Functional Coverage Development Tips: Do's and Don'ts

by Samrat Patel, ASIC Verification Engineer, and Vipul Patel, ASIC Engineer, eInfochips

INTRODUCTION

A verification engineer's fundamental goal is ensuring that the device under test (DUT) behaves correctly in its verification environment. Achieving this goal gets more difficult as chip designs grow larger and more complex, with thousands of possible states and transitions.
A comprehensive verification environment must be created that minimizes wasted effort. Using functional coverage as a guide for directing verification resources by identifying tested and untested portions of the design is a good way to do just that.

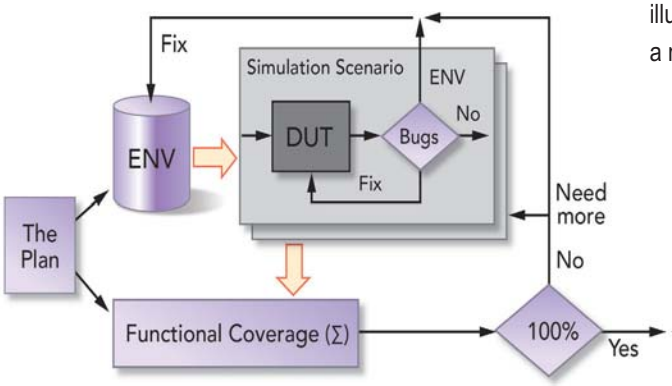


Figure 1: Functional Coverage Flow Diagram

Functional coverage is user-defined, mapping all functionality defined in the test plan to be tested to a cover point. Whenever the functionality is hit during simulation, the functional coverage point is automatically updated. A functional coverage report can be generated summarizing how many coverage points were hit, metrics that can be used to measure overall verification progress.

The two key aspects of functional coverage:

- It is user-specified and is not automatically inferred from the design.
- It is based on the design specification and is thus independent of the actual design code or its structure.

This article gives several scenarios that might come up when building a functional coverage model. We also illustrate use of several constructs relevant to building such a model.

USE OF DEFAULT SEQUENCE FOR TRANSITION COVERAGE

Similar to 'Default Construct' which is useful for catching unplanned or invalid values, 'Default Sequence' is used to catch all transitions (or sequences) that do not lie within any of the defined transition bins.

When any non-default sequence transition is incremented or any previously specified bin transition is not in pending state, then 'bin allother' of that cover point will be incremented.

In the following example, a transition from value 1 to 0 is specified by bins flag_trans. So if any transition occurs from 1 to 0, then it will be captured by flag_trans bins.

Here, by using default sequence you can capture other transitions that are not specified exclusively.

```
cp_flag: coverpoint data [24] {
  bins flag_trans   = (1 => 0);
  bins allother     = default sequence;
}
```

While using default sequence, it's best to avoid the following scenarios:

Scenario-1: The default sequence specification does not accept multiple transition bins (the [] notation). It will give an error. Therefore, avoid using the default sequence with multiple transition bins.

```
cp_flag: coverpoint data [24] {
  bins flag_trans   = (1 => 0);
  bins allother []   = default sequence;
}
```

Scenario-2: The default sequence specification cannot be explicitly ignored. It will be an error for bins designated as ignore_bins to specify a default sequence. Therefore, avoid using the default sequence with ignore_bins.

```
cp_flag: coverpoint data[24]  {
   bins flag_trans            = (1 => 0);
  ignore_bins allother  = default sequence;
  }
```

EXCLUSION OF CROSS COVERAGE AUTO GENERATED BINS

Problem: Suppose the requirement is to do cross coverage between two cover points and capture only specific user defined bins as follows:

```
cp_thr: coverpoint data [11:8] {
  bins thr_val_0 = {0};
  bins thr_val_1 = {1};
}
cp_reg_addr: coverpoint addr {
  bins reg_addr_1 = {12'h01C};
  bins reg_addr_2 = {12'h020};
 }
cr_thr_addr: cross cp_thr, cp_reg_addr {
  bins thr_add = binsof(cp_reg_addr) intersect
{12'h01C};
 }
```

In the above example, the coverage report will capture user defined bins along with auto generated bins. However, the requirement is to capture only specific user bins.

The limitation of cross coverage is that even on specifying only user bins, it will also generate cross coverage bins automatically.

Solution: To disable auto generated cross bins, you should use ignore_bins as shown below:

```
cr_thr_addr: cross cp_thr, cp_reg_addr {
  bins thr_add                  = binsof(cp_reg_addr)
                                    intersect {12'h01C};
  ignore_bins thr_add_ig  = !binsof(cp_reg_addr)
                                    intersect {12'h01C};
 }
```

Another way is that instead of specifying user defined bins simply use ignore_bins.

```
cr_thr_addr: cross cp_thr, cp_reg_addr {
  ignore_bins thr_add  = binsof(cp_reg_addr)
                              intersect {12'h020};
 }
```

AVOID USING MULTIPLE BIN CONSTRUCT (THE [] NOTATION) WITH NONCONSECUTIVE REPETITION

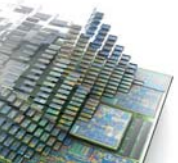
The nonconsecutive repetition is specified using trans_item [= repeat_range]. The required number of occurrences of a particular value is specified by the repeat_range.

Problem: Using nonconsecutive repetition with the multiple bins construct (the [] notation) gives a fatal run time error as follows:

```
cp_flag: coverpoint data [24] {
  bins flag_trans[]  = (1 => 0[=3]);
}
```

Simulation Error:
** Fatal: (vsim-8568) Unbounded or undetermined varying length sequences formed using Repetitive/ Consecutive operators are not allowed in unsized Array Transition bins. A transition item in bin 'err_flag' of Coverpoint 'cp_err_flag' in Covergroup instance '\/ tx_env_pkg::tx_coverage::cg_err_ctrl_status_reg ' has an operator of kind '[=]'. Please fix it

Solution: During nonconsecutive repetition, any number of sample points can occur before the first occurrence of the specified value and between each occurrence of the specified value. The transition following the nonconsecutive repetition may occur after any number of sample points, as long as the repetition value does not occur again.



As length varies for nonconsecutive repetition, you cannot determine it. So avoid using the multiple bin construct (the [] notation) with nonconsecutive repetition.

```
cp_flag: coverpoint data[24]  {
  bins flag_trans  = (1 => 0[=3]);
}
```

Here, in flag_trans bins 0[=3] is same as … 0 => …=> 0…=> 0 means any number of sample points can occur before occurrence and between each occurrence of the specified value. But as you have specified bins flag_trans as static bin, it will avoid fatal errors.

AVOID USE OF DEFAULT

As per LRM, the default specification defines a bin that's not associated with any of the defined value bins. That is, it catches the values of coverage points that do not lie within any of the defined bins.

Problem-1: If you use the multiple bin construct (the [] notation) then it will create a separate bin for each value.

In the following example, the first bin construct associates bin rsvd_bit with the value of zero. Every value that does not match bins rsvd_bit is added into its own distinct bin.

```
cp_rsvd_bit: coverpoint data[31:13] iff (trans ==
                                          pkg::READ) {
  bins rsvd_bit = {0};
  bins others[] = default;
}
```

But as mentioned above, if the coverage point has a large number of values and you run simulation for it, the simulator crashes giving the following fatal error:

```
# ** Fatal: The number of singleton values exceeded
the system limit of 2147483647 for unconstrained
array bin 'other' in Coverpoint
'data' of Covergroup instance '\/covunit/cg_err_reg'.
```

The question: do you really need 2147483647 values?

Solution:

Use default without multiple bin construct (the [] notation): If you use default without the multiple bin construct for large values, then it will create a single bin for all values, thus helping you avoid fatal errors.

```
cp_rsvd_bit: coverpoint data[31:13] iff (trans ==
                                          pkg::READ) {
  bins rsvd_bit = {0};
  bins others    = default;
}
```

Use ignore_bins: If you use the ignore_bins construct for large values, then it will ignore unnecessary large values, also helping you avoid fatal errors.

```
cp_rsvd_bit: coverpoint data[31:13] iff (trans ==
                                          pkg::READ) {
  bins rsvd_bit                = {0};
  ignore_bins  ig_rsvd_bit  = {[1:$]};
```

Problem-2: The coverage calculation for a cover point generally does not take into account the coverage captured by the default bin, which is also excluded from cross coverage.

In the following example, for cover point data, bins thr_val is specified as default. So values 0 to 15 are added into its own distinct bin. Also cover point data is used in cross coverage with addr cover point.

```
cp_thr: coverpoint data[11:8]  {
 bins thr_val_ 0    =  0;
 bins thr_val[15]  =  default;
}
cp_reg_addr: coverpoint addr  {
  bins reg_addr_1  = {12'h01C};
  bins reg_addr_2  = {12'h020};
 }
cr_thr_addr : cross  cp_thr, cp_reg_addr;
```

Here, cover point data has no coverage because bins are specified using "default" and also there is no cross coverage because we don't have coverage for cover point data.

Solution:

Use wildcard bins: It captures the combination of all possible values.

```
cp_thr: coverpoint data[11:8]  {
bins thr_val_ 0                =  0;
wildcard bins thr_val_wc =  {[4'b???1]};
}
```

Use min/max ($) operators: It specifies minimum or maximum values range.

```
cp_thr: coverpoint data[11:8]  {
 bins thr_val_ 0    =  0;
  bins thr_val_op = {[1:$]};;
}
```

AVOID USE OF ILLEGAL_BINS

If you specify any bin as illegal_bins, then it will remove unused or illegal values from the overall coverage calculation.

Problem: In the following example, during the read operation the reserved bit value should be zero;

```
cp_rsvd_bit: coverpoint data[31:25] iff (trans ==
                                        pkg::READ) {
  bins rsvd_bit                 = {0};
  illegal_bins il_rsvd_bit  = {[1:$]};
}
```

any other value will produce an error.

In this scenario, there are certain questions that arise:

Question-1: *Is it reasonable to rely on a passive component to capture an active error? Because, if you want to capture active errors using illegal_bins, and you do not use the passive coverage component (i.e., you turn it off and use only the active component), then you will not capture any active errors.*

Solution-1:

Use assertion and checkers to capture active errors: If you want to capture active errors, use assertion and checkers. And if you have defined checkers and assertions and still want to cross check for any run time error through the passive component, then you can also use illegal_bin.

Question-2: *How do you avoid such a condition without using illegal_bins?*

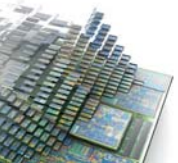
Solution-2:

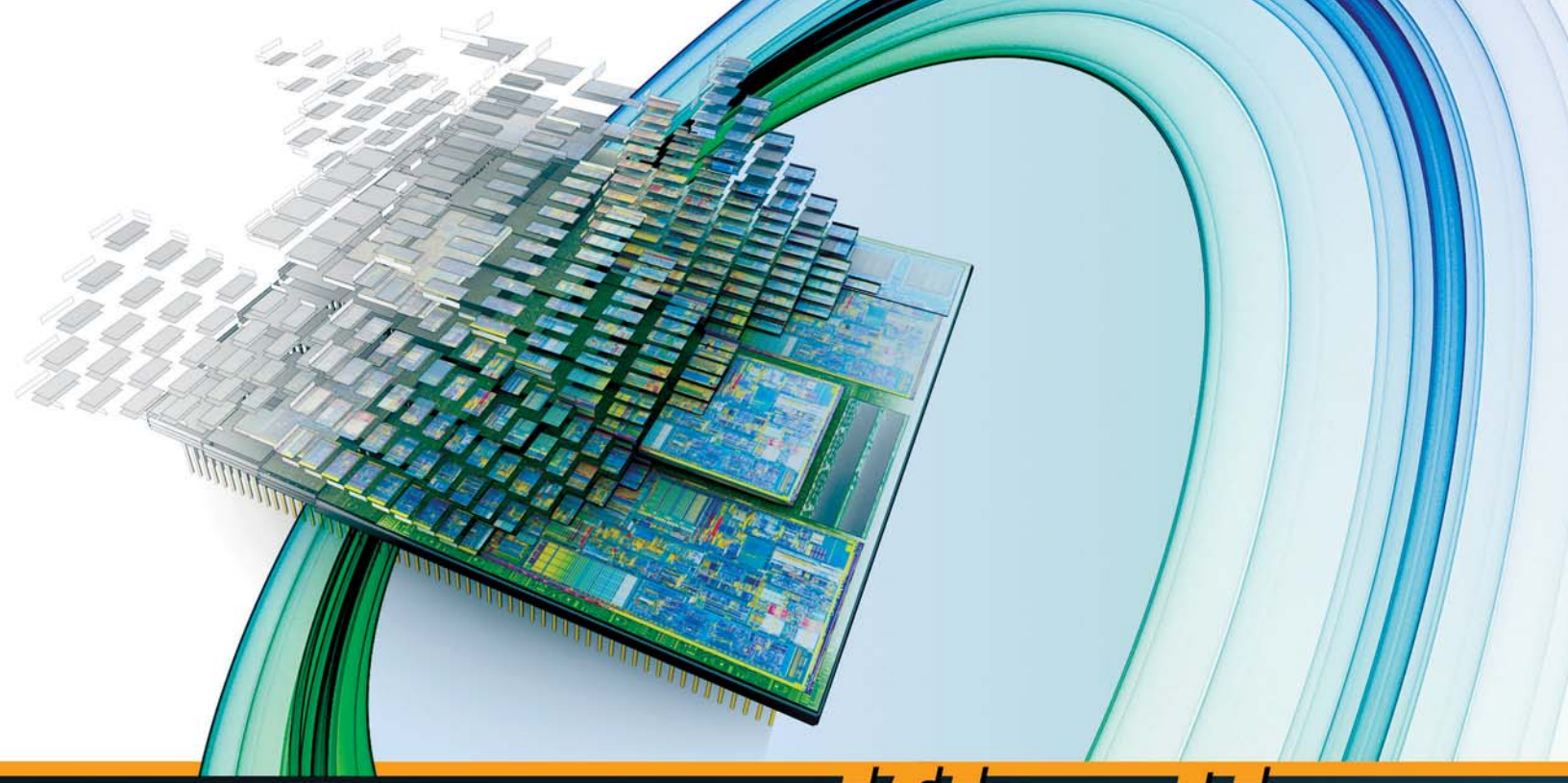
```
cp_rsvd_bit: coverpoint data[31:25] iff (trans ==
                                        pkg::READ) {
  bins rsvd_bit                 = {0};
  ignore_bins ig_rsvd_bit  = {[1:$]};
}
```

Use ignore_bins: Since ignore_bins ignore other values and does not throw any type of active errors, it will exclude those values from overall coverage.

Best of luck experimenting with these constructs.
We hope they are useful in your verification endeavors.

.





verification HORIZONS

Editor: Tom Fitzpatrick
Program Manager: Rebecca Granquist

Wilsonville Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777
Phone: 503-685-7000

To subscribe visit:
www.mentor.com/horizons

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM

Mentor Graphics®

www.mentor.com