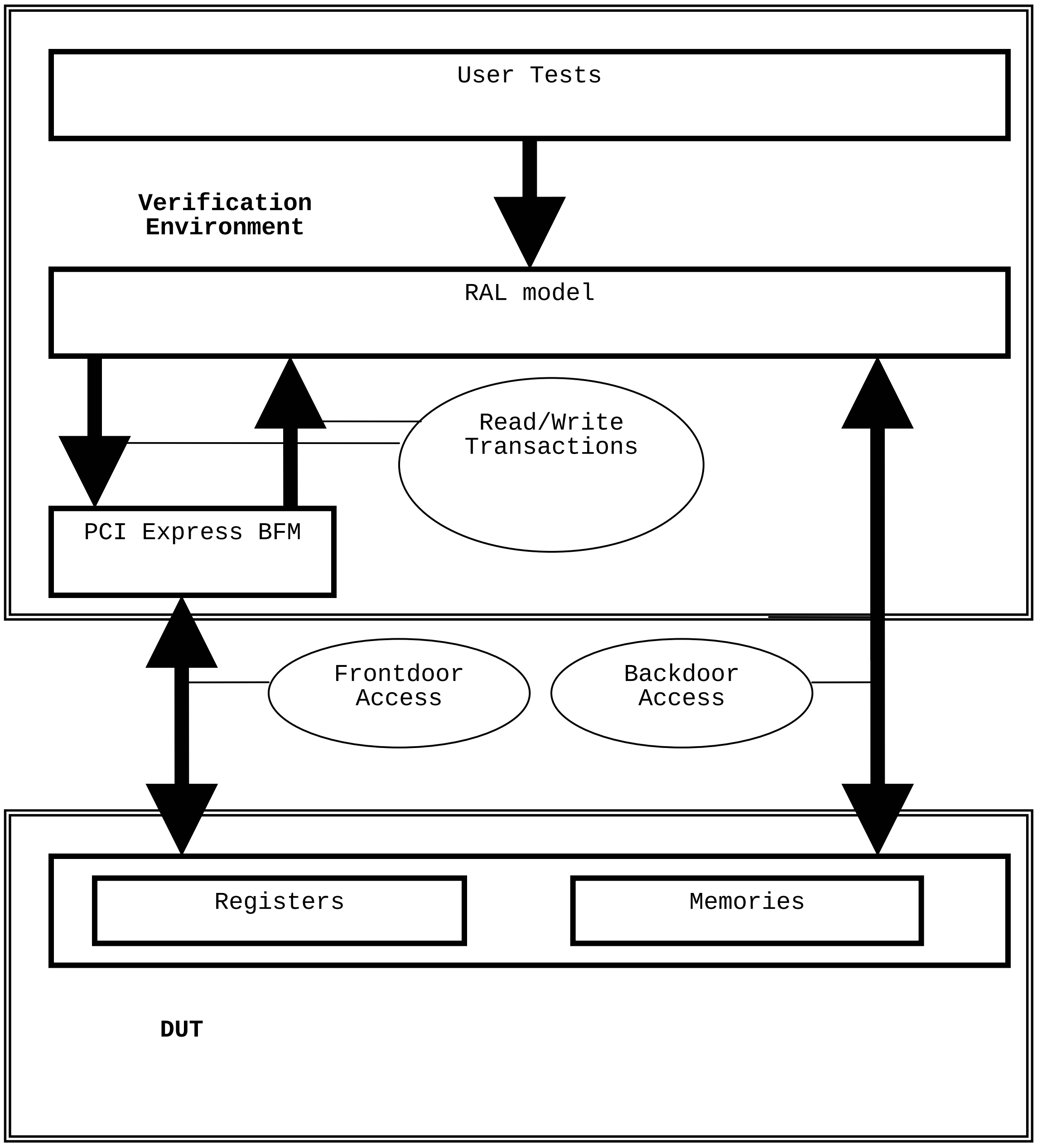

Figure 1. RAL implementation in a VMM environment

2.1. Resources Required to Integrate RAL

An existing VMM environment is among the first requirements for a RAL deployment. Some minor changes were needed to a classic VMM environment in order to accommodate RAL. The first change is reflected in Figure 2 below, and it refers to the use of `vmm_ral_env` base class instead of classical `vmm_env` base class in order to create the main VMM environment. Basically, this creates a RAL aware environment in VMM. RAL needs to be declared next in `dut_env` as the predefined class `ral_sys_dut`. RAL model is created in the new function of `dut_env` class as shown in Figure 2.

```
// build a RAL based env
class  dut_env extends vmm_ral_env;
…
// declare RAL class model
ral_sys_dut RAL;
…
endclass: dut_env

// create RAL model
function dut_env::new();
…
RAL=new();
ral.set_model(RAL);
…
endfunction: new
```

Figure 2. Building a RAL aware environment

Another challenge was creating a script to translate the spec into RAL files [1] accepted by the Synopsys' provided script (`ralgen`) to generate the RAL model. These two scripts together basically create a translation flow from spec to the RAL model. The information from every register from the spec needs to be translated into a class with members and tasks/functions operating on the members. This translation is not totally under the user's control. The script which translates from the spec to the RAL files (see Figure 1Figure 3) is fully under the user's control because of the large variety of spec formats available. The `ralgen` script is an executable and it takes previously generated RAL files as arguments and creates a RAL model (classes) which is not supposed to be altered by the user.

```
// script to parse the spec and generate RAL files
// this script is totally under user's control
>gen_regs

// script to parse previously generated RAL files and generate RAL model
//(classes)
// this script is an executable created by Synopsys; no user control
>$VCS_HOME/bin/ralgen –b –l sv –t dut_name –I ../ralf_files –ext_ud
dut_top.ralf
```

Figure 3. Scripts required in RAL integration

The RAL model is basically the bridge between the spec and the rest of the environment. In addition, it translates spec changes into automatic testcase updates since it contains the entire documentation written into the object oriented language. Once a process has been put in place to generate RAL files from the spec, the total number of registers in a project becomes irrelevant. The script is able to handle as many registers as possible as long as the correct syntax and rules to generate RAL files specified in [1] are followed. However, some issues may appear with regard to the total physical memory usage and computing resources if the number of registers is very high. Also the automatic RAL coverage model for such a memory map may prove challenging to build.

The final piece needed in RAL integration is a way to connect the above generated RAL model to the DUT. This is done via existing write/read tasks which the user needs to create in a transactor. The widely known VMM base class vmm_xactor from a classic VMM verification environment has to be changed only for this particular transactor to vmm_rw_xactor as shown in Figure 4. A RAL required task (ie. `execute_single()`) is the main communication point between RAL environment and the BFM used in `vmm_rw_xactor`.

```
class pcie extends vmm rw xactor;
…
extern task write();
extern task read();
extern task execute_single(vmm_rw_access tr);
…
endclass: pcie
```

Figure 4. Transactor class for writes/reads

This unique point of communication between the RAL model and the physical bus gives the user the flexibility to shift addresses generated by RAL environment before entering into the BFM. Basically the user has full control of addressing into the DUT, matching exactly the internal DUT implementation. This aspect will be detailed later on in the paper in 3.4

3. RAL Integration Flow: from spec definition to coverage feedback

Based on the experience acquired in this project, some general guidelines can be identified before initiating the work with RAL:

- Read the RAL user guide and fully understand rules to follow in defining the spec in order to comply with some RAL requirements. If the recommended rules are not followed there may be limitations as to what can be modeled in RAL. Spec definition is essential in success with RAL as the registers and memories should have a standard format that is strictly used throughout the spec.
- Create a script which is able to parse the spec and generate syntax correct RAL files.
- Be prepared to follow some guidelines of writing RTL code in DUT to comply with some backdoor requirements of RAL. Otherwise automatic backdoor capability becomes difficult to implement without extensive churn and rework.
- Change the existing VMM environment as mentioned in **Section** 2.1

3.1. Defining the Spec for Registers and Memories

As mentioned above, some rules have to be implemented when writing the spec in order to comply with some RAL requirements. The guidelines below were followed in writing the spec:

- All fields in within a block were required to have different names in order to ease automatic test case generation when fields were moved from one register to another inside of the same block. However, RAL can handle the case when same field name has been assigned to different registers. In this case the generated field names at the block level in the RAL mirror are still different and they have the name of the register appended to the field name.
- Register names have to be unique within a block and different than the field names. In order to avoid unnecessary renaming of unique fields in a register, upper case and lower case characters were used to differentiate between register names and field names. This allowed a register and a field to both have the same name since they will be different from the compiler's point of view.
- No SystemVerilog keywords are used in the name of a field, register or block.

In order to make the scripting and/or verification easier and still comply with project requirements some further guidelines were followed:

- No gaps between register addresses in memory map. This rule eased the testing of unused space in the memory map. Also writing coverage bins for the unused space was simplified by the adoption of this rule.
- No gaps between fields in the same register. Some exceptions were made here especially for the registers useful in the lab debug and for backward compatibility with software. This rule assisted later on in automatic backdoor implementation. More details in **Section 4.2.**
- Same field type inside a register. If a new field type was needed, a new register was created. This rule made user controlled script (`gen_regs`) easier to write.
- No field crossing between boundaries of an access was allowed for memories with a row larger than one single access (32 bits). Each field was defined inside one single 32 bit access region.
- Memories had same access type inside of a single processor access.

An example of a register spec with no gaps and another one with gaps between fields is shown in Figure 5. The spec of a memory is also illustrated there.

Register Name		DEVID	This register represents Device and Vendor ID.
Address		0x00000	
Type		RO	
Field	**Default**	**Name**	**Description**
31:16	0xabab	dev_id	Device ID
15:0	0xacac	vendor_id	Vendor ID

Register Name		QSTATM_ACC	This register is used indirect software access to queue statistics memory
Address		0x00501C	
Type		RW	
Field	**Default**	**Name**	**Description**
31	0x0	done_go	0:done; 1:go
30	0x0	read_wrt	0:read; 1:write
16:0	0x0	addr	Queue Stats Memory relative address

Memory Name			CWOLUTMEM	CW LUT Memory
Base Address			0x300000	
Rows			1024	
Bytes Per Row			4	
Field	**Type**	**Default**	**Name**	**Description**
14:8	RW	0	count_offset_1	Count Word Offset 1
6:0	RW	0	count_offset_0	Count Word Offset 0

Figure 5. Example of registers and a memory spec

DEVID is the spec for the register with contiguous fields (no gaps between two consecutive fields) while QSTATM_ACC is the spec for a register with non-contiguous fields (gaps between consecutive fields). CWOLUTMEM represents the spec for a memory with one single access per row.

It is very important to note that defining these rules before the spec is written saves a lot of time later on when RAL features are turned on (i.e. automatic backdoor). Creating a separate chapter in the software interface spec for each design block was another important decision made at the very beginning and it saved time later on during the development of the project when registers or memories changed for one block at a time rather than for the whole design. However, creating the RAL mirror is independent of the number of the spec files changing at one time. The Synopsys `ralgen` script generates one single file for the system anyway.

The appearance of the main RAL file (`dut.ralf`) depends on the way the spec was structured. Figure 6 shows the main RAL file used in the project after automatic backdoor feature was enabled. Having the flexibility to turn off the automatic backdoor generation in a block by simply removing the block from the main RAL file allowed the testing for the backdoor feature one block at a time.

```
source global reg.ralf
source sch_reg.ralf
source sch_mem.ralf
source qdr_reg.ralf
source frm_store_reg.ralf
source frm_store_mem.ralf
source adm_reg.ralf
source adm_mem.ralf
source qm_reg.ralf
source qm_mem.ralf
source stats_reg.ralf
source stats_mem.ralf
source tx_reg.ralf
source rx_reg.ralf
source script_eng_reg.ralf
source script_eng_mem.ralf
system  dut {
   bytes 4;
   block global_reg    =GLR (swif.swif_core_inst.pclk_regs)@'h000000;
   block qdr_reg       =QDRR(qdr_ctrl.dut_qdr_swif_inst)   @'h000800;
   block adm_reg       =ADMR(adm.dut_adm_swif)             @'h001000;
   block qm_reg        =QMR (qm.qm_swif)                   @'h002000;
   block frm_store_reg =FCR (fc_swif)                      @'h003000;
   block sch_reg       =SCHR(sch.sch_swif)                 @'h004000;
   block stats_reg     =STATR(stats.stats_swif)            @'h005000;
   block rx_reg        =RXR (dut_rx_swif_inst)             @'h006000;
   block tx_reg        =TXR (ross_tx_swif_inst)            @'h007000;
   block script_eng_reg=SER (swif.se.se_swif)              @'h007800;
   block script_eng_mem=SEM                                @'h100000;
   block frm_store_mem =FRMM                               @'h200000;
   block stats_mem     =STATM                              @'h300000;
   block adm_mem       =ADMM                               @'h400000;
   block qm_mem        =QMM                                @'h500000;
   block sch_mem       =SCHM                               @'h600000;
}
```

Figure 6. Main RAL file

3.2. Generated RAL files

Each independent RAL file in Figure 6 was created from the spec of the individual block. The specs for the blocks were written with Frame Maker in .fm format. The .fm files were first exported into .mif files. The user controlled script (`gen_regs` from Figure 3) translated the .mif files into RAL file format.

Figure 7 shows an example of a RAL file created by the `ralgen` script for `global_reg` (GLR) block containing the register with no gaps between fields with the spec described in Figure 5. Figure 8 shows the RAL file for the block `stats_reg` (STATSR) containing the register with non-contiguous fields with the spec described in Figure 5.

Figure 7 provides the example of DEVID register made of “ro” (read only) type fields. The reset value can be assigned for each field using the keyword “reset”. Also the RAL file specifies the number of bits used to define each field. This register, for instance, uses 16 bits to define each field. The beginning of the fields is also shown in the RAL file and it is assigned by using “@0 or @16” syntax. This specifies what bit position the next field starts.

In Figure 8 QSTATM_ACC register with non-contiguous fields is illustrated. The first field which begins @0 is 17 bits wide while next field which begins at @30 is 1 bit wide. The last bit starts @31 and it is also 1 bit width. This type of register makes up to 5% of the spec and has a different format than the rest of the registers in the device. This was mostly because of the backdoor description in RAL file which will be discussed in more detail in **Section 4.2**.

```
block global reg {
   bytes 4;
   register DEVID (r_DEVID) @'h00 {
      bytes 4;
      field vendor_id @0 {
         access ro;
         reset 16'hacac;
         bits 16;
         constraint valid {
         }
      }
      field dev_id @16 {
         access ro;
         reset 16'habab;
         bits 16;
         constraint valid {
         }
      }
   }
}
```

Figure 7. RAL file for register with contiguous fields

```
block stats reg {
bytes 4;
register QSTATM_ACC @'h1c {
      bytes 4;
      field addr (r_QSTATM_ACC_addr) @0 {
         access rw;
         reset 17'h0;
         bits 17;
         constraint valid {
         }
      }
      field read_wrt (r_QSTATM_ACC_read_wrt) @30 {
         access rw;
         reset 1'h0;
         bits 1;
         constraint valid {
         }
      }
      field done_gone (r_QSTATM_ACC_done_gone) @31 {
         access rw;
         reset 1'h0;
         bits 1;
         constraint valid {
         }
      }
   }
}
```

Figure 8. RAL file for register with non contiguous fields

Registers were implemented by specifying the offset from the base address. Thus, the user has control of the address of each register in the memory map. For DEVID register, the offset was specified @'h00 while for QSTATM_ACC, the offset was specified @'h1c in Figure 8 above.

Moreover, the name of the register inside of the DUT was selected to be `'r_name'` simplifying the scripting and making the rule easy to follow. In general, it is difficult to create such rules for a design team if they are not easy to follow.

3.3. Memories as Register Arrays

Because of project requirements, field access is widely used in verification environments testing this type of design. This presented a need for a tool and a method to write or read the value or field from a memory without performing too much address decoding.

```
block stats mem {
   bytes 4;
   endian big;
   register CWOLUTMEM[1024] @'h000000 {
      bytes 4;
      field count_offset_0 @0 {
         access rw;
         reset 7'h0;
         bits 7;
         constraint valid {
         }
      }
      field count_offset_1 @8 {
         access rw;
         reset 7'h0;
         bits 7;
         constraint valid {
         }
      }
   }
}
```

Figure 9. ral generated file for a standard memory as register array

RAL implemented the concept of a memory with `vmm_ral_mem` class. However, the user cannot access fields in this class. A solution using `vmm_ral_reg` class combined with the use of register arrays was considered and basically all memories were treated as register arrays with all the benefit this class has to offer.

Large memories in this design created concerns that the physical memory allocated for their RAL mirror may be too large and available computing resources quickly exhausted. After instantiating the biggest memories in the design and running a simple test to create a RAL mirror it was proven that the maximum physical memory usage did not exceed 250MB which was considered acceptable. Figure 9 shows the RAL generated file for a memory. It is important to mention that no field crosses two 32 bit access regions for memories with a row bigger than one single access (8 bytes or more). However, a big memory leak appeared when trying to turn coverage on for the RAL model. It was proven that the coverage model wasn't quite ready at that time and it was turned off back waiting for a fix from Synopsys. The fix came in later on in a subsequent VMM/RAL release but it was still unusable because of the large coverage database size created at runtime. Synopsys found that it took about 7.4 minutes to dump 278MB of coverage information from all of 16480 covergroups from the entire memory map. Most of these covergroups were part of the register arrays.

The RAL capability to internally decode the address of an access when a field is selected by name aided in implementing memory accesses based on field selection. In the old days, verification engineers had to decode all these accesses based on the field position in the memory row and assign the correct processor address for the transaction. RAL made all of this transparent for the user. The user is only required to know the field name from the spec. An important saving in implementation and debug time is realized by having this feature implemented in RAL.

3.4. Connecting RAL to a BFM generating valid reads/writes

The connection between RAL and the transactor is described in the PCIe class (see Figure 4). The PCIe class is extended from `vmm_rw_xactor` and implements the `execute_single()` task. This section details the composition of this task.

As Figure 10 indicates, the user needs two tasks to perform writes and reads. These tasks can be built either in a BFM or they can be built around a 3rd party IP core which basically does the transaction. The project used 3rd party verification IP to create PCIe transactions to the DUT, thus, the `read()` and `write()` were built around predefined read/write tasks. This is a very simple solution and reduces everything to the two tasks necessary to build any access in the transactor.

For register arrays, RAL generates consecutive addresses in increments of 1, starting from the base address for each single 32 bit access. In order to match RAL generated address to the address map from the spec and RTL implementation, some address shifting was needed before the correct address (increment of 4 per 32 bits access) was placed on the bus. A simple decoder was put in place for memory decoding, although not shown entirely in Figure 10. In essence, a shift by 2 to the left of the address generated by RAL was performed and the final address became identical with the address shown in the spec. Register decoding was unnecessary since all registers had fixed offsets in RAL files so the generated files obeyed the spec in this case. Unfortunately, forcing the address for each memory row or each single access is not possible for register arrays. This would be a nice feature to have in a future RAL release. It may give more flexibility to the user.

```
task pcie::execute_single(vmm_rw_access tr);
        // address decoder used to correctly shift addresses
        // used for memory access decoding only
        pkt.addr=tr.addr<<2;
        pkt.trans=tr.kind;
        pkt.valid=1;

        if(tr.kind == vmm_rw::WRITE)
        begin
                pkt.compl=0;
                pkt.data=tr.data;
                write();
        end
        else if(tr.kind == vmm_rw::READ)
        begin
                read();
                tr.data=pkt.data;
        end
   endtask
```

Figure 10. `execute_single()` task from `pcie` class

3.5. Defining the DUT Initialization Method

One of the most important steps in creating a verification environment is defining the initialization method of all internal registers and configurable memories from DUT. However, this may be one of the most time consuming parts of the simulation if all registers or memories are accessed frontdoor.
RAL provides the capability to update all registers in one shot by using `RAL.update()` but this may also prove very time consuming if every register has to be initialized again with its own reset value. At this stage one of the RAL features came nicely into play: the `update()` task issues a physical write only if the value from the mirror is different than the previously written value into DUT at the same address. If the `update()` task is used immediately after reset to write a register with a mirror that hasn't changed from the reset value, no processor write is issued. This simplifies the coding of the initialization task.

Initialization can be performed in different ways from project to project. However, in order to benefit from RAL features, external constraints were used for the fields modified after reset. Figure 11 shows the user defined constraints in the generated classes for the registers and the memory with the spec shown earlier in Figure 5. Predefined constraints for each field were not used in this project in order to unload the script to generate RAL files.

```
class ral_reg_global_reg_DEVID extends vmm_ral_reg;
        rand vmm_ral_field vendor_id;
        rand vmm_ral_field dev_id;

        local ral_cvr_reg_global_reg_DEVID reg_bits[*];
        constraint vendor_id_valid {
        }
        constraint dev_id_valid {
        }

        constraint user_defined;

        function new(vmm_ral_block parent, string name,
bit[`VMM_RAL_ADDR_WIDTH-1:0] offset, string domain, int cvr,bit[1:0] rights =
2'b11, bit unmapped = 0);
                super.new(parent, name, 32, offset, domain, cvr, rights,
unmapped);
                this.vendor_id = new(this, "vendor_id", 16, vmm_ral::RO,
16'hacac, 16'hx, 0, 1, cvr);
                this.dev_id = new(this, "dev_id", 16, vmm_ral::RO,
16'habab, 16'hx, 16, 1, cvr);
                Xadd_constraintsX("vendor_id_valid");
                Xadd_constraintsX("dev_id_valid");
                Xadd_constraintsX("user_defined");
        endfunction: new

endclass : ral_reg_global_reg_DEVID
```

```
class ral_reg_stats_mem_CWOLUTMEM extends vmm_ral_reg;
        rand vmm_ral_field count_offset_0;
        rand vmm_ral_field count_offset_1;

        constraint count_offset_0_valid {
        }
        constraint count_offset_1_valid {
        }

        constraint user_defined;

        function new(vmm_ral_block parent, string name,
bit[`VMM_RAL_ADDR_WIDTH-1:0] offset, string domain, bit cvr,
                                bit[1:0] rights = 2'b11, bit unmapped = 0);
                super.new(parent, name, 32, offset, domain, cvr, rights,
unmapped);
                this.count_offset_0 = new(this, "count_offset_0", 7,
vmm_ral::RW, 7'h0, 7'hx, 0, 1, cvr);
                this.count_offset_1 = new(this, "count_offset_1", 7,
vmm_ral::RW, 7'h0, 7'hx, 8, 1, cvr);
                Xadd_constraintsX("count_offset_0_valid");
                Xadd_constraintsX("count_offset_1_valid");
                Xadd_constraintsX("user_defined");
        endfunction: new
endclass : ral_reg_stats_mem_CWOLUTMEM
```

Figure 11. Generated classes for registers and memories

For the initialization portion of the device, the user defined constraint was used in a separate file to additionally constraint the fields which required different values after reset. The switch option `-ext_ud` needs to be passed as an argument to the `ralgen` script in order to create this additional user defined constraint in the generated files.

An example of an external user defined constraint is provided in Figure 12 below. The constraints were applied at the block level as opposed to the register level using the block level fields. The constraint from the example is applied to fields from CWOLUTMEM register array from Figure 11 and it restricts the fields from being generated equal during the randomization. The class `ral_block_stats_mem` was used instead of `ral_reg_stats_mem_CWOLUTMEM` to define the user defined constraint. RAL allows the constraint to be defined at the register level as well, so a constraint `ral_reg_stats_mem_CWOLUTMEM::user_defined` is allowed and identical.

```
constraint ral block stats mem::user defined
{
        foreach(count_offset_0[i])
                count_offset_0[i].value != count_offset_1[i].value;
}
```

Figure 12. External user_defined constraint defined in external file

Minimal work was required to create the automatic process to initialize the DUT based on the constraint set in these external files. Some blocks were excluded from randomization process since they worked directly with the reset values, so no additional initialization was required. The skipping process was based on the name of the block as defined in the main RAL file and it is shown in Figure 13 below. A `foreach` loop thru all registers in the DUT and the `update()` task at register level was used to configure the memory map backdoor . The more elegant RAL.update() task at system level, which internally loops thru all internal blocks, was initially used as "one line only command" to update the memory map but it didn't work as expected in backdoor mode with current VMM/RAL release. Synopsys is looking into this issue.

```
RAL.get blocks(blocks,domains);
        foreach(blocks[i])
        begin
                case(blocks[i].get_name())
                        "FCR",
                        "QDRR",
                        "QMM",
                        "FRMM",
                        "SEM",
                        "SCHM",
                        "TXR":
                        begin
                        end
                        default:        blocks[i].randomize();
                endcase
        end

RAL.update(status, door);
```

Figure 13. Automatic DUT initialization

This method makes the DUT initialization procedure fully tied to the spec changes and a set of external constraints which are completely under the user's control. The external set of constraints was created to match the blocks used in the design and the main RAL file presented in Figure 6. In addition, the decision whether or not to randomize a block or a register can be very easily

implemented in RAL, as only the name is required. This is a fully reusable initialization method in future projects where only the name of the register or block will change and the rest of the infrastructure remains the same.
It is important to note that some registers in some blocks required frontdoor only initialization as they were registers inside of IP models from a 3rd party vendor. In this case automatic initialization was performed first in backdoor mode and then the registers were initialized frontdoor using `write()` command instead of `update()`. The `update()` command may prove troubling to use in this case after backdoor initialization was performed with random data. If the value of a register initialized frontdoor happens to be equal with the backdoor written value `update()` is missed so no physical write is performed. The verification team actually used the same coding style for write compared with `update()` with a slight change in the argument list:

```
RAL.DUT.reg_name.field1.set(value);
write(status,RAL.DUT.reg_name.get(),door);
```

Using `write()` instead of `update()` in this case forces a physical write on the bus regardless of any previous history for the register.

3.6. Backdoor Transactions

While the initialization process is very easy to code, it may prove to be very simulation intensive if all registers and memories are accessed via frontdoor access. Each processor access generally takes a couple of clocks to read/write a register and sometimes a couple of hundred clocks to read/write a memory location due to internal implementation of the memories in the DUT. Therefore, a method of accessing the registers and memories that dramatically reduces the initialization time would represent a major improvement. Because of actual RAL limitations, automatic backdoor solution is not possible for memories implemented as register arrays because of the internal RAL architecture. Hopefully, future RAL versions will address this limitation. However, automatic backdoor is possible for registers, details of which are provided in **Section 4.1**.

A custom backdoor solution suggested by [1] was used for memories and it is detailed in **Section 4.2**. In order to deploy a backdoor solution in majority of the tests, an equivalence check of the frontdoor and backdoor methods was required. The equivalence check was built in the register and memory testcases, and it is further detailed in **Section 4.3**.

3.7. Automatic Register/Memory Testcases

Given the simplicity of the initialization method using RAL one may think that writing a testcase is a trivial thing and it is just one step added on top of the initialization. This statement is partially true if the DUT is simple and the registers are just control registers with static functionality, which means that writing into a register triggers no unpredictable action in DUT.

This DUT was complex enough that some registers were not static and once written into them, they trigger different responses from DUT which were difficult to model. Therefore they were treated and tested separately, not in the register test.

A similar rational applied to memories. While the majority of memories and registers were tested in a simple `foreach` loop, some exceptions were handled separately. The test that ensured correct functionality of the registers/memories contained three steps:

1) Test of the reset values.
2) Test of the write/read functionality of the register/memory.
3) Test of the unused memory space which can be easily implemented in a `foreach` loop using RAL tasks and features.

Each of these steps is detailed below:

1) The reset test was based on the reset value of the RAL mirror, which automatically takes the default values for each register/memory from the spec. A foreach loop was used to go thru each entry one by one and check the compliance. The `mirror()` task was used to ensure the read value from DUT matches the reset value of DUT from the mirror. Figure 14 provides an example of reset test using RAL.

```
RAL.reset()

foreach (blocks[i])
begin
        blocks[i].get_registers(regs);
        foreach (regs[j])
        begin
                case(regs[j].get_name())
                        "BYTE_SWAP_EN",
                        "QDR_STAT_INIT_END":
                                no_reg_test=1;
                        default:no_reg_test=0;
                endcase
                if(!no_reg_test)
                begin
                        reg_cfg.shuffle_door(0);
                        regs[j].mirror(status, vmm_ral::VERB,
env.door);
                        regs[j].display();
                end
           end
end
```

Figure 14. Reset test using RAL

As Figure 14 illustrates, some registers were not checked against the mirror because of their special nature. Thus, they were configured immediately after reset and their default value was not important in the correct functionality of DUT. Moreover, the RAL mirror was reset at the beginning of the test in order to ensure it was loaded with proper spec values from RAL files. A `shuffle_door`() task was used in this testcase in order to randomize the type of access (frontdoor or backdoor) for the read operation from the registers. This is part of frontdoor backdoor equivalency discussed more in detail in **Section 4.3**. The `mirror()` task used in this test has a built in check when using `vmm_ral::VERB` as argument. Thus, it will fail if any read value is different than the value specified in the spec.

2) The write/read test will ensure correct field access functionality for each field in the spec. For instance:

- A "RO" field once written into will never update with the value the user tried to write.
- Also, a "RW" field always updates with the value the user requires to be written into the field.

Starting from this "per field" access type model the test will loop for all the fields in the spec and apply random values or dynamically constrained values in order to ensure maximum coverage. **Figure 15** shows the simplified write portion of a write/read test.

The exclusion list increased in this step because more registers were sensitive to write operations rather than reads. Most fields were randomized using patterns with a given probability in order to maximize the coverage. Other fields from some registers had to be tested constantly with the same value to avoid unexpected behaviour from DUT. These types of registers were separately tested for the other value(s) of the field(s).

As in the previous reset test, a random access door is set before `write()` task is called. However, frontdoor only access needs to be used in some cases in order to trigger correct functionality inside DUT (in **Figure 15** see `vmm_ral::BFM` for W1C type registers). In this case if a backdoor would be used to clear a register nothing would happen in DUT since a physical processor write with the value "1" in each field has to be detected before the register is cleared.

Changing between backdoor and frontdoor was done by randomly switching between `vmm_ral::BFM` and `vmm_ral::BACKDOOR` as arguments of `write()` and `mirror()` tasks. In addition, some fields, which cannot be predicted by RAL after a write of a random value, need to be predicted manually and the check mirror needs to be overwritten to the predicted value before `mirror()` task is called. FSMERRCNT is one of a few registers with unpredictable functionality in DUT, as it depends of other registers.

As it can easily be seen in **Figure 15** the `write()` task was used instead of the `update()` task in order to ensure the random value goes thru regardless of the previous access type and value written into the specific register.

3) For this particular project, the unused memory space outside of the registers and memories had to be tested and some interrupts asserted when an access was placed in this area. All processor accesses outside the memory map modeled by RAL, should be done using direct BFM accesses not RAL accesses. RAL has no knowledge about a memory map outside the regions defined in RAL files and its area of competence is limited to this space. In order to write and read to/from outside RAL space processor accesses like `pci.write()` and `pci.read()`

were used in order to access this address space. RAL determined the minimum and maximum addresses for this unused space but cannot perform the access by itself.

```
foreach (regs[j])
begin
 // get all fields in register[j]
 regs[j].get_fields(fields);
 // make a list with no test registers
 case(regs[j].get_name())
  "INT_COLLECTOR",
  "PCIE_INT",
  "PLL_RESET",
  "RWP_LOCK":no_reg_test=1;
  default:        no_reg_test=0;
 endcase
 if(!no_reg_test)
 begin
  foreach(fields[k])
   begin
    case(fields[k].get_name())
     "crf_en": fields[k].randomize() with { value inside {0}; }
     default: fields[k].randomize() with {
                  value  dist { [1:'hfffffffe]:/1,
                                'hffffffff:=1,
                                'haaaaaaaa:=1,
                                'h55555555:=1];   };
    endcase
   end
  end
  case(regs[j].get_name())
   "FSMERRCNT":
     if(regs[j].get()>0)
     begin
      env.RAL.FCMR.FSMERRADR.addr.predict(blocks[i].get_base_addr());
      env.RAL.FCMR.FSMERRINF.addr_mask.predict('h1f);
     end
  endcase
  reg_cfg.shuffle_door(1);
  if(fields[0].get_access() == vmm_ral::W1C)
   regs[j].write(status,regs[j].get(),vmm_ral::BFM);
  else
   regs[j].write(status,regs[j].get(),env.door);
end
```

Figure 15. Field write access type test using RAL

3.8. Coverage analysis and feedback

At the time this paper was written a re-architected, coverage model for RAL was in development and testing was being done with an alpha release. The initial release built a coverage model that created a memory leak when used with large register arrays and it was considered unusable. The

RAL model under test now had coverage only for the bits making up the used fields and not for unused space inside the registers. This functionality was considered satisfactory for this stage of the project. Results of the coverage runs will be given after the integration of the new RAL model is completed. Due to the lack of this coverage model a fully random test using RAL was not possible but solutions were found to check coverage for unused memory space testing. A manual coverage model with addresses from the spec was defined and put in place. A portion of the `covergroup` will be given in Figure 16 as an example of how such as coverage can be written using RAL fields.

```
class reg test cov;

  dut_env env;

  covergroup register_cov;

    // check invalid access coverage
    invalid_glr: coverpoint env.pci.pkt.addr
    {
        bins   low = { ['h38:('h38+5*'h4)] };
        bins   med = { [('h38+6*'h4):('h7fc-5*'h4)] };
        bins   high = { [('h7fc-5*'h4):'h7fc] };
    }
    invalid_write:  coverpoint env.pci.pkt.trans
    {
        bins    write = {1};
    }
    invalid_read:  coverpoint env.pci.pkt.trans
    {
        bins    read = {0};
    }
    glrXread:       cross invalid_glr,      invalid_read;
    rwp_int_en:     coverpoint env.RAL.GLR.PCIE_INT_EN.pcie_rwp_en.value
    {
        bins disabled = {0};
        bins enabled = {1};
    }
    rwp_int:        coverpoint env.RAL.GLR.PCIE_INT.pcie_rwp.value
    {
        bins    disabled = {0};
        bins    enabled = {1};
    }
    rwpX:           cross rwp_access,rwp_int_en,rwp_int;
  endgroup: register_cov

  function new(dut_env env);
        this.env=env;
        register_cov = new();
  endfunction

endclass: reg_test_cov
```

Figure 16. Coverage model using RAL

After running the test a couple of times the coverage was collected automatically and then analyzed with `URG` tool from Synopsys. Figure 17 shows a result of the coverage based for the `bins` created in the `covergroup` from Figure 16. It can be easily seen that meaningful coverage can be obtained using RAL fields in the `covergroups`. Basically RAL variables offer a great way of getting coverage information from inside of the mirror. However, this information needs to be extracted in custom `covergroups` for the coverage info not strictly related to a register.

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
invalid_glr	3	0	3	100.00	100	1
invalid_write	1	0	1	100.00	100	1
invalid_read	1	0	1	100.00	100	1
rwp_access	2	0	2	100.00	100	1
rwp_int_en	2	0	2	100.00	100	1
rwp_int	2	0	2	100.00	100	1
glrXread	3	1	2	66.67	100	1
glrXwrite	3	2	1	33.33	100	1
rwpX	8	5	3	37.50	100	1

Figure 17. Example of coverage results for a limited number of RAL variables

4. Increasing productivity and simulation speed with RAL backdoor

When the project started no backdoor solution was available, so the verification team concentrated mostly on having the frontdoor access solution stable. However, the RAL user guide specified the rules for having backdoor created way before the real product was in place. The design had to be written with that in mind and some coding guidelines followed while writing RTL code in DUT. This will be discussed more in detail in **Section 4.4**. Once the backdoor release arrived, which was in alpha stage at the time the integration took place, the RAL model was properly created and integrated in verification environment after a couple of iterations of the scripts fromm Synopsys. A switch had to be created in order to transparently change the access door from frontdoor to backdoor and vice versa.
Running a test with backdoor access for registers reduced the simulation time to zero for register updates while it can take a significant number of real clocks to initialize all the registers frontdoor. Because the access times were very long to memories due to the internal design limitations a frontdoor initialization was not a solution and only backdoor access was used. The amount of work involved to have backdoor work for both automatic and custom solutions is minimal compared with the immense simulation time gain. Using RAL to create the backdoor access was proven to be a good solution even if the path to success was not that smooth due to

the alpha status of the RAL code at the time backdoor was tested and integrated. What used to be a work of a couple of verification engineers in the past was reduced to writing a smart script and following some guidelines in coding the design to comply with RAL requirements. Also the RAL capability to automatically create the correct access in the memory row when a field was selected saved a significant amount of coding time necessary to properly decode the accesses based on the address and also saved debug time associated with writing of the new code.
The most important and visible gain in simulation time was noticed in running regressions when the initialization phase was repeated over and over again by all tests.

4.1. Building Design for RAL Automatic Backdoor Solution

Final success for automatic backdoor was possible mostly due to strict coding styles while writing registers in RTL. Two types of registers were identified during the process of documenting the software interface:

1. Registers for which the fields inside were written in contiguous manner (no gaps between two consecutive fields). As it can be seen from Figure 7 and Figure 18 both fields of this register are collapsed into 32 bit field basically using maximum of flops. For shorter fields not all 32 bits may be used minimizing the number of flops used in design.
2. Registers which by design needed to have some fields in already specified positions basically allowing spaces between two consecutive fields. See Figure 8 and Figure 19. For registers from #1 the number of flops used in RTL was identical with the total number of used bits but all fields used same register as it can be seen in Figure 18. For registers from #2 the number of flops used in RTL to define the register was identical with the number of bits in all fields but each field had its own flops being declared individually as it can be seen in.

```
logic [31:0] r DEVID;
```

Figure 18. RTL representation of a register with contiguous fields

```
logic   [16:0]  r QSTATM ACC addr;
logic           r_QSTATM_ACC_read_wrt;
logic           r QSTATM ACC done gone;
```

Figure 19. RTL representation of a register with non-contiguous fields

As it can be seen from Figure 18 and Figure 19 in order to minimize scripting `'r_name'` was used as a rule of defining the registers in RTL. The above defined registers were concatenated with the path given in main RAL file as shown in Figure 7Figure 6.
Another rule from [1] which had to be followed while coding registers in RTL was to create real registers for the types which didn't really need a register in RTL due to its nature. For example

some registers which triggered some actions by writing a given value in it they didn't need flops to store that value in, so a register wasn't physically needed. In order for the backdoor to work flops were added and the value of the register had to hold reset value at all the times which was in line with spec requirements and it satisfied RAL requirements too. Also, registers which were supposed to hold a fixed value for the duration of the simulation usually don't need flops in RTL. The fixed value is directly wired to the read value when the address of that specific register is accessed. In order for RAL backdoor to work these registers needed real flops in place so the coding had to be changed to accommodate this requirement. Basically RAL automatic backdoor needed the name of the register to be accesses at all times inside the DUT.

4.2. Automatic Backdoor for Registers

Once the rules to write the design for automatic backdoor were respected and the script was modified to add the backdoor path for registers inside the block a couple of actions were needed to complete the automatic backdoor generation: the paths to the block containing the register definitions had to be defined in main RAL file (see Figure 6**)**. Also a new define statement was added into the `Makefile` in order to create the common root path to all blocks in the design:

`+define+DUT_TOP_PATH=dut_top.dut` .

The above Verilog path is automatically concatenated with the paths from the blocks manually created in main RAL file and then with the automatically generated register name from each individual RAL file for registers with contiguous fields, or in the case of registers with non-contiguous fields the path is concatenated with the name of the field.

An example for automatic backdoor in case of registers with contiguous fields is presented below:

The backdoor path to `r_DEVID` physical register in RTL as illustrated in Figure 18 will be created as follows:

```
`DUT_TOP_PATH.swif.swif_core_inst.pclk_regs.r_DEVID
```

The read/write tasks are created by the `ralgen` script as seen in Figure 20 below:

```
class ral reg dut GLR DEVID bkdr extends vmm ral reg backdoor;
        virtual task read(output vmm_rw::status_e status,
output bit
[`VMM_RAL_DATA_WIDTH-1:0] data,
                                        input int data_id,
input int
scenario_id, input int stream_id);
                data =
`DUT_TOP_PATH.swif.swif_core_inst.pclk_regs.r_DEVID;
                status = vmm_rw::IS_OK;
        endtask

        virtual task write(output vmm_rw::status_e status,
input bit
[`VMM_RAL_DATA_WIDTH-1:0] data,
                                        input int data_id,
input int
scenario_id, input int stream_id);
                status = vmm_rw::ERROR;
        endtask
endclass

class ral_reg_dut_STATR_QSTATM_ACC_bkdr extends
vmm_ral_reg_backdoor;
        virtual task read(output vmm_rw::status_e status,
output bit
[`VMM_RAL_DATA_WIDTH-1:0] data,
                                        input int data_id,
input int
scenario_id, input int stream_id);
                data = `VMM_RAL_DATA_WIDTH'h0;
                data[16:0] =
`DUT_TOP_PATH.stats.stats_swif.r_QSTATM_ACC_addr;
                data[30:30] =
`DUT_TOP_PATH.stats.stats_swif.r_QSTATM_ACC_read_wrt;
                data[31:31] =
`DUT_TOP_PATH.stats.stats_swif.r_QSTATM_ACC_done_gone;
                status = vmm_rw::IS_OK;
        endtask

endclass
```

Figure 20. Backdoor Read/Write tasks created by ralgen script

where `` `DUT_TOP_PATH `` is the define from the `Makefile`, the path to the block (swif.swif_core_inst.pclk_regs) is taken directly from the file in Figure 6:

```
block global_reg  =GLR (swif.swif_core_inst.pclk_regs)@‘h000000;
```

while `r_DEVID` is taken from the individual RAL file for the block GLR (see Figure 7) as seen below:

```
register DEVID (r_DEVID) @'h00
```

The result of the concatenation is shown in the write/read tasks from Figure 20 for DEVID register.

For the non-contiguous field registers the path is defined to each individual field as opposed to the register since each individual field is a register in RTL. See Figure 20 for details of implementation but everything is similar with above example the only difference is that the name of the register was changed with the name of the field and repeated for every field in the register. It is very important to notice this method of creating the backdoor by pointing to individual fields in RTL rather than registers creates a balance between the need for having an automatic script to generate the backdoor mechanism for registers, and the restrictions needed in writing the design. This is a well known concept in the industry: writing design for verification.
Once these elements were in place, the switch from frontdoor to backdoor compiled right away and it worked well for the vast majority of the registers (no memories were tested initially). However, a couple of register types showed initial problems (ie. RO, W1C type registers) because of a bug discovered in the released alpha version of RAL. The `write()` task was writing the values backdoor using `poke()` which did not obey the type of the registers and this affected the outcome for writing into RO or W1C register types. The fix was performed in time by Synopsys and after that no other significant issues were found.

4.3. Custom Backdoor for Memories

As presented in previous chapters, memories were modeled using register arrays. Unfortunately, field access was not possible using standard RAL memory models. The automatic backdoor solution was not ready in RAL at the time this paper was written. The only solution left to model backdoor access in register arrays was using predefined custom backdoor class from RAL. Every entry in the register array defining a memory was modeled separately. A `dut_reg_backdoor` class extending predefined `vmm_ral_reg_backdoor` was built and write/read tasks were created to implement backdoor access into the register arrays. An example of such as implementation of write/reads for CWOLUTMEM described in Figure 5 is given in Figure 21

```
class reg backdoors extends vmm ral reg backdoor;
        local int id;

        function new(int id);
                this.id = id;
        endfunction: new

virtual task write(output vmm_rw::statue_e status,
         input bit[`VMM_RAL_DATA_WIDTH-1:0] data);
  bit[`VMM_RAL_DATA_WIDTH-1:0] temp;

  if(id>=0 && id<1024) begin //for all 1024 memory entries
    temp={data[14:8], data[6:0]…}; // eliminate all unused
bits
    $root.dut.stats_swif.mem_data[id]=temp;
  endtask

virtual task read(output vmm_rw::statue_e status,
        output bit[`VMM_RAL_DATA_WIDTH-1:0] data);
  bit[`VMM_RAL_DATA_WIDTH-1:0] temp;

  if(id>= 0 && id<1024) begin //for all 1024 entries
    temp=$root.dut.stats_swif.mem_data[id];
    data={1’b0,temp[14:8], 1’b0, data[6:0]}; // restore all
unused bits
  end
endtask
endclass:reg_backdoors
```

Figure 21. Custom backdoor implementation for a memory modeled as register array

Mapping memory data/address in between design specs and the actual models is one of the major work items in customized memory backdoor write/read tasks. In the spec, the memory structure could be defined so that it might contain unused bits for every entry. However, these data ‘bubbles’ would be excluded in the physical memory model. Only meaningful data fields are physically mapped into RTL. Backdoor write task mapped test input data by eliminating any unused bits to memory model indexing by backdoor ID as it can be seen from Figure 21 . On the other hand, backdoor read task de-mapped RTL read back data by restoring unused bits to test output read data also reflected in Figure 21.

The class `vmm_ral_reg_backdoor`, was used for each array element. Given a specific memory, all array elements are identical in terms of field definition and data size/format only, but they are all addressed at different locations. Each memory element needs to create its own backdoor class that passes the address information (memory ID or index) to RAL in order to establish direct write/read access for a specific memory location. A predefined RAL system task, `vmm_ral_reg::set_backdoor(vmm_ral_reg_backdoor bkdr)`, must be used to activate such as backdoor write/read “registration” process. The implementation of the registration process for CWOLUTMEM memory is shown in Figure 22. It is very important to note that the IDs used in the registration process should match the IDs used in decoding the memory entries shown in Figure 21. Not matching these IDs may result in the wrong memory entry being accessed or even wrong memory being accessed in a design with more memories. Due to the large number of memories present in the current design, the verification team created a special decoding scheme for all memory elements in the design. Special attention has been paid to matching IDs for different memories in the system. Unfortunately, writing the decoding scheme is not a simple task and it is prone to human errors. Having automatic backdoor for register arrays in an actual RAL release would have saved a lot of coding and debug time especially if combined with an automatic ECC generator.

```
vmm ral reg backdoor bkdr[];

foreach(RAL.STATSM.CWOLUTMEM[id])
begin //for each of the 1024 memory entries
    bkdr[id]=new(id); //create backdoor access for each memory
entry
    RAL.STATSM.CWOLUTMEM.set_backdoor(bkdr[id]);
end
```

Figure 22. Memory entry backdoor registration process

Another important aspect of the custom backdoor access is implementing the memory protection scheme (for some memories only) in the form of a parity or ECC bits calculated over parts of the memory row or for the entire row. This task also needs special attention since it has the potential of creating false errors when accessing the physical memories. Memory protection mechanism and its implementation in above write/read tasks is not presented in this paper but it was mentioned in this paragraph just to let the reader understand the complexity of the task of creating custom backdoor for memories. A nice to have in RAL would be a generic ECC or parity generator for every memory entry. This may not be easy to do but it is worth mentioning that it would be a great feature if ever implemented.

4.4. Frontdoor vs. Backdoor Equivalence

In order to speed up simulations a frontdoor vs backdoor equivalence scheme was put in place in register and memory test cases. Basically, a mechanism to randomly choose the type of access was built into the `shuffle_door()` task from Figure 14 and **Figure 15.** This task will basically give 50% chance that frontdoor or backdoor access is chosen every time a read or a write from/to a register is performed. By running hundreds of tests per night, practically each register or memory is tested with all two of access type for each read and write. RAL coverage model is supposed to give this information as well but this is not verified yet.
Once the equivalence is proven and these tests pass all test cases accessing registers or memories will use backdoor to perform writes and reads. This has the potential to save huge amounts of time in device initialization and increase overall simulation speed for entire regression.

Figure 23 shows `shuffle_door()` task.

```
  task shuffle door(bit wr);
        bit     previous_door = door_choice;
        randomize(door_choice);
        if(door_choice)
env.set_backdoor_off();
        else env.set_backdoor_on();
        if(previous_door && wr)        #1000;
  endtask
```

Figure 23. Access randomization method

Because frontdoor writes are giving the handle to software in zero time, but it takes some time to propagate thru the design, the following read had to be delayed by #1000 to make sure the write completed in RTL. When frontdoor reads are performed this is not necessary since the read command itself takes enough time to propagate thru the logic and the previous write is completed by then.

4.5. RAL Limitations and Solutions During Backdoor Implementation

A couple of limitations of the backdoor solution appeared during implementation in the current project. The most important one was the lack of automatic backdoor for register arrays, which was replaced by the custom backdoor solution presented in detail in **Section 4.4**. This limitation is currently being addressed by Synopsys but no specific date was given for a release date. Another limitation worth mentioning was the restrictions required in order to create automatic backdoor access. There is usually resistance from the design team when some coding guidelines are required. Creating such rules because of RAL requirements created some discipline but on the other hand limited the designer's ability to write the code to match their coding style. As a solution for that problem a prototype module for register/memory access has been created and the designers had to follow the rules of writing registers inside their blocks. The trade off was to have stricter rules in the design rather than to complicate the pearl script generating RAL files and create all sorts of exceptions for all register types.

5. Conclusions and further work

Based on the experience of this project, which utilized the RAL beta version, it was concluded that this tool is an effective solution for addressing increased register complexity in current designs. The complete RAL package, when finalized with all remaining features, may prove to be a versatile tool in any verification engineer's tool box.

Although, the integration of RAL into the existing VMM environment has proven to be quite simple, the initial start up period was extensive due to project complexity and the RAL's beta status.

During this project, the following advantages using RAL were identified:

- Better documentation than what a custom solution would offer
- Streamlined and rigorous methodology
- Name-based register and field access
- Built-in internal mirroring of the registers
- Independence of tests and register specifications
- Automatic backdoor for registers.

Some features of RAL (i.e. automatic backdoor for registers, field access for memories using register arrays) significantly improved productivity and saved coding and debug time.

The automatic backdoor feature for registers saved about 2-3 weeks/person by reducing the coding time of backdoor access to all registers to almost zero, although 2-3 days were spent upfront to create the scripts. The downside of this feature is that strict guidelines for register coding inside DUT had to be followed.

The RAL field access by name for memories in current project saved about 3-4 weeks/person for frontdoor access by eliminating the need to decode accesses to all memories in the design.

The time spent to create custom backdoor for memories in RAL was roughly the same as the time that would have been spent to create it in a custom solution, namely about 8 weeks, due to the large number and complexity of the memories used in the project.

On the other hand, the verification team noticed the following limitations:

- Lack of automatic backdoor for memories
- Rigidity in specifying register array addresses
- Lack of blocking option for write/read tasks in backdoor
- Reduced list of predefined register types
- Rigid syntax in specifying external user defined constraints
- Lack of working coverage in the latest VCS release.

Some of these limitations, however, were manageable and workarounds were easily applied by the verification team with assistance from Synopsys.

What follows are suggestions of enhancements that pertain to some of the limitations identified above:

- The most important one would be the addition of automatic backdoor access for memories either as register arrays or individual memories modeled without the need of register arrays. This would save time spent in coding custom backdoor access for memories with no protection.
- The lack of blocking option for write/read tasks in backdoor may be corrected by adding a delay parameter into the write/read tasks and modify the mirror of the register only after a specified delay occurred. The user will match this delay with the right number of clocks needed for a real access into DUT.
- Another improvement for RAL would be a way to avoid the rigid syntax in specifying external user defined constraints. This has the potential to tie in an external file to the automatic generated file and to a specific register or block name which may change during the development of the project.

Apart from these limitations, a desirable feature would be the automatically generated ECC or parity protection for memories. This would have the potential to save time in building such a mechanism for each project separately.

For future projects the verification team is looking forward to testing automatic backdoor feature for memories or register arrays and RAL built in coverage.

6. Acknowledgments

The authors gratefully acknowledge and thank Chris Thompson from Synopsys for his help with the paper and quick turnaround for issues encountered with different RAL releases. Also the team thanks Janick Bergeron for his the help with RAL integration and suggestions during the development of the project. The authors also want to thank Markus Wandel from Nortel for the perl script to generate RAL files.

7. References

[1] VMM Register Abstraction Layer, RAL Version 1.7.2
[2] An Introduction to the VMM Register Abstraction Layer, Featured article, contributor: Synopsys Inc. in www.soccentral.com
[3] Brian Slater, et al, Complex Register Verification Utilizing Register Abstraction Layer (RAL), SNUG Boston 2006.

8. Authors and Contact Information

Paul Lungu, lungu@nortel.com
Bo Zhu, bozh@nortel.com