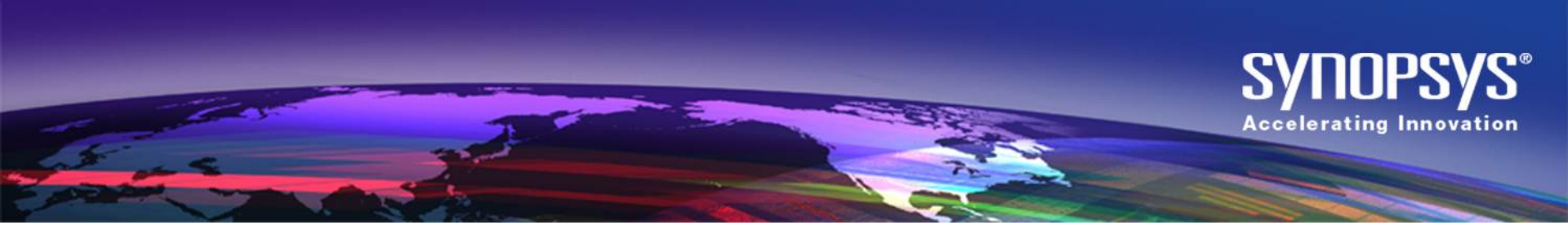


Switch the Gears of the UVM Register Package to cruise through the street named “Register Verification”.

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

Gaurav Gupta
Freescale Semiconductors
gauravG@freescale.com

Advanced Register Verification

Optimized Register Backdoor Access

Modeling iSR

Register Coverage Convergence Sequences

Reusing Tests Across Verification and Validation

Optimized Register Backdoor Access

High simulation performance as no PLI interactions

Interface with read/write tasks for backdoor

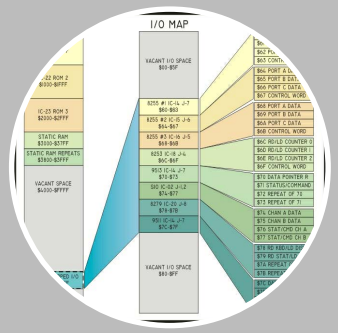
```
interface host_regmodel_intf;
   import uvm_pkg::*;
   // Tying the interface to the virtual
   // interfaces being used in the
   // UVM reg backdoor infrastructure
   initial uvm_resource_db#
   (virtual host_regmodel_intf)::
           set("*", "uvm_reg_bkdr_if",
                 interface::self());

   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
```

Modified backdoor class using the interface tasks

```
//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

endclass
```

No HDL XMR ensures SystemVerilog package-able

UVM REG Coverage



Address map

- Have all address location in the map been accessed?
- Pre-defined sequences covers this.

Bit -level

- A regressive coverage that covers values for each bit position
- Pre-defined UVM register sequence will cover this too.

Field value

- Configuration value coverage for register fields.
- User written configuration sequences. A bottle neck!

Why need fast lane sequence?

Fast lane sequence (config aware)

100% field coverage targeted

Add-on to pre-defined plus user sequences

Custom user sequences

Pre defined UVM RAL sequences

100% address map coverage targeted

100% register bit coverage targeted

Aiding the fast lane sequence

Generate the field coverage models by mapping the respective bins to individual coverpoints. This ensures that more attributes of the model can be queried dynamically through the SV constructs

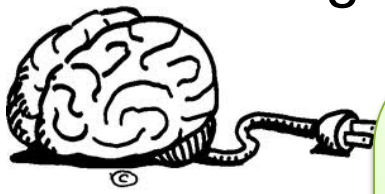
```
covergroup cg_vals ();
  option.per_instance = 1;

  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
```

Hierarchical model of the coverage architecture enables traversal of the entire coverage model

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

Let your register spec do the talking

- Embed configuration information into your register spec

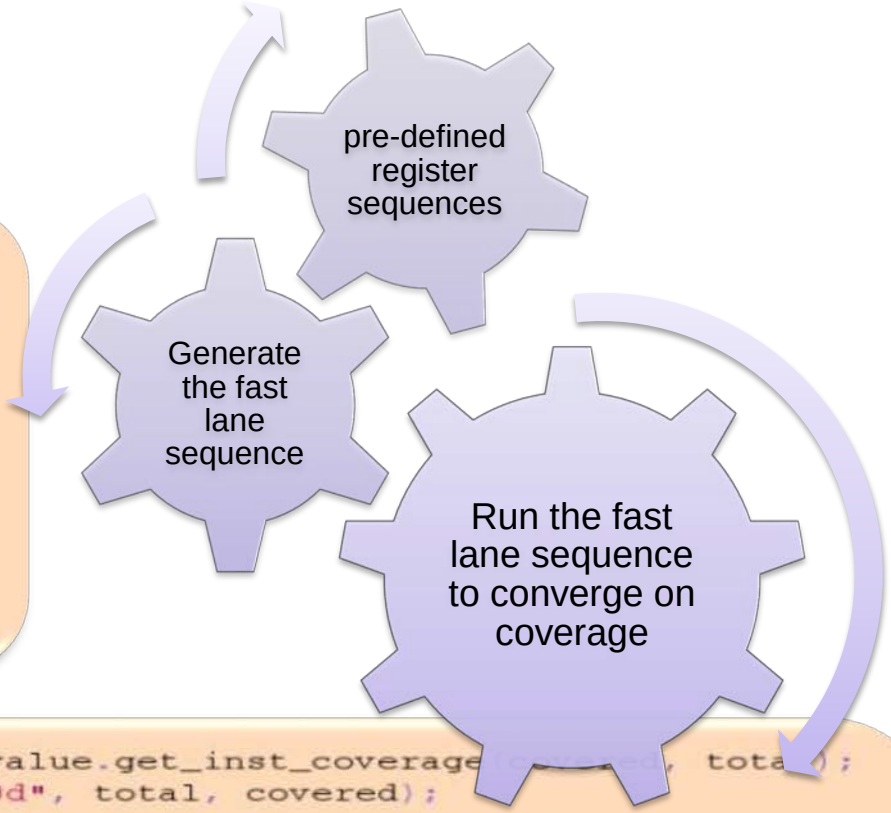


```
field f2 {
    bits 2;
    enum {bus_mode, switch_mode};
    coverpoint {
        bins bus_mode = {0};
        bins switch_mode = {1};
        bins reserved = {2, 3};
    }
}
```

- Enable the model generator to generate configuration sequences to cover all the cases.

Switch gears

```
covergroup cg_vals ();
        option.per_instance = 1;
        REVISION_ID_value : coverpoint REVISION_ID.value {
                bins value = { 8'h03 };
        }
        CHIP_ID_value : coverpoint CHIP_ID.value {
                bins value = { 8'h05 };
        }
        PRODUCT_ID_value : coverpoint PRODUCT_ID.value {
                bins value = { 10'h176 };
        }
endgroup : cg_vals
```




```
model.CHIP_ID.cg_vals.REVISION_ID_value.get_inst_coverage(covered, total);
$display("total = %0d, covered = %0d", total, covered);
if(total!=covered) begin
  model.CHIP_ID.REVISION_ID.write(status, data, .parent(this));
  model.CHIP_ID.sample_values();
end

model.CHIP_ID.cg_vals.CHIP_ID_value.get_inst_coverage(covered, total);
$display("total = %0d, covered = %0d", total, covered);
if(total!=covered) begin
  model.CHIP_ID.CHIP_ID.read(status, data, .parent(this));
  model.CHIP_ID.sample_values();
end

model.CHIP_ID.cg_vals.PRODUCT_ID_value.get_inst_coverage(covered, total);
$display("total = %0d, covered = %0d", total, covered);
if(total!=covered) begin
  model.CHIP_ID.write(status, data, .parent(this));
  model.CHIP_ID.sample_values();
end
```

On the Fast Lane

Total Groups Coverage Summary

SCORE	INST SCORE	WEIGHT
92.39	99.88	1

Total groups in report: 9

Coverage After using Fast Lane sequence

SCORE	INSTANCES	WEIGHT	GOAL	NAME
66.67	66.67	1	100	test::ral_reg_slave_CHIP_ID::cg_bits
79.17	86.96	1	100	test::ral_reg_slave_STATUS::cg_bits
85.71	85.71	1	100	test::ral_reg_slave_STATUS::cg_vals
100.00	100.00	1	100	test::ral_block_slave::cg_addr
100.00	100.00	1	100	test::ral_reg_slave_MASK::cg_vals
100.00	100.00	1	100	test::ral_reg_slave_CHIP_ID::cg_vals
100.00	100.00	1	100	test::ral_reg_slave_MASK::cg_bits
100.00	100.00	1	100	test::ral_reg_slave_COUNTERS::cg_vals
100.00	100.00	1	100	test::ral_reg_slave_COUNTERS::cg_bits

UVM library provided base sequences were run and coverage was cumulatively collected over these runs.

This was followed up with the auto-generated sequence which checked for the uncovered points and generated accesses to cover the remaining coverage holes.

Total Groups Coverage Summary

SCORE	INST SCORE	WEIGHT
49.54	50.01	1

Total groups in report: 9

Coverage before using Fast Lane sequence

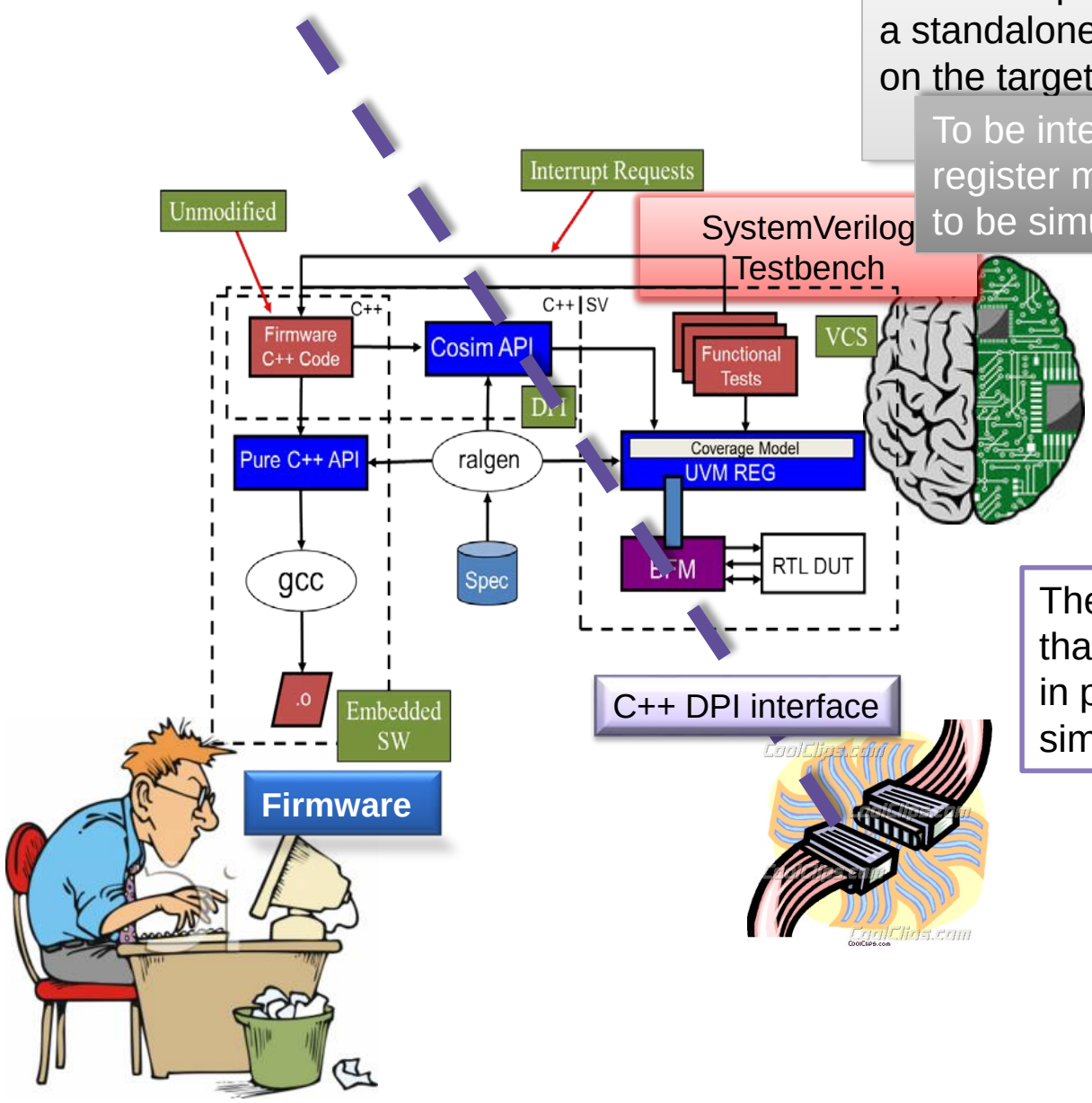
SCORE	INSTANCES	WEIGHT	GOAL	NAME
0.00	0.00	1	100	test::ral_reg_slave_STATUS::cg_vals
0.00	0.00	1	100	test::ral_reg_slave_MASK::cg_vals
0.00	0.00	1	100	test::ral_reg_slave_CHIP_ID::cg_vals
0.00	0.00	1	100	test::ral_reg_slave_COUNTERS::cg_vals
66.67	66.67	1	100	test::ral_reg_slave_CHIP_ID::cg_bits
79.17	86.96	1	100	test::ral_reg_slave_STATUS::cg_bits
100.00	100.00	1	100	test::ral_block_slave::cg_addr
100.00	100.00	1	100	test::ral_reg_slave_MASK::cg_bits
100.00	100.00	1	100	test::ral_reg_slave_COUNTERS::cg_bits

REUSING TESTS

To be compiled and executed as a standalone C++ code on the target processor

C++ Register library

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



The need of the hour is to ensure that the sequences can be reused in post-silicon validation from RTL simulation.

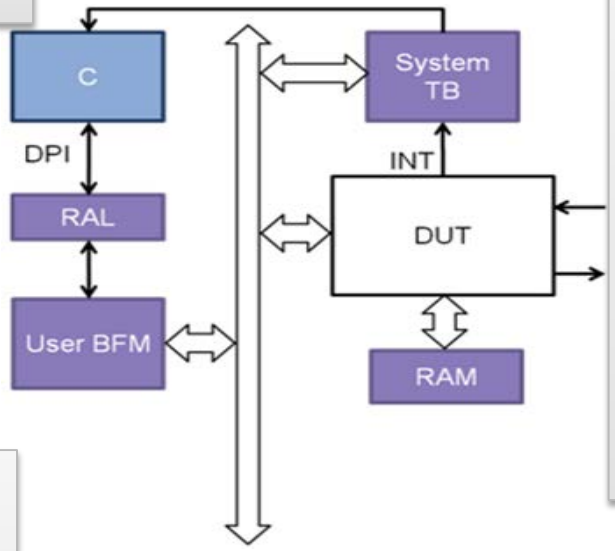
REUSING TESTS

C++ test entry taking context as an input

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

Device driver accepting reference of the reg model

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

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




```
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
```

The C++ Device driver being used in simulation test

```
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);

   }
};
```

C++ device driver code

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

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

Device driver scheduled for execution as software

Environment for an interrupt-driven C++ interaction

MODELING ISR

```
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
```

Base sequence to checking the sequence priority

```
 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 assertio
       @(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
```

ISR sequence waiting for the interrupt

Running the ISR concurrently with other sequences

```
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
```