Functional Coverage in SystemVerilog

Jason Sprott, Verilab
jason.sprott@verilab.com

Agenda

- Introduction
- Overview of cover property
- Overview of covergroup
- Compare cover property with covergroup
- Coding for coverage result analysis
- Coverage feedback

What Do We Cover?

- Specified functionality
  - Interface and internal protocols, expected usage, performance/QoS, configurations, etc.
- Design implementation features
  - Unspecified behaviour, FSMs, states or sequences not obvious from the outside
- Abstract data
  - Inter-related configurations (HW and SW), scenarios/sequences
  - It’s an important metric in determining completeness against a plan

SVA Cover Property

```
hburst_single_write : cover property ( @(posedge hclk) (hreset) throughout ((p_hburst == `SINGLE) && p_hwrite && ok_response) );
```

- Similar to a concurrent assertion
- Used to monitor SVA sequences and properties
- Used to cover any interesting event
- Defines a single coverage point
SVAB Cover Property

- Similar to a concurrent assertion
- Used to monitor SVA sequences and properties
- Used to cover any interesting event
- Defines a single coverage point

```vhdl
hburst_single_write : cover property ( @(posedge hclk) (!hreset) throughout ((p_hburst == `SINGLE) && p_hwrite && ok_response));
```

What Coverage Results Are Collected?

- Results for Sequences
  - Number attempted
  - Number matched
  - Recommendation: only cover sequences

- Results for Properties
  - Number attempted
  - Number passed
  - Number failed
  - Number vacuous passes

Turning A Property Into A Sequence

- We can turn a properties into sequence

```
req `1:10 ack `1 (lack) `1 idle
```

- What's important is that we stick to a consistent methodology
### Summary of cover property

<table>
<thead>
<tr>
<th></th>
<th>package</th>
<th>interface</th>
<th>module</th>
<th>program</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>cover property</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

- Embedded in the design or separate module
- Good for design/RTL level coverage
- Handles complex temporal expressions
- Not good at mapping samples to multiple bins
- Can’t be encapsulated in a class
  - Limits use in testbench

### What is a covergroup? (1)

- User-defined type
- Encapsulates a set of coverage samples
  - Coverage with the same sample condition
  - State and transitions (sequences)
  - Cross coverage of samples can be defined
  - Maps sampled data to multiple coverage bins

```verilog
covergroup p_burst
    SINGLE
    INCR
    WRAP16:
    coverpoint
    bins single = ...
    bins burst = ...
```

- Maps sampled data to multiple coverage bins
- Procedural interface for sampling and control
- Works with classes in an OO testbench
What is a covergroup? (2)

covergroup
  API
  options
  coverpoint
    API
    options
    bins
  cross
    API
    options
    bins

What is a covergroup? (3)

Procedural interface for group, e.g.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample()</td>
<td>Triggers sampling of the group</td>
</tr>
<tr>
<td>start()</td>
<td>Start collecting coverage</td>
</tr>
<tr>
<td>stop()</td>
<td>Stop collecting coverage</td>
</tr>
<tr>
<td>get_coverage()</td>
<td>Get type coverage</td>
</tr>
<tr>
<td>get_inst_coverage()</td>
<td>Get instance coverage</td>
</tr>
<tr>
<td>set_inst_name()</td>
<td>Set instance name</td>
</tr>
</tbody>
</table>

What is a covergroup? (4)

Procedural interface for coverpoint/cross, e.g.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start()</td>
<td>Start collecting coverage</td>
</tr>
<tr>
<td>stop()</td>
<td>Stop collecting coverage</td>
</tr>
<tr>
<td>get_coverage()</td>
<td>Get type coverage</td>
</tr>
<tr>
<td>get_inst_coverage()</td>
<td>Get instance coverage</td>
</tr>
</tbody>
</table>

What is a covergroup? (5)

Coverage options for groups, coverpoint and cross, e.g.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>per_instance</td>
<td>Only valid at group level, turns on tracking on a per instance basis</td>
</tr>
<tr>
<td>at_least</td>
<td>Minimum number of hits for each bin</td>
</tr>
<tr>
<td>weight</td>
<td>Weight used to compute the coverage score</td>
</tr>
<tr>
<td>goal</td>
<td>Target goal as a percentage</td>
</tr>
<tr>
<td>name</td>
<td>Name for covergroup instance</td>
</tr>
<tr>
<td>comment</td>
<td>Comment string for the associated element. It is saved in coverage database and appears in reports</td>
</tr>
</tbody>
</table>
What is a covergroup? (6)

Filtering on individual bins, e.g.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iff()</code></td>
<td>Optional condition for disabling coverage on a coverpoint, cross, or bin</td>
</tr>
<tr>
<td><code>ignore_bins</code></td>
<td>Marks values or transitions to be ignored. These are excluded from</td>
</tr>
<tr>
<td></td>
<td>coverage calculation</td>
</tr>
<tr>
<td><code>illegal_bins</code></td>
<td>Marks values or transitions as illegal in a coverpoint. If they occur a run-</td>
</tr>
<tr>
<td></td>
<td>time error is issued</td>
</tr>
<tr>
<td><code>binsof(x)</code></td>
<td>Selects the bins of coverpoint <code>x</code>, whose values intersect with range <code>y</code></td>
</tr>
</tbody>
</table>

Manually specified, or automatically generated bins, e.g.

```
bin high = [32'hFFFF_FF00:$];  // 1 bin covering a range
bin other[8] = [32'h0000_0021:32'hFFFF_FEFF];  // 8 bins across range
```

Summary of covergroups

<table>
<thead>
<tr>
<th>Feature</th>
<th>Cover Property</th>
<th>Covergroup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Interface</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Module</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Program</td>
<td>✔</td>
<td>❌</td>
</tr>
<tr>
<td>Class</td>
<td>❌</td>
<td>✔</td>
</tr>
</tbody>
</table>

- A must for coverage in an OO testbench
- Good for design/RTL level coverage
- Handles complex crosses and multiple bins
- Lots of filtering features
- Good control of coverage measurements

Now let’s compare the two types of coverage
Sample Event (1)

```verilog
cover property c_single : cover property (  
  @(posedge clk) iff (!reset)  
  ((p_hburst == SINGLE) &&  
    ok_response)  
);
```

```verilog
covergroup cg_ahb @(posedge clk);
  cp_hbust : coverpoint p_hbust;
endgroup
```

Sample Event (2)

```verilog
cover property c_single : cover property (  
  @(cover_me) iff (!reset)  
  ((p_hburst == SINGLE) &&  
    ok_response)  
);
```

```verilog
covergroup cg_ahb @(posedge clk);
  cp_hbust : coverpoint p_hbust;
endgroup
```

Sample Event (3)

```verilog
cover group cg_ahb @(posedge clk);
  cp_hbust : coverpoint p_hbust;
endgroup
```

```verilog
cg_ahb.sample();
```

```verilog
c_event cover_me;
```

```verilog
c_event cover_me; // in procedural code
```

```verilog
c_event cover_me; // in procedural code
```

```verilog
c_event cover_me; // in procedural code
```

```verilog
c_event cover_me; // in procedural code
```

Sample Event (3)

```verilog
c_event cover_me;
```

```verilog
c_event cover_me; // in procedural code
```

```verilog
c_event cover_me; // in procedural code
```

```verilog
c_event cover_me; // in procedural code
```
State Coverage

- c_single : cover property ( @(posedge clk) iff (!reset) ((p_hburst == SINGLE) && ok_response) );

  - cover property
  - Condition on sample
  - Samples RTL state

Sequence Coverage (Single Variable)

- c_fsm_startup : cover property ( @(posedge clk) iff (!reset) (state == IDLE) ##[1:5] (state == READY) );

  - cover property
  - Supports complex temporal relationships

Sequence Coverage (Multi Variable)

- c_req2ack : cover property ( @(posedge clk) iff (!reset) (req ##[1:5] ack #1 (!ack && !req) );

  - cover property
  - Supports complex temporal relationships

Hook-up A Covergroup To A Sequence

- s_inst_commit : sequence ( bit[7:0] inst_submitted;
  @(posedge clk) (req, inst_submitted = inst) ##[1:20] (commit, do_cover(inst_submitted));

  - covergroup
  - Doesn't handle this
  - However we can ...

  // triggers a covergroup sample
  function void do_cover(bit[7:0] inst);

  // put data somewhere covergroup can see it ...
  // trigger sample
cg_foo.sample();
endfunction

- We can attach actions to a sequence: e.g. store data in a local variable
- Or call function when it matches
- Function triggers covergroup sample
- The sequence must be used in a verification statement to be evaluated
Sequences

A few things to remember with SVA …

SVA: Unwanted Matches Gotcha

```verilog
c_unwanted_matches : cover property(
    @(posedge clk)
    x[1:6] y z
);
```

SVA: Some Things To Remember

- Sequences can have multiple matches, even if they have an endpoint, e.g. seq1.ended
- A property that is a sequence (no implication) behaves like first_match()
- The RHS (consequent) of a property behaves like first_match()
- The LHS (antecedent) cannot be a property
Mapping Samples onto Multiple Bins

- cover property
  - Doesn't handle this

- covergroup
  - Automatically creates bins for enums and variable ranges
  - Lots of useful features

```
enum {INC, DEC, NO_CHANGE} count_modes;
bit [31:0] address;
covergroup cg_ahb;
  cp_modes   : coverpoint count_modes;
  cp_address : coverpoint address {
    bins no_access = {[0:255]};
    bins other[8] = {[256:$]};
  }
endgroup
```

Multiple Bins with SVA

It's clumsy but some automation is possible ...

Using “generate” for Multiple Bins

- Burst Type
  - 00 LINEAR
  - 01 WRAP-4
  - 10 WRAP-8
  - 11 WRAP-16

- Read/Write
  - 0 READ
  - 1 WRITE

- Response Type
  - 00 OK
  - 01 RETRY
  - 10 X
  - 11 ERROR

We want to ignore this value

Total number of cover points to generate = $4 \times 2 \times 3 = 24$

Using “generate” for Multiple Bins

```
sequence s_burst_xfer(burst_type write_en response_type);
  @(posedge clk)
  (xfer_started && (burst_o == burst_type)) &&
  (wr_en == write_en) &&[0:8]
  (resp_o == response_type);
endsequence : s_burst_xfer
```

Reuse the same sequence for each bin
The values for comparison are passed as arguments
Using “generate” for Multiple Bins

sequence s_burst_xfer(burst_type, write_en, response_type);
@posedge clk
(sfer_started && (burst_o == burst_type) &&
(write_en == write_en) && (resp_o == response_type);
endsequence : s_burst_xfer

generate
for(i=0;i<4;i++) begin :
burst_type
for(j=0;j<2;j++) begin :
write_en
for(k=0;k<4;k++) begin :
response_type
if(k != 2)
c_burst_xfer_write : cover property(
s_burst_xfer(i,j,k) );
end
end
end
endgenerate

Cross Coverage of Sampled Data

cover property vs covergroup

cover property
- Doesn’t handle this

covergroup
- Automatically handles crosses
- Lots of filtering options

Using with Classes In Testbenches

class DmaTransactionCoverage;
DmaTransaction t;
covergroup cg_foo;
cp_src_master : coverpoint t.src_master;
cp_dst_master : coverpoint t.dst_master;
endgroup
endclass

Handling Some Abstraction With SVA

We can’t use classes, but we can use structs ...
Using Structs With Cover Statements

```c
typedef struct 
{ 
  bit [3:0] burst_type; 
  bit wr_en; 
  bit [1:0] response; 
} bus_transaction;

bus_transaction my_transaction;
```

sequence s_wrap8_write_retry(
  t.burst_type == `WRAP8) && (t.wr_en == 1) &&
endsequence : s_wrap8_write_retry

c_wrap8_write_retry : cover property (
  @ cover_bus_transaction s_wrap8_write_retry my_transaction)

User-defined event Cover specific transaction

Comparing Coverage Methods

<table>
<thead>
<tr>
<th></th>
<th>cover property</th>
<th>covergroup</th>
</tr>
</thead>
<tbody>
<tr>
<td>In design</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>In a class (OO testbench)</td>
<td>❌</td>
<td>✓</td>
</tr>
<tr>
<td>States</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Single var. sequences</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Complex TEs</td>
<td>✓</td>
<td>❌*</td>
</tr>
<tr>
<td>Multiple bins</td>
<td>❌</td>
<td>✓</td>
</tr>
<tr>
<td>Crosses</td>
<td>❌</td>
<td>✓</td>
</tr>
<tr>
<td>Control goals &amp; weights</td>
<td>❌</td>
<td>✓</td>
</tr>
</tbody>
</table>

* We can get a sequence to trigger a covergroup sample

Coding for Coverage Analysis (1)

- Assume humans will be reading coverage reports
- Include comments for groups, coverpoints and crosses
- Rename covergroup instances to something meaningful

Sensible Names and Comments etc.
Don’t collect coverage you are not using
- Be tough with this (it’s easy to go overboard)

- Weed out unwanted auto-generated bins
- Use filtering to target only interesting samples

Coding for Coverage Analysis (3)

- Make sure goals and weights represent your intentions
- Watch out for sample points only used in crosses
- Only enable instance or type coverage when it’s useful
  - Type coverage: cumulative for all instances of a covergroup
  - Instance coverage: is for a specific instance of a covergroup

Exclude coverpoints only used in cross by setting goals and weights to 0
Exclude Pointless Statistics

- Exclude group coverage by setting weights and goals

Accurate Weights For Cover/Cross

- cross goal = number of all non zero weight bins

Accurate Weights For Group

- group goal = number of cover and cross points with non-zero weight

Coding for Coverage Analysis (4)

- Review coverage like any other code – there will be bugs
Coverage Feedback

Useful Built-in Coverage Run-time Query Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_coverage()</td>
<td>Calculates type coverage (0-100)</td>
</tr>
<tr>
<td>get_inst_coverage()</td>
<td>Calculates instance coverage (0-100)</td>
</tr>
<tr>
<td>get_coverage()</td>
<td>Overall coverage for all groups (0-100)</td>
</tr>
</tbody>
</table>

A Good Idea?

- Creating new tests (new constraints, or seed) and test grading is trivial by comparison
- Coverage results rarely map cleanly back to specific constraints
- You’ll miss things
  - Coverage models evolve – they are not completed early
  - You will not know all the interesting states yet
- Running the same tests with new seeds can find bugs – even with perceived 100% coverage

Extending Class For Coverage Feedback

```java
class DmaChannelTransactionWithCovFeedback extends DmaChannelTransaction;

// src_master weight distribution
protected integer weight_src_m0=10;
protected integer weight_src_m1=10;
...

// allow import of coverage results
protected DmaChannelTransactionCoverage cov;
constraint src_master_dist_c {
  src_master dist {
    M0 := weight_src_m0,
    M1 := weight_src_m1,
  } ...
}

function void pre_randomize();
if (cov.dma_channel.src_master_m0.get_inst_coverage() == 100)
  weight_src_m0 = 1; // reduce probability of occurring
...
endfunction : pre_randomize
endclass : DmaChannelTransactionWithCovFeedback
```

Summary

- Coverage in design
  - Specified functionality and implementation
  - Scenarios with complex temporal relationships
  - Crosses between different samples
  - Handled using cover property and covergroups
- Coverage in testbench
  - Should fit into object-oriented testbench design
  - Complex stimulus scenarios
  - Data sampling decoupled from design
  - Best handled using covergroups
- Coverage feedback
  - Possible, but typically not the best solution
Functional Coverage in SystemVerilog

Jason Sprott, Verilab

jason.sprott@verilab.com