INTRODUCTION
The fundamental goal of a verification engineer is to ensure that the Device Under Test (DUT) behaves correctly in its verification environment. As chip designs grow larger and more complex with thousands of possible states and transitions, a comprehensive verification environment must be created that minimizes development effort. To minimize effort, functional coverage is used as a guide for directing verification resources by identifying tested and untested portions of the design. The approach should give any verification engineer confidence in DUT functionality.

Functional coverage is a user-defined coverage which maps each functionality defined in the test plan to be tested to a cover point. Whenever the functionality to be tested is hit in the simulation, the functional coverage point is automatically updated. A functional coverage report can be generated which summarizes how many coverage points were hit. Functional coverage metrics can be used to measure the progress of a verification effort.

The key aspects of functional coverage are as follows:

• It is user-specified and 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 represents some of the important features of a functional coverage model that will be useful for verification engineers to develop a functional coverage model with high quality. The scenarios discussed subsequently cover some of the finer aspects of using these constructs more productively.

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, the 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 (shown on the next page) you can capture other transitions that are not specified exclusively.

Figure 1: Functional Coverage Flow Diagram
While using default sequence, 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 a default sequence with multiple transition bins.

```vhdl
cp_flag: coverpoint data [24] {
  bins flag_trans = (1 => 0);
  bins alother [] = 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 a default sequence with ignore_bins.

```vhdl
cp_flag: coverpoint data [24] {
  bins flag_trans = (1 => 0);
  ignore_bins alother = 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:

```vhdl
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};
}
```

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:

```vhdl
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. This will ignore other bins except required user-defined bins so we can get the expected result.

```vhdl
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 NON-CONSECUTIVE REPETITION**

The non-consecutive 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 non-consecutive repetition with multiple bins construct (the \([\) notation) gives a fatal run time error as follows:

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

Simulation Error:

```
** Fatal: (vsim-8568) Unbounded or undetermined
```

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:

```vhdl
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. This will ignore other bins except required user-defined bins so we can get the expected result.

```vhdl
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 NON-CONSECUTIVE REPETITION**

The non-consecutive 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 non-consecutive repetition with multiple bins construct (the \([\) notation) gives a fatal run time error as follows:

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

Simulation Error:

```
** Fatal: (vsim-8568) Unbounded or undetermined
```
Solution: During non-consecutive 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 non-consecutive repetition may occur after any number of sample points, as long as the repetition value does not occur again. As length varies for non-consecutive repetition, you cannot determine it. So you should avoid using the multiple bin construct (the [] notation) with non-consecutive repetition.

AVOID USE OF DEFAULT

As per LRM, the default specification defines a bin that is associated with none of the defined value bins. It catches the coverage point values that do not lie within any of the defined bins.

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

In 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 to its own distinct bin.

But as mentioned above, if the coverage point has a large number of values and you run simulation for it, then the simulator crashes and it gives 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'.

Here the question is: Do you really need 2147483647 values?

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

Use ignore_bins: If you use ignore_bins construct for large values then it will ignore unnecessary large values, similarly avoiding fatal errors.

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

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

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

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

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

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

| 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, data cover point has no coverage because bins are specified using “default;” also there is no cross coverage because we don’t have coverage for data cover point.

Solution:
Use wildcard bins: This captures combinations of all possible values.

```plaintext
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.

```plaintext
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, this 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; any other value will return an error.

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

Solution-1:
Use assertion and checkers to capture active errors:
If you want to capture active errors then you can do so using assertion and checkers. This will throw errors; the problem of relying on a passive component to capture active errors will be resolved.

Note:
1. If you have defined checkers and assertions and still want to cross check for any run time error through passive component, you can also use illegal_bins.
2. If you are sure of any scenario that should not occur in any condition, then you can use illegal_bins.

Question-2: How to avoid such condition without using illegal_bins?

Solution-2:
Use ignore_bins: This ignores other values and does not throw any type of active errors. Also, it excludes those values from overall coverage.

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

We hope these suggestions are useful in your verification efforts!

In this scenario, certain questions arise:

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