Switch the Gears of the UVM Register Package to cruise through the street named “Register Verification”.

Amit Sharma,
Varun S, Abhisek Verma
Synopsys
amits@synopsys.com
svarun@synopsys.com
abhiv@synopsys.com

Gaurav Gupta
Freescale Semiconductors
gauravG@freescale.com

I. INTRODUCTION

“Register verification” could seemingly be a simple task but in reality it tends to have an adverse effect on project schedules on account of factors such as changes to the RTL specification, design changes/optimizations, migration efforts from block to sub-system or system level and so on. Traditionally design houses had their own methodologies to reduce the time and effort spent in creating and maintaining register tests. The UVM register package advocates best practices like object oriented abstraction, automation etc. to provide a robust platform for register verification. This package easily automates the creation of object-oriented abstract model of the registers/memories inside a design and has been adopted by a lot of users in the industry. In this paper, we present some techniques which could be used to leverage the existing package efficiently for advanced register verification. We also present an auxiliary package to exploit the abstraction levels provided by the library to ease the process of software validation which is closely tied to registers and memories in a system.

Categories and Subject Descriptors

Verification, Validation, Registers, Memories, Coverage, Automation

Keywords

DPI-C, UVM, SystemVerilog.

II. REGISTER BACKDOORS

Backdoor access to registers and memories is very crucial because it is a very useful way to check the correct operation on the physical interface. It also improves the efficiency of verification as the accesses can be completed in little or no simulation time. Once the interface is proven to be working correctly, one can use the register backdoors to load the register reducing the time to configure the DUT, which can sometime be a lengthy process. Backdoor access operates by accessing the simulation constructs that implement the register or memory model through a hierarchical reference within the hierarchy of the design. The challenges with backdoor accesses are the identification and maintenance of the hierarchical backdoor paths and also the nature of the constructs that are used to implement the registers or memory models. To access a register or memory directly into the design, it is necessary to know how to get at it. The UVM register library can specify arbitrary hierarchical path components for blocks, register files, registers and memories that, when strung together, provide a unique hierarchical reference to a register or memory. For example, a register with a hierarchical path component defined as X, inside a block with a hierarchical path component defined as Y, inside a block with a hierarchical path component defined as Z has a full hierarchical path defined as Z.Y.X.” [1]

The input specification of a register contains all of the necessary information to generate the UVM register model leveraging the UVM-1.1 base class library. This includes the information w.r.t the backdoor paths of individual elements in the register model. ‘*ralgen*’ takes in the input register specification through RALF or IP-XACT and generates the final UVM register model including the backdoor classes associated with the individual registers and memory elements.

DPI-C Based Backdoor Access: The UVM register library has a set of routines defined within in that act as an interface to access the simulation constructs of the design using DPI-C. This is a very powerful feature, because in practice the user simply has to specify a set of string values while configuring the elements of the register model and the library would automatically create a DPI-C implementation to access the register construct or the memory model construct. The task gets simpler if you were to be using a register model generator in which case the user would have the backdoor construct specified in an abstract description (such as RALF, IP-XACT) of the register space of the design. Then a register model generator like ‘*ralgen*’ enables the generation of the backdoor classes leveraging the SV DPI accesses as well as through hierarchical XMRs. The generated model with the DPI based backdoor accesses is quite portable as such a model can easily be compiled into a SystemVerilog package and imported where the model is required.

```
virtual function void build();

this.PRT_LCK=reg_PORT_LCK::type_id::create(…);
this.PRT_LCK.configure(this, null, "");
this.PRT_LCK.build();

//Adding the HDL path information for register
this.PRT_LCK.add_hdl_path('{'{"lck", -1, -1}});

endfunction
```

Fig 1: DPI-C based backdoor using add_hdl_path

This mechanism however could prove costly in terms of simulation overhead because of the PLI interactions, which typically slows down simulation. Enabling read and writes access in the design through the DPI accesses can also potentially prevent the simulator for enabling some aggressive optimizations across the design hierarchy. Also if an incorrect path was specified the user wouldn’t be notified until a backdoor access is performed resulting in a runtime error. This could prove costly in terms of verification cycles because the user will have to check backdoor path for validity and with really huge designs this might not be desirable.

Verilog Cross-module Reference (XMR) Based Backdoor Access: The UVM base classes also enables the user to create extensions of the *uvm_reg_backdoor* class and have the *write()* and *read()* tasks overloaded with the implementations to access the design construct that models the register/memory.

The user would then have to register each of these class extensions with the corresponding register/memory block element of the UVM register model. If these classes are extended and registered to the corresponding hierarchy, then the register package uses this extension instead of the default DPI methods for its backdoor accesses. '*ralgen*' uses this feature provided in the base class library to enable the generation of the backdoor classes leveraging hierarchical cross module references (XMRs).

This option in theory would be relatively more efficient in terms of simulation performance as it doesn't involve any PLI interactions. Also the static compile time checks done by the HDL simulator for the existence of the HDL cross-module references helps improve the TAT when incorrect strings are specified for the backdoor paths. The user-wouldn't have to test for the validity of the HDL paths anymore as the HDL compiler would perform that check. The primary limitation of the HDL XMR based accesses is that the generated classes cannot be compiled into a reusable SystemVerilog package as SystemVerilog packages don't allow the use of cross module paths or hierarchical paths within the package scope. This can impact the portability of the block level verification environments using the generated register model.

```
// backdoor class for register PRT_LCK.
class reg_PRT_LCK_bkdr extends
uvm_reg_backdoor;

  // Task to write via backdoor
  virtual task read(uvm_reg_item rw);
     do_pre_read(rw);
     rw.value[0] = `HOST_REGMODEL_TOP_PATH.lck;
     rw.status = UVM_IS_OK;
     do_post_read(rw);
  endtask

  // Task to read via backdoor
  virtual task write(uvm_reg_item rw);
     do_pre_write(rw);
     `HOST_REGMODEL_TOP_PATH.lck = rw.value[0];
     rw.status = UVM_IS_OK;
     do_post_write(rw);
  endtask
endclass

// build() - of the uvm_reg_block
virtual function void build();
   …
begin
   reg_PRT_LCK_bkdr bkdr = new(…);

   // Setting backdoor
   this.PRT_LCK.set_backdoor(bkdr);
end
endfunction
```

Fig 2: HDL XMR based backdoor using set_backdoor

Optimized Backdoor Access: Looking at the pros and cons of both the backdoor access modes, a flow is desired which will ensure that the backdoor paths are checked for correctness at compile time, provide for improved simulation performance as well as help generate a model which can be compiled into a package. Thus, the proposed solution would generate the backdoor classes without XMRs, but with an interface instance through which the register model would access a top level interface which will compiled in the unit scope. The top level interface would be tied in automatically to all the backdoor class instances through the UVM Resource database and its associated functions.

The usage of the backdoor classes and the signatures will not change which will allow existing user code to work seamlessly with the newly generated model. There would be one interface which would be created for the entire register model. The interface would contain tasks defined within it to access the constructs that are used to model registers/memories in the RTL of the DUT.

An instance of the newly defined SystemVerilog virtual interface is then declared within the register model and initialized with the static interface instance. By doing this we now get access to the HDL design constructs via the API's defined within the interface. The backdoor classes are then re-modeled to call the interface API's instead of directly referring to the HDL design constructs.

```
interface host_regmodel_intf;
   import uvm_pkg::*;
   // Tying the interface to the virtual
   // interfaces being used in the
   // UVM reg backdoor infrastructure
   initial uvm_resource_db#
   (virtual host_regmodel_intf)::
           set("*", "uvm_reg_bkdr_if",
                 interface::self());

   task reg_PRT_LCK_bkdr_read(…);
      rw.value[0] = `HOST_TOP_PATH.lck;
   endtask

   task task reg_PRT_LCK_bkdr_write(…);
      `HOST_TOP_PATH.lck = rw.value[0];
   endtask
endinterface
```

Fig 3: Interface with read/write tasks for backdoor.

```
//modified backdoor register PRT_LCK class.
class reg_PRT_LCK_bkdr extends
                              uvm_reg_backdoor;

   virtual host_regmodel_intf __reg_vif;

  function new(string name);
    super.new(name);
    // initializing the virtual interface with
    // the real interface
    uvm_resource_db#
    (virtual host_regmodel_intf)::
       read_by_name(…, "uvm_reg_bkdr_if",
                                    __reg_vif);
  endfunction

  virtual task read(uvm_reg_item rw);
     do_pre_read(rw);

// performing a read access to register
__reg_vif.host_regmodel_PRT_LCK_bkdr_read(rw);
     rw.status = UVM_IS_OK;
     do_post_read(rw);
  endtask

  virtual task write(uvm_reg_item rw);
     do_pre_write(rw);

// performing a write access to register
__reg_vif.host_regmodel_PRT_LCK_bkdr_write(rw)
;
     rw.status = UVM_IS_OK;
     do_post_write(rw);
  endtask
```

Fig 4: Modified backdoor class using the interface tasks.

III. FAST LANE SEQUENCES

Coverage Convergence using Fast Lane Sequence: Coverage is another added benefit the UVM register package brings to the table. In this section we discuss the coverage options defined in the methodology: bit-based coverage, address map coverage & field coverage. We then present a case study that shows how bit and address map coverage's are fully addressed/covered by the pre-packaged sequences that are readily available in the library. We specifically discuss field coverage as this covers different device configurations. As we move from block to cluster to subsystem verification, the accesses which we would want to track would vary and would be defined by different flavors of coverage models for the registers in the design. The different permutations of coverage bins for the multitude of registers at each step make it a challenging scenario for coverage convergence. We also provide a case study where the UVM REG model generator is enabled to automatically generate sequences. We show how these sequences can be easily applied at different stages of the verification cycle by dynamic extraction of coverage holes to meet coverage goals minimizing user efforts in writing test sequences.

UVM provides guidelines to implement appropriate functional coverage models to verify and track that the different combinations of values being driven to the DUT registers. As we move from block to cluster to subsystem verification, the accesses which we would want to track would vary and would be defined by different flavors of coverage models for the registers in the design. The different permutations of coverage bins for the multitude of registers at each step make it a challenging scenario for coverage convergence. We demonstrate how one can enable the UVM REG model generators extract the required information from the register specifications to automatically generate '***coverage converging***' sequences. These sequences when applied at different stages in the verification cycles can help users dynamically extract the coverage holes and enable the required accesses to meet the verification goals.

UVM REG defines the following three functional coverage models for register and memories but it doesn't provide any rules for implementing them:

Register bit coverage: This is used to confirm that every specified bit in a register abstraction model has been thoroughly exercised and is implemented as specified. This functional model can be quite large and is, therefore, best used at the block level. At the SOC level, we need to ensure that different addresses ranges are hit along with a combination of specific field values across different registers. This is to ensure all the IPs have been accessed and has been configured with appropriate values for the SOC simulations. This brings in the other two coverage models

Field coverage: This model is designed to confirm that every configuration of a design has been verified. It is best used at the top-level. Additionally, the user can define specific bins explicitly as shown in the code snippet below:

```
field f2 {
  bits 8;
  enum {AA, BB,CC=15};
  coverpoint {
    bins AAA = {0, 12};
    bins BBB[] = {1, 2, AA, CC};
    bins CCC[3] = {14,15,[BB:10]};
    bins DDD = default;
  }
}
```

Fig 5: Embedding field coverage in register specification.

The generator creates the coverage bins automatically based on the field values and constraint model or as specified through the input specification.

Address Map coverage: This model is designed to confirm that the address map of a design has been thoroughly exercised. This is valid for registers. A register coverage point contains only one bin named "accessed". The bin is covered whenever the register is accessed using either a read or a write operation.

The pre-defined UVM REG sequences would help to target a significant percentage of the coverage bins as specified in the coverage models. Some of the important ones in the context of coverage are:

Register bit-bash sequence: This test verifies the implementation of a single register by attempting to write 1's and 0's to every bit in it via every address map.

Register access sequence: Verifies the accessibility of the register by writing via the default map and then reading via the backdoor.

Memory walking-ones sequence: Performs the walking-ones algorithm on the memory for each map in which the memory is defined.

Shared register access test sequence: Verifies the accessibility of register/memories that are shared between multiple physical interfaces.

These ensure that coverage bins specified in the Register Bit Coverage and Address Mapped coverage models are targeted.

The next step is to create the required stimulus for the field coverage. The number of bins can be significant here as we are looking at multiple permutations of values of fields across various registers. The built-in sequences do not necessarily help in this front beyond an initial threshold. Analyzing all the myriad holes and creating specific sequences to target all such bins would require significant amount of effort, disk space and additional resources. Sequences might have to be rewritten for changes in the register model. This led to the creation of a utility for generating the coverage converging sequences.

Since the generator has all the information on the hierarchical functional models that were created. It was further enhanced to generate sequences which will have a collection of register accesses required to achieve closure on the functional coverage. The SystemVerilog language provides capabilities to query the functional coverage results dynamically. Thus the sequence generated would query individual coverpoints to determine whether a register write/read operation is required to achieve full coverage, following which it would perform the same as shown below:

```
task body();
model.ID.cg_vals.R_ID_value.get_inst_coverage(c
overed, total);
if(total!=covered) begin
   model.ID.R_ID.read(status, data,
                      .parent(this));
   model.ID.sample_values();
end
endtask
```

Fig 6: Embedding field coverage in register specification.

The other important aspect that should be kept in mind while generating the field coverage models is to map the respective bins to individual coverpoints. This ensures that more attributes of the model can be queried dynamically through the SV constructs as shown below:

```
covergroup cg_vals ();
  option.per_instance = 1;

  R_ID_value: coverpoint R_ID.value
  {
    bins value = { 8'h03 };
  }
  CHIP_ID_value: coverpoint CHIP_ID.value
  {
    bins value = { 8'h5A };
  }
  PROD_ID_value: coverpoint PRODUCT_ID.value
  {
    bins value = { 10'h176 };
  }
endgroup : cg_vals
```

Fig 7: Generated covergroup for field coverage.

IV. REUSING TESTS

The register package of UVM coupled with vendor tools provide ways to automate the generation of register models and register accesses from input specifications. The need of the hour is to ensure that the same sequences can be reused in post-silicon validation and in simulation. A convenient way for verification engineers to develop firmware code which can be debugged on a RTL simulation of the design would significantly help in reducing the validation cycles.

Typically firmware runs a set of pre-defined sequences of writes/reads to the registers on a device performing functions for boot-up, servicing interrupts etc. These functions are generally coded in C/C++, and these would need to access the registers in the DUT. Using the UVM Register Model with the appropriate C++ DPI interface, these functions can be generated so that the firmware can now perform the register access through the SystemVerilog register model. Thus, this allows firmware and application-level code to be developed and debugged on a simulation platform and the same functions can later be used as part of the device drivers to perform the same tasks on the hardware.

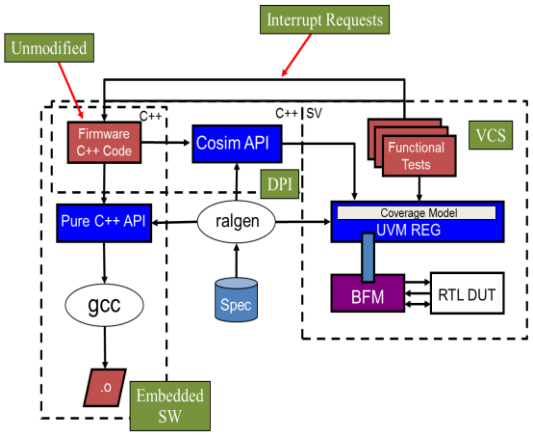

Fig 8 : Block diagram of the C++/SV test environment

To facilitate this, C++ library would be required which would allow users to define register accesses as C++ functions. Using the functions defined within this library, the user can create C++ sub-routines that perform control functions on a device-under-test.

The library has been defined in a way that it can be used to:

I. Be compiled and executed as a standalone C++ code on the target processor or

II. To be interfaced to the SystemVerilog register model using DPI-C to be simulated on a HDL simulator.

When executing the C++ code within a simulation, it is necessary for the C++ code to be called by the simulation to be executed and hence the application software's main() routine must be replaced by one or more entry points known to the simulation. The C++ side reference is then used by the C++ API to access required fields, registers or memories. The C++ code is executed natively on the same workstation that is running the SystemVerilog simulation, eliminating the need for an instruction set simulator or a RTL model of the processor.

The C++ function serves as the service entry point which is called from the SV simulation which provides the reference of the object based register model. When the C++ code executes, the simulation freezes until the control is shifted to the SV side. That DPI-C entry point creates an instance of the register model based on the context specified by the UVM simulation.

```
extern "C" int
usb_dev_isr_entry(int context)
{
   usbdev_t usb(context);
   return usb_dev_isr(usb);
}
```

Fig 9: C++ test entry taking context as an input

The C++ code can then be called from a UVM simulation by calling its corresponding entry point and specifying the context of the register model.

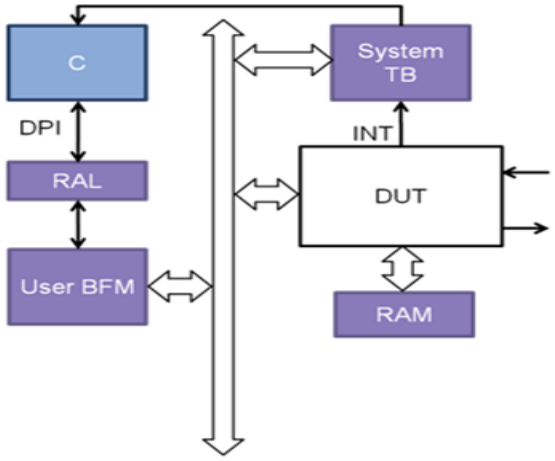

Fig 10: Environment for an interrupt-driven C++ interaction

When executing within a simulation of the design, all C++ code executes atomically. It is unlike the real application code running as object code on a real processor, where the execution of the code happens concurrently with other processing in the neighboring hardware.

When the C++ code executes in simulation, only that code performs any form of processing and the rest of the design is frozen. The only way for the design simulation to proceed, is for the C++ code to return or for the C++ code to perform a read or write operation through the register model. In the latter case, once the read or write operation completes and the control is returned back to the C++ code, the simulation is again frozen. The entire execution timeline in the C++ code thus occurs in zero-time in the simulation timeline.

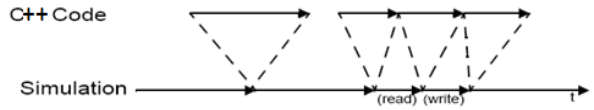

Fig 11: Execution timeline for a C++ register test

If a polling strategy is used, the simulation will have the opportunity to advance only during the execution of the repeated polling read cycles. It would likely require many hundreds of such read cycles for the design to reach a state that is relevant and significant for the application software. With a physical device, that is not an issue as this can happen in less than a microsecond. However, in simulation, this would require a lot of processing for simulating essentially useless read cycles and exchanging data between the C++ world and the simulation world.

If an interrupt-driven strategy is used, the simulation will proceed until something of interest to the application software has happened before transferring control to the C++ code and only the necessary read and write operations would need to be performed. Therefore, it is important that you use a service-based approach as much as possible.It is also very important that the execution of the C++ code not be blocked by an external event—such as waiting for user input or a file to be unlocked—as it will prevent the simulation from moving forward while it is blocked. If the application software requires such synchronization, it should similarly use an asynchronous interrupt-driven approach.

To enable this infrastructure, there are two requirements. The first, as mentioned earlier, would be the C++ library which would allow users to define register accesses as C++ functions. The next requirement would be to have representation of the register model on the C++ side. This has to be facilitated by a register model generator. It would generate a hierarchical model of the registers in a design that are accessible via a specific address map.

Device driver code should be written in functions accepting a reference to the register model corresponding to the device. That register model is then used to identify the registers to be accessed.

```
reqs=snps_reg::regRead(usbdev.status());

snps_reg::regWrite(usbdev.intMask(),0xFFFF);
```

Fig 12: Device driver accepting reference of the reg model

The code illustration below shows the API's which invoke accesses on registers defined in an example design module. *regWrite()* and *regRead()* are library functions which takes a reference of the register model of the device registers to access them in one of either modes mentioned above.

```
void slave_driver::dev_drv(slave_t dev)
{
    uint32 mode_status;
    regWrite(dev.SESSION.SRC(), 0x0000FA);
    regWrite(dev.SESSION.DST(), 0x000E90);
    regRead(dev.MODE_STATUS);
    switch ( mode_status )
    {
       case 0x0001:
         regWrite(dev.IDX(), 0x5aa5);
         break;
       case 0x0080:
         regWrite(dev.IDX(), 0xa55a);
         break;
       default:
         regWrite(dev.IDX(), 0x0000);
    }
};
```

Fig 13: C++ device driver code.

```
static slave_t Sys("Sys", 0);

int
main(int argc, char* argv[])
{
    return slave_driver::dev_drv(Sys);
}
```

Fig 14: Device driver scheduled for execution as software.

In the illustration above we have the C++ function *dev_drv* being scheduled for execution within the *main()*. And the same function is being called as a DPI-C function within the SystemVerilog UVM test as shown below. Please note that the function takes the object reference of the register model as an input from the SystemVerilog side.

```
import "DPI-C" context task dev_drv(int ctxt);

class cpp_test extends uvm_test;
   int context_val;
   `uvm_component_utils(cpp_test)

   virtual function void connect_phase(…);
      super.connect_phase(phase);
      context_val =
         snps_reg::create_context(env.model);
   endfunction: connect_phase

   virtual task run_phase(uvm_phase phase);
      super.run_phase(phase);
      phase.raise_objection(this);

      // C++ device driver called via DPI-C
      dev_drv(context_val);

      phase.drop_objection(this);
   endtask: run_phase

endclass: cpp test
```

Fig 15: The C++ Device driver being used in simulation test

As mentioned earlier, there are two versions of the UVM register C++ API that can be used. One is designed to interface to the UVM register model running in the SystemVerilog simulator using the Direct Programming Interface. The other is pure stand-alone C++ code and is designed to be compiled on the target processor in the final application. The version of the C++ API that will be used is determined at compile time by including the appropriate header file. The illustrations mentioned above show how the same piece of code can be used once over without having to change anything while going from simulation to validation without going into the details of how these API's calls translate either to a simulation level register accesses or a software instruction.

V. MODELING ISR'S IN UVM

In a verification environment, different components may be trying to access the DUT registers and memories. For example, the BFM might be programming some registers while the bus monitor might be sampling the values of these registers. In specific cases, there may be an interrupt monitor which triggers an Interrupt Service Routine (ISR) whenever it sees an Interrupt pin toggling in the interface. The ISR might end up having to read the Interrupt registers and end up clearing the Interrupt bit/s through a front door access.

The base register package assumes register accesses to be atomic in nature and so any access will be completed (successfully/as an error) before scheduling the next. Things have been set this way to ensure register accesses are not corrupt. However, this atomic

nature of a register access would impede the modeling a system interrupt which is a common occurrence in designs. And hence for the occurrence of an interrupt we have to have the pass the control over from an ongoing register access thread to a thread that services the interrupt.

To ensure that different components in a verification environment can access the DUT registers at any given point in time, the register model instantiated in the environment can be passed to different UVM components through the UVM Resource Database. These different components whose methods are executing in separate parallel threads can now access the same set of registers in the DUT through the RAL model. A question many folks ask is: when there are multiple parallel register accesses, how do they get scheduled through the register layer?

A Register read/write from different threads is comparable to an atomic sequence being started on the sequencer associated with the different threads. Hence it gets scheduled in the order of pipelining of the threads. A write/read would basically consist of the following atomic operations:

- A generic register (uvm_reg_item) transaction with its fields (addr, data, kind etc.,) being populated and posted onto the reg2bus() function of the register adapter.
- The transaction being translated in the reg2bus() function and posted as a UVM sequence onto the associated sequencer for the specified 'map'.
- The transaction being retrieved in the User BFM main thread and then subsequently driven to the DUT interface.

Thus 'posting' of register accesses whenever a Read/Write/Mirror/Update is invoked is in the same order they are issued. Subsequently, the Register Adapter translates the generic UVM REG transaction to a User Sequencer comprehensible sequence, and doesn't change the order.

Now, how do we handle a scenario when specific register accesses like those coming from an ISR need to be given a higher priority than accesses coming from other threads in a verification environment? The UVM base class library and the UVM register package provide a way to achieve this by controlling the sequencer to prioritize an interrupt subroutine sequence above other normal sequences. Let us see how this can be done with UVM.

To start with we will need to model base register sequence to devise an arbitration scheme and have all other register sequences extend from this base sequence. The arbitration scheme will be defined using function *is_relevant()* & task *wait_for_relevant()* which is part of the UVM base sequence.

```
class host_base_sequence extends
 uvm_reg_sequence #(uvm_sequence #(host_data));

    function bit is_relevant();
       return (p_sequencer.state == NORMAL);
    endfunction

    task wait_for_relvant();
       p_sequencer.state = NORMAL;
    endtask
endclass
```

Fig 16: Base sequence to checking the sequence priority.

All sequences should be extended from the sequence defined above in figure 15. The ISR sequence will have *is_relevant()* & *wait_for_relevant()* methods defined to adjust the priority to the sequence with the reception of an interrupt event.

The sequence defined in Fig 17 waits for the assertion of the interrupt to before flagging relevance. With ISR sequence becoming relevant all normal sequences will be pushed down in the sequence queue of the sequencer. The sequencer would schedule & execute the ISR sequence which would restore the state of the sequencer to NORMAL when it is done servicing the interrupt. Also note how the sequence grabs the sequencer on which it executes to create an exclusive access preventing other threads from intervening with the interrupt sub-routine process.

```
class host_isr_sequence extends
host_base_sequence #(uvm_sequence #(host_data));

    function bit is_relevant();
       return (p_sequencer.state == INTERRUPT);
    endfunction

    task wait_for_relvant();
       // Waits for the interrupt assertion
       @(dut_top.inta);
       p_sequencer.state = INTERRUPT;
    endtask

    virtual task body();
       forever begin
          grab(p_sequencer);
          // Task that contains the routine set
          // register accesses relevant to the
          // interrupt
          isr();
          ungrab(p_sequencer);
          p_sequencer.state = NORMAL;
       end
    endtask : body

endclass
```

Fig 17: ISR sequence waiting for the interrupt.

Finally, the ISR sequence will have to be run concurrently with other regular sequences on the sequencer that connects to the host driver. The figure below shows how this can be done.

```
class top_sequencer extends uvm_sequencer;
  …

  host_isr_sequence interrupt_handler;

  virtual task run_phase(uvm_phase phase);
    interrupt_handler =
host_isr_sequence::type_id::create("interrupt_se
q");

    // Forking off the interrupt thread
    fork
       interrupt_seq.start(this);
    join_none

    super.run();
  endtask : run

endclass
```

Fig 18: Running the ISR concurrently with other sequences

VI. RESULTS & CONCLUSIONS

We have discussed some key features of the UVM register package across the length of this paper and in some sections we have stressed on the impact the package can have on the simulation. In particular, we have discussed the backdoors and seen how one option is better than the other. We also presented a way to pick the best out of the two options.

The simulation profiling showed that the HDL XMR based backdoors provides a simulation speedup by a factor of nearly 30%. And also the hybrid backdoor performs consistently with the HDL XMR based backdoor. This difference accumulated over multiple test sequences could be very costly.

Coverage is another topic that we broached upon and discussed strategies to converge quickly on to reach the coverage goals. We found that the auto-generated coverage converging sequence to be useful in filling up a lot of coverage holes in a very short span of time efficiently. Such an approach ensured that no additional code was written by the user. This could apply this at different stages in the verification cycle depending on the requirement. If applied earlier, it would generate more accesses while if it is applied later in the cycle, it would generate a lesser number based on the reduced number of holes in the register coverage model.

```
Testbench Group List

Total Groups Coverage Summary
SCORE  INST SCORE WEIGHT
 49.54  50.01     1


Total groups in report: 9
----------------------------------
SCORE  INSTANCES WEIGHT GOAL   NAME
  0.00   0.00    1      100    STATUS::cg_vals
  0.00   0.00    1      100    MASK::cg_vals
  0.00   0.00    1      100    CHIP_ID::cg_vals
  0.00   0.00    1      100    COUNTERS::cg_vals
 66.67  66.67    1      100    CHIP_ID::cg_bits
 79.17  86.96    1      100    STATUS::cg_bits
100.00 100.00    1      100    slave::cg_addr
100.00 100.00    1      100    MASK::cg_bits
100.00 100.00    1      100    COUNTERS::cg_bits
```

Fig 19: Coverage score from the pre-defined sequences.

The sequences provided as part of the UVM base cumulatively generated coverage of **49.4%** as shown in the coverage report above. When the auto-generated sequence was run following the pre-defined sequences we saw a steep curve in the coverage rise as summarized by the report shown below:

```
Testbench Group List

Total Groups Coverage Summary
SCORE  INST SCORE WEIGHT
 92.39  99.88     1


Total groups in report: 9
----------------------------------
SCORE  INSTANCES WEIGHT GOAL NAME
 66.67  66.67    1      100  CHIP_ID::cg_bits
 79.17  86.96    1      100  STATUS::cg_bits
 85.71  85.71    1      100  STATUS::cg_vals
100.00 100.00    1      100  slave::cg_addr
100.00 100.00    1      100  MASK::cg_vals
100.00 100.00    1      100  CHIP_ID::cg_vals
100.00 100.00    1      100  MASK::cg_bits
100.00 100.00    1      100  COUNTERS::cg_vals
100.00 100.00    1      100  COUNTERS::cg_bits
```

Fig 20: Coverage score after using the Fastlane sequences.

The other common problem reported by users is with dynamic object allocation as encountered with any object based models. Devices with a sizable register space would limit the usage as simulators cannot handle beyond a certain threshold of objects. So it would be good to explore ways to limit object allocation on demand and not have them allocated by default.

VII. REFERENCES

[1] UVM User Guide
[2] Verification Martial Arts: A Verification Methodology Blog: http://www.vmmcentral.org/vmartialarts/
[3] UVM Reference Guide
[4] UVM RAL Primer, Janick Bergeron
[6] UVM Register Abstraction Layer Generator User Guide
[7] IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool flows