A dynamic approach towards Register coverage generation and collection to reduce compilation overhead of traditional UVM register layers

Subham Banerjee
DVCon’2020, San Jose
Content

> Motivation
  >> Why Register Coverage?
  >> Problem Statement
  >> Proposed Solution

> Basic Register Coverage Flow
  >> Typical UVM based Register Model
  >> Implementation
  >> Sampling Techniques
  >> Sampling Timeline
  >> User Control
  >> Reporting

> Register Cross Coverage Extension
  >> Flow Implementation
  >> Generic Cross Coverage Class
  >> Example Spreadsheet Format
  >> Reporting

> Results & Performance Analysis
Motivation

> Why Register Coverage?

- Register Coverage is an integral part of any verification sign-off.
- It helps to detect coverage holes, which otherwise go undetected.
  - Let’s say there’s a register field ‘ADDR’, which is 3 bit wide, and not all values are realized in simulations.
    - Two distinct writes of 3'b000 & 3'b111 can cover all the toggle coverage.
    - Based on the RTL implementation, code coverage can also be 100%.
  - Analog registers used inside Behavioral Model (BV) & Real Number Model (RNM), are never analyzed as part code coverage.
  - Coverage prior to reset and powerdown may be collected, as part of code coverage.
- Register Coverage can unearth all these holes, and many more.

> Problem Statement

- Register Coverage comes with an inherent problem of compilation overhead.
- Any typical UVM-RAL based automated-flow, attempts to generate covergroups/coverpoints per register/fields.
- The overhead grows with higher number of registers.
- Case Study:
  - The Gigabit Transceivers (SERDES) subsystems that we are working on, we have around 5000 odd register fields.
  - In conventional approach, this was resulted in a massive line of code (around 25000), and classes.
  - This increased the compilation & elaboration time by ~17 minutes comparing to the NO_COVERAGE compilation.

> Proposed Solution

- The proposed approach addresses this issue, by devising a fully reusable methodology which helps dynamic creation of all the covergroups/coverpoints.
- All covergroups are created during run or simulation time, as oppose to compilation time.
- Can be seamlessly extended to generate register cross coverage with minimal user intervention.
Basic Register Coverage Flow

> A typical UVM-RAL based register model realization

>> Coverage database will be created based on each field of the registers
Basic Register Coverage Flow

> Implementation

![Diagram](Image)

- **UVM-Build**
  - **LoadRegDB()**
  - **BuildCov()**
  - **RegField\[valid_val<0-N>\]**

- **UVM-Main**
  - **CovSample()**
  - **RegField[<>].sample()**

*Reusing one coverage group across all fields and values*
Basic Register Coverage Flow

> Generic Covergroup Creation
  > One covergroup for
    - All Registers
    - All Fields within each Registers
    - All values of the field

> Covergroup Instantiation
  > Creation of covergroup wrapper class
  > Instantiate wrapper-class in uvm_build_phase
  > Better runtime control through uvm_config_db
Basic Register Coverage Flow

Sampling Techniques

- **Automatic Sample**
  - field.sample() gets called whenever there’s a corresponding register write
  - Coverage collection happens with minimal or no user effort/intervention
  - [Very prone to false-coverage]

- **Manual Sample()**
  - At End of UVM-Main-Phase, sample() gets called on whole register block
  - User can call sample at any point-in-time through following APIs
    1) RegBlock.sample()
    2) RegBlock.Reg.Sample()
    3) RegBlock.Reg.Field.sample()
Basic Register Coverage Flow

> Sampling Timelines

UVM Run-Phase

Register Randomization

Register Configuration & DUT POR

Reset Sequences & User Test-Sequence

- Change Attribute
- Reset PLL/ILO/Channel
- Re-Run Test-Sequence

End-Of-Test Checks

Manual Sample @ end-of-each iteration

Automatic Sample @ end-of-Main-Phase

UVM Main-Phase
Basic Register Coverage Flow

Use Control for flow integration

<table>
<thead>
<tr>
<th>testparam/API Name</th>
<th>Config/API Usage</th>
</tr>
</thead>
</table>
| xvm_reg_coverage_on        | • Turn on/off the register coverage model creation and sampling in dynamic manner.  
                              | • For Register read/write tests, coverage model creation will be bypassed, to avoid any false coverage collection. |
| xvm_reg_coverage_auto_sample| • This will turn-on automatic coverage collection/sampling, whenever any registers are updated/written.  
                              | • By default, it’s tuned off, and the sampling is manually controlled from the test.                                                      |
| xvm_reg_coverage_all_undef  | • This will control the cover group creation of fields with undefined valid encoding in IPXACT (e.g. A_SDM_DATA [15:0]).  
                              | • If turned on, then create one covergroups for all possible field values.  
                              | • If tuned off, then create two basic covergroups, one for ‘zero’ value and one for ‘non-zero’ value.                                      |
Basic Register Coverage Flow

- Reporting
## Register Cross Coverage Extension

> Flow Implementation

<table>
<thead>
<tr>
<th>Basic Register Coverage</th>
<th>Register Cross Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDB.txt</td>
<td>cross_cov_db.txt</td>
</tr>
<tr>
<td>uvm_reg</td>
<td>xvm_cross_cov</td>
</tr>
<tr>
<td>LoadRegDB()</td>
<td>BuildCross_Cov()</td>
</tr>
<tr>
<td>BuildCov()</td>
<td>CrossField[val&lt;0-N&gt;]</td>
</tr>
<tr>
<td>RegField[val&lt;0-N&gt;]</td>
<td>CrossCovSample()</td>
</tr>
</tbody>
</table>

*Reusing one covergoup across all fields and values*
Register Cross Coverage Extension

> Generic Cross Coverage Class

- One covergroup for any cross
- Each cross has multiple contributing fields
- Any of these fields changes, cross is sampled
- ‘cross_details’ field in the covergroup will have all the individual field info

```cpp
class xvm_field_cross;

// Add coverage
covergroup cross_cov_cg(string cross_name, string cross_details) with function sample(bit match);
option.per_instance = 1;
option.name = cross name;
option.comment = cross details;
match_cp : coverpoint match {
  bins match_c = {1};
}
endgroup : cross_cov_cg

extern function new (string a_cross_name, string a_cross_details);
extern virtual function void eval_and_sample_cross();
extern virtual function void sample_cg();
endclass : xvm_field_cross
```
Register Cross Coverage Extension

> Cross Coverage Spreadsheet Format

<table>
<thead>
<tr>
<th>Cross Name</th>
<th>RX PCS CFG0 RX DATA_WIDTH</th>
<th>RX PCS CFG0 RX INT_DATA_WIDTH</th>
<th>RX PCS CFG0 RX FABRIC DATA_SEL</th>
<th>RX PCS CFG0 RX FIFO DATA_SEL</th>
<th>RX PCS CFG0 RX EPG</th>
<th>RX PCS CFG0 RX 81B</th>
<th>RX PCS CFG0 RX 81B EN</th>
<th>RX CTRL CFG0 USB_MODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CROSS_BOXerox1</td>
<td>XXX</td>
<td>XXX</td>
<td>1</td>
<td>XXX</td>
<td>XXX</td>
<td>1b1</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>CROSS_BOXerox2</td>
<td>XXX</td>
<td>XXX</td>
<td>1</td>
<td>XXX</td>
<td>XXX</td>
<td>[v]</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>CROSS_BOXerox3</td>
<td>XXX</td>
<td>XXX</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>0x2</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>CROSS_USB_MODE</td>
<td>XXX</td>
<td>XXX</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>b1</td>
<td></td>
</tr>
</tbody>
</table>

> Reporting

Testbench Group List

dashboard | hierarchy | modlist | groups | tests | asserts | userdata | hvp

Group Instance: CROSS_Bb10b_CL_VRF_100

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
<th>Weight</th>
<th>Goal</th>
<th>At Least</th>
<th>AUTO BIN MAX</th>
<th>PRINT MISSING</th>
</tr>
</thead>
<tbody>
<tr>
<td>CROSS_Bb10b_CL_VRF_100</td>
<td>100.00</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>CROSS_Bb10b_CL_VRF_101</td>
<td>100.00</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>CROSS_Bb10b_CL_VRF_103</td>
<td>100.00</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>CROSS_Bb10b_CL_VRF_106</td>
<td>100.00</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>CROSS_Bb10b_CL_VRF_137</td>
<td>100.00</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>CROSS_Bb10b_CL_VRF_18</td>
<td>100.00</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>
## Results & Analysis

> Comparison Between Predefined vs Dynamic Flow

<table>
<thead>
<tr>
<th>Coverage Methodology</th>
<th>Data Metrics</th>
<th>Performance/Results</th>
</tr>
</thead>
</table>
| Pre-Defined Coverage Database      | • 5000 covergroups created  
• ~15000 coverage bins & no cross-coverage support  
• 25000 lines of extra code got added for each SERDES block  
• Total 12800 tests are running as part of the whole regression suit | • Total Compilation time including parsing & elaboration was at 25 minutes for one SERDES block/QUAD  
• For a subsystem with 2 SERDES-QUAD, the compilation time was around 35~40 minutes  
• Simulation time for 1 test with 1024KB data transfer was 25 minutes  
• Merging time for all 12800-coverage database from individual tests, is ~3 to 3.5 hours |

<table>
<thead>
<tr>
<th>Coverage Methodology</th>
<th>Data Metrics</th>
<th>Performance/Results</th>
</tr>
</thead>
</table>
| Dynamic Coverage Database          | • 5000 covergroups created  
• ~15000 coverage bins & 126800 cross coverage bins were created  
• <100 lines of code are added, this is constant for block level & subsystem level  
• Total 12800 tests are running as part of the whole regression suit | • On a VCS based simulation platform, the time consumed to create the whole coverage database was between 12~15 seconds  
• Compilation time reduced to 10~12 minutes  
• For a subsystem with 2 SERDES-QUAD, the compilation time is around 18 minutes  
• Simulation time for 1 test with 1024KB data transfer is same as before, no significant change  
• Merging time for all 12800-coverage database from individual tests, is ~3 to 3.5 hours, which is comparable with previous flow |
Results & Analysis

> Conclusion

>> This approach fits well into any chip or IP tapeout execution
>> Makes verification engineer’s life a bit easier through a push button methodology for register coverage creation & collection
>> The actual effort can be quickly put into analysis, as oppose to spending time in coverage creation and dealing with higher compilation overhead
Thank You