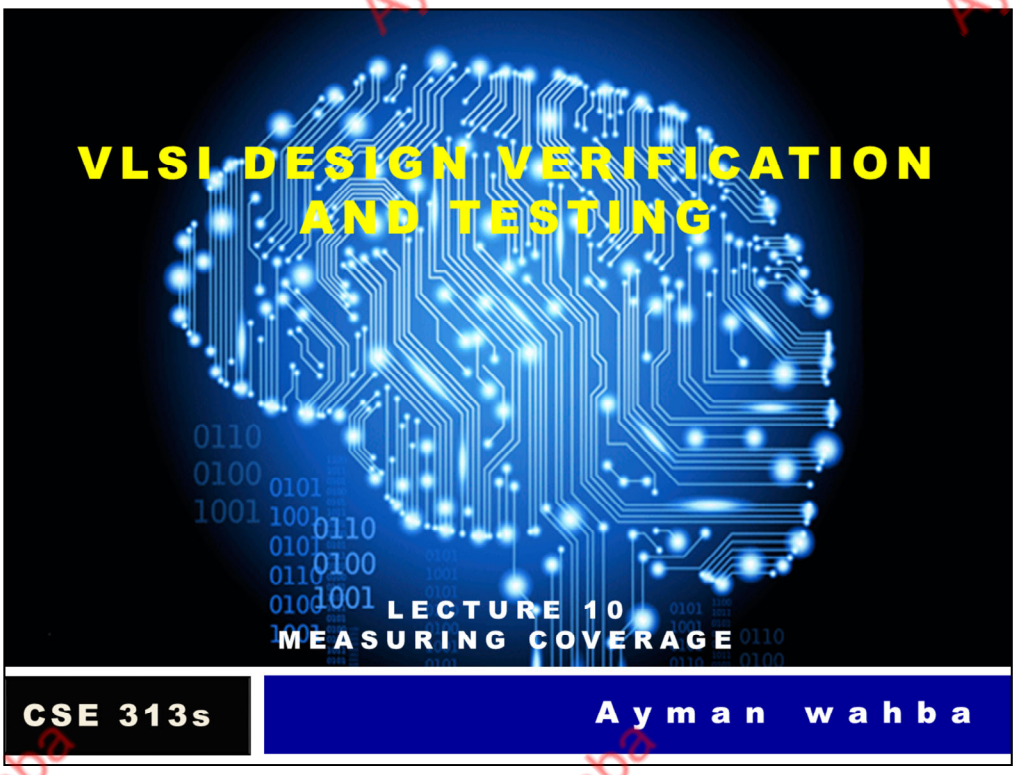

VLSI DESIGN VERIFICATION AND TESTING
LECTURE 10
MEASURING COVERAGE
CSE 313s
Ayman wahba

Functional Coverage Example

2

- To see the detailed coverage reports on eda playground (www.edaplayground.com) you need to make the following setting:
  - Select Mentor Questa Simulator
  - Put -voptargs=+acc=npr as Run Options
  - Tick on Use run.do TCL file
- Create a *run.do* TCL file should contain the following commands:

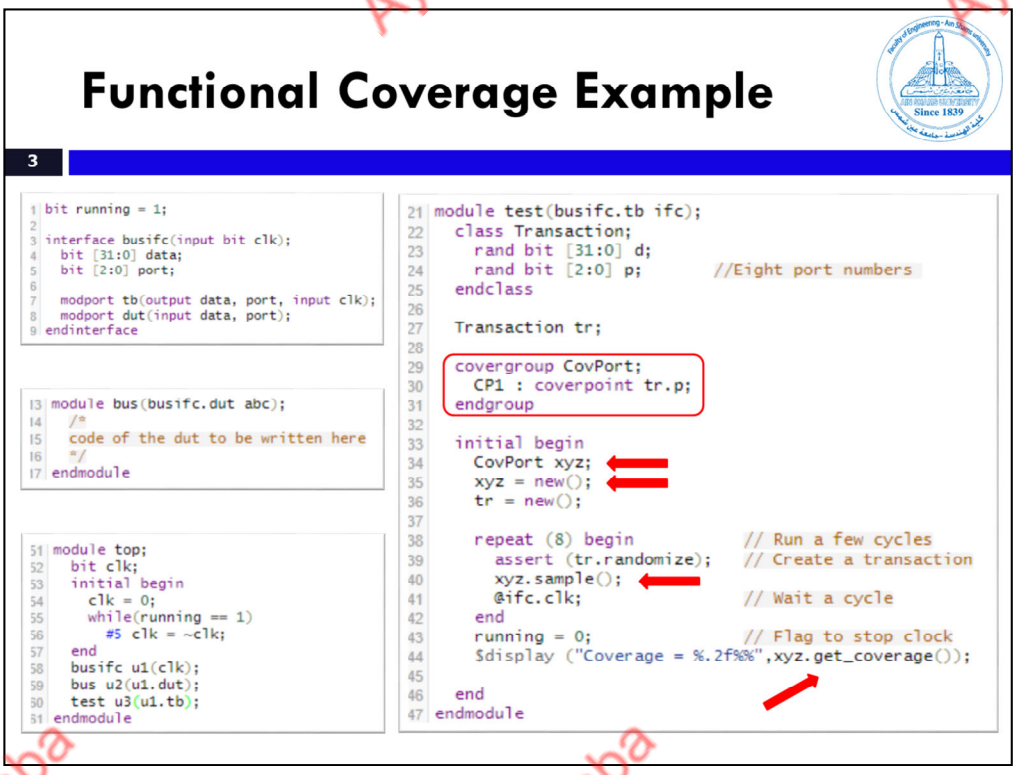
testbench.sv | run.do

```
1 run -all
2 coverage report -detail
3
```

Testbench + Design
SystemVerilog/Verilog
UVM / OVM
None
Other Libraries
None
OVL 2.8.1
SVUnit 2.11
Enable TL-Verilog
Enable Easier UVM
Enable VUnit
Tools & Simulators
Mentor Questa 2021.3
Compile Options
-timescale 1ns/1ns
Run Options
-voptargs=+acc=npr
Run Time: 10 ms
Use run.do Tcl file
Use run.bash shell script
Open EPWave after run
Download files after run

- T

Functional Coverage Example



- To measure functional coverage, you begin with the verification plan and write an executable version of it for simulation.
- In your SystemVerilog testbench, sample the values of variables and expressions. These sampling locations are known as *cover points*.
- Multiple cover points that are sampled at the same time (such as when a transaction completes) are placed together in a *cover group*.
- The shown code describes a bus that accepts data and port number (the details of the bus are omitted). We want to test the bus with all possible values of the port number. We will use randomly generated tests, and then measure the coverage to make sure that all the port numbers have been used. There are 8 possible values for the port. The verification plan requires that every value be tried.
- The testbench samples the value of the port field using the CovPort *cover group*. The above code generates 8 random transactions. Did they cover all the possible values of the port? Did your testbench generate them all?
- To know the answer, we have to look at the coverage report. The next slide shows part of a coverage report generated by the simulator.

- **Practical note:**
- If $finish is called explicitly, or implicitly (when a *program* finishes execution), eda playground will not run the TCL file run.do, and you will not see the detailed report. You will see only the line reporting the coverage from the get_coverage() function.
- That is why we used a module for the testbench instead of a program. (We previously talked about the differences between *modules* and *programs*)

Functional Coverage Example

4

```
# ----------------------------------------------------------------------------------------------------
# Covergroup                                          Metric       Goal       Bins    Status
#  Covergroup instance \/top/u3/#ublk#502948#51/xyz    75.00%        100          -    Uncovered
#    covered/total bins:                                   6          8          -
#    missing/total bins:                                   2          8          -
#    % Hit:                                           75.00%        100          -
#    Coverpoint CP1                                   75.00%        100          -    Uncovered
#        covered/total bins:                               6          8          -
#        missing/total bins:                               2          8          -
#        % Hit:                                       75.00%        100          -
#        bin auto[0]                                       2          1          -    Covered
#        bin auto[1]                                       2          1          -    Covered
#        bin auto[2]                                       0          1          -    ZERO
#        bin auto[3]                                       1          1          -    Covered
#        bin auto[4]                                       0          1          -    ZERO
#        bin auto[5]                                       1          1          -    Covered
#        bin auto[6]                                       1          1          -    Covered
#        bin auto[7]                                       1          1          -    Covered
#
```

- The simulator automatically generates a bin for every value for the signal to be sampled,
- If a certain value happened, the contents for the corresponding bin will be incremented.
- Bins are the basic units of measurement for functional coverage. When you specify a variable or expression in a cover point, System Verilog(SV) creates a number of "bins" to record how many times each value has been seen.
- The bins can be explicitly defined by the user or automatically (implicitly) created by SV. The number of bins created implicitly can be controlled by *auto_bin_max* parameter as will be seen later.
- To calculate the coverage for a point that has 3-bit variable (i.e. has the domain 0:7) System Verilog creates 8 bins (one for each of the values 0, 1, 2, …7). So coverage is then measured as the number of non-empty bins divided by the total number of bins. If during simulation, values belonging to seven bins are sampled, the report will be 7/8 or 87.5% coverage for this point.

- As you can see, the testbench generated the values 0,1,3,5,6, and 7, but never generated a port value of 2 nor 4.
- The "*Goal*" column specifies how many hits are needed before a bin is considered covered.
- To improve your functional coverage, the easiest strategy is to just run more simulation cycles, or to try new random seeds.
- Look at the coverage report for items with two or more hits. Chances are that you just need to make the simulation run longer or to try new seed values.
- If a cover point had zero or one hit, you probably have to try a new strategy, as the

testbench is not creating the proper stimulus.

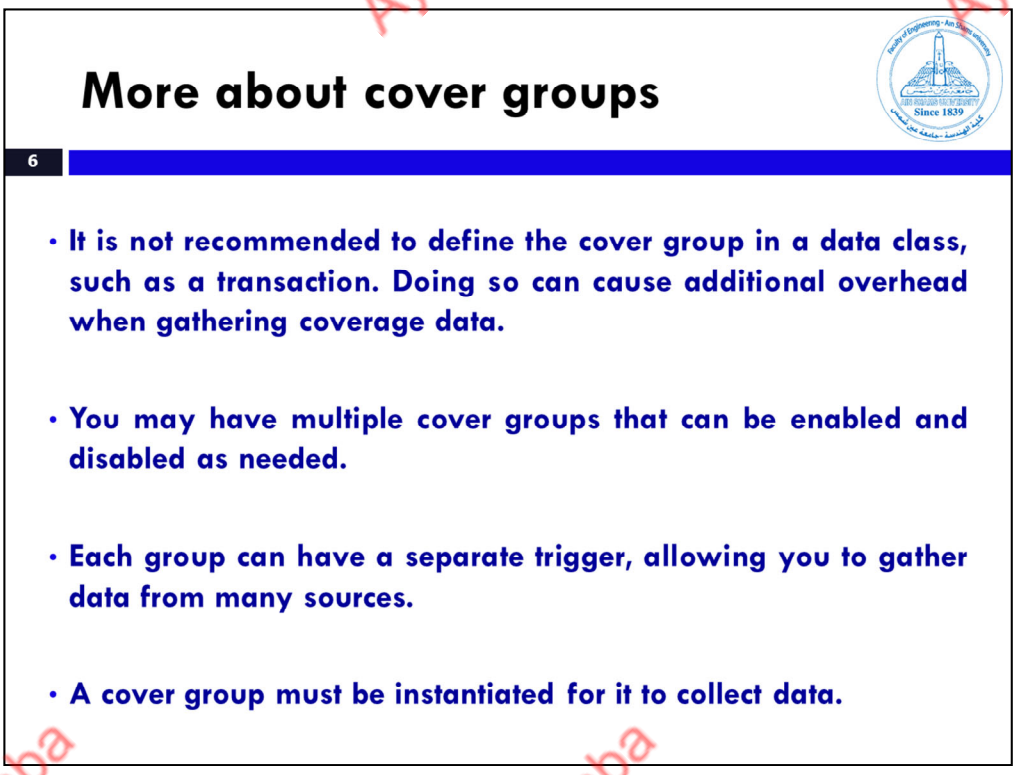
More about cover groups

5

- A cover group is similar to a class - you define it once and then instantiate it one or more times.
- It contains *cover points*, *options*, *formal arguments*, and an *optional trigger*.
- Cover points inside a cover group are sampled at the same time.
- Cover group names should be as clear as possible.
- Cover groups can be defined in a *class* or at the *program* or *module* level.

- A cover group is similar to a class - you define it once and then instantiate it one or more times.
- It contains *cover points*, *options*, *formal arguments*, and an *optional trigger*.

- A cover group contains one or more data points (called cover points), all of which are sampled at the same time. You should create very clear cover group names that explicitly indicate what you are measuring and, if possible, reference to the verification plan.
- **For example**, the name Parity_Errors_In_Hexaword_Cache_Fills may seem verbose, but when you are trying to read a coverage report that has dozens of cover groups, you will appreciate the extra detail.
- You can also use the *comment option* for additional descriptive information (will be seen later).
- A cover group can be defined inside a *class* or at the *program* or *module* level. It can sample any visible variable such as program/module variables, signals from an interface, or any signal in the design (using a hierarchical reference). A cover group inside a class can sample variables in that class, as well as data values from embedded classes.



- **A recommendation:** Don't define the cover group in a data class, such as a transaction, as doing so can cause additional overhead when gathering coverage data.
- Each cover group may have a separate trigger, allowing you to gather data from many sources.
- A cover group must be instantiated for it to collect data. If you forget, no error message about null handles is printed at run-time, but the coverage report will not contain any mention of the cover group. This rule applies for cover groups defined either inside or outside of classes.

Data Sampling

7

- When you specify a variable or expression in a cover point, System Verilog creates a number of "bins" to record how many times each value has been seen.

- At the end of each simulation, a database is created with all bins that have a token in them.
- Analysis tools read all databases and generate a report with the coverage for each part of the design and for the total coverage.

- How is coverage information gathered? When you specify a variable or expression in a cover point, SystemVerilog creates a number of "bins" to record how many times each value has been seen.
- These bins are the basic units of measurement for functional coverage.
- If you sample a one-bit variable, a maximum of two bins are created. If you have 3-bit variables, a maximum of 8 bis are created.
- You can imagine that System Verilog drops a token in one or the other bin every time the cover group is triggered.
- At the end of each simulation, a database is created with all bins that have a token in them.
- You then run an analysis tool that reads all databases and generates a report with the coverage for each part of the design and for the total coverage.

Individual Bins and Total Coverage

8

- The total number of possible values, is known as *domain*.
- There may be one value per bin or multiple valued bins.
- Coverage percentage is calculated as follows:
  - Assume a cover point that is a 3-bit variable.
  - It has the domain 0:7, so you have eight bins.
  - If the simulation fills only in 7 bins, the report will show 7/8 or 87.5% coverage for this point.
- If you have more than one cover point in a cover group, all these points are combined to show the coverage for the entire group.
- If there are more than one cover group, the coverage percentages are combined together for all of the groups.

- To calculate the coverage for a point, you first have to determine the total number of possible values, also known as the domain.
- There may be one value per bin or multiple values (i.e. you can make a bin for 0, a bin for 1, … a bin for 7. Alternatively you can make one bin for 0 and 1, and another bin for 2 and 3, and so on).
- Coverage percentage is calculated as follows:
  A cover point that is a 3-bit variable has the domain 0:7 and is normally divided into eight bins. If, during simulation, values belonging to seven bins are sampled, the report will show 7/8 or 87.5% coverage for this point.
  All these points are combined to show the coverage for the entire group (in case the group has more than one cover point), and then all the groups are combined to give a coverage percentage for all the simulation databases.
- Now you can better predict when verification of the design will be completed.

Creating Bins Automatically

9

- System Verilog automatically creates bins for cover points.
- For an expression that is *n* bits wide, there are 2^n possible values. For the 3-bit variable port, there are 8 possible values.
- The domain for enumerated data types is the number of named values.
- You can also explicitly define bins as will be seen later.

- As you saw in the previous examples, System Verilog automatically creates bins for cover points.
- It looks at the domain of the sampled expression to determine the range of possible values.
- For an expression that is N bits wide, there are 2^N possible values. For the 3-bit variable port, there are 8 possible values.
- The domain for enumerated data types is the number of named values.
- You can also explicitly define bins as will be seen later.

Limiting the Number of Automatic Bins Created

10

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit [31:0] d;
    rand bit [2:0] p;        //Eight port numbers
  endclass

  Transaction tr;

  covergroup CovPort;
    CP1 : coverpoint tr.p {option.auto_bin_max = 2;}
  endgroup

  initial begin
    CovPort xyz;
    xyz = new();
    tr = new();

    repeat (8) begin              // Run a
      assert (tr.randomize);      // Creat
      xyz.sample();
      @ifc.clk;                   // Wait
    end
    running = 0;                  // Flag
    $display ("Coverage = %.2f%%",xyz.g

  end
endmodule
```

```
# COVERGROUP COVERAGE:
# ----------------------------------------------------------------------
# Covergroup                 Metric       Goal       Bins    Status
#
# ----------------------------------------------------------------------
#     Coverpoint CP1         100.00%        100          -    Covered
#         covered/total bins:       2          2          -
#         missing/total bins:       0          2          -
#         % Hit:             100.00%        100          -
#         bin auto[0:3]             5          1          -    Covered
#         bin auto[4:7]             3          1          -    Covered
#
# TOTAL COVERGROUP COVERAGE: 100.00%  COVERGROUP TYPES: 1
```

- The cover group *option.auto_bin_max* specifies the maximum number of bins to automatically create.
- The default value of *option.auto_bin_max*, is 64 bins.
- If the domain of values in the cover point variable or expression is greater than the value of *auto_bin_max*, System Verilog divides the range into *auto_bin_max* bins.
- For example, a 16-bit variable has 2^{16} possible values. When divided on 64 bins (i.e. 2^6 bins) and so each of the 64 bins covers ($2^{16}/2^6 = 2^{10}$ values). Each bin will cover 1,024 values.
- In reality, you may need to specify the number of bins by yourself. As shown on the above code. Here we limit the number of bins to 2 bins. The sampled variable is still *port*, which is three bits wide, for a domain of eight possible values. The first bin holds the lower half of the range, 0-3, and the other hold the upper values, 4-7.
- The coverage report shows the two bins. This simulation achieved 100% coverage because the eight port values were mapped to two bins. Since both bins have sampled values, your coverage is 100%.
- The above code used *auto_bin_max* as an option for the cover point only. You can also use it as an option for the entire group, as will be shown in the next slide.

Limiting the Number of Automatic Bins Created - continued

11

```
covergroup CovPort;
  option.auto_bin_max = 2;
  CP1 : coverpoint tr.p;
  CP2 : coverpoint tr.d;
endgroup
```

This applies to all the cover points in the group

```
COVERGROUP COVERAGE:
----------------------------------------------------------------------------
Covergroup                          Metric       Goal       Bins    Status

----------------------------------------------------------------------------
Coverpoint CP1                      100.00%        100          -    Covered
    covered/total bins:                   2          2          -
    missing/total bins:                   0          2          -
    % Hit:                          100.00%        100          -
    bin auto[0:3]                         5          1          -    Covered
    bin auto[4:7]                         3          1          -    Covered
Coverpoint CP2                      100.00%        100          -    Covered
    covered/total bins:                   2          2          -
    missing/total bins:                   0          2          -
    % Hit:                          100.00%        100          -
    bin auto[0:2147483647]                5          1          -    Covered
    bin auto[2147483648:4294967295]       3          1          -    Covered
```

- The above code used *auto_bin_max* as an option for the entire group.

Sampling Expressions

12

```
1  bit running = 1;
2
3  interface busifc(input bit clk);
4    bit [31:0] data;
5    bit [2:0] port;
6    bit [3:0] kind;
7
8    modport tb(output data, port, kind,
9               input clk);
10   modport dut(input data, port, kind);
11 endinterface

15 module bus(busifc.dut abc);
16   /*
17   code of the dut to be written here
18   */
19 endmodule

54 module top;
55   bit clk;
56   initial begin
57     clk = 0;
58     while(running == 1)
59       #5 clk = ~clk;
60   end
61   busifc u1(clk);
62   bus u2(u1.dut);
63   test u3(u1.tb);
64 endmodule
```

```
23 module test(busifc.tb ifc);
24   class Transaction;
25     rand bit [31:0] d;
26     rand bit [2:0] p;        //Eight port numbers
27     rand bit [3:0] k;
28   endclass
29
30   Transaction tr;           // Create a handle
31
32   covergroup CovPort;
33     CP1 : coverpoint (tr.p + tr.k); // 4-bit expression
34   endgroup
35
36   initial begin
37     CovPort xyz;
38     xyz = new();
39     tr = new();
40
41     repeat (70) begin          // Run some cycles
42       assert (tr.randomize);   // Create a transaction
43       xyz.sample();
44       @ifc.clk;                // Wait a cycle
45     end
46     running = 0;               // Flag to stop clock
47     $display ("Coverage = %.2f%%",xyz.get_coverage());
48
49   end
50 endmodule
```

The coverage report will contain 16 bins

- You can sample expressions, but always check the coverage report to be sure you are getting the values you expect.
- When you are adding a 3-bit operand to 4-bit operand, System Verilog considers the expression width to be 4 bits, and thus, if this expression is used as a cover point, only 16 bins will be generated automatically.
- Actually, this will be a problem, as the result of 7 + 15, for example, will not find a bin to be put in. You have bins for the values 0 to 15, but the values 16, 17, 18, 19, 20, 21, 22 will have no bins. In this case you may get 100% coverage, while the values 16 … 22 are never generated.
- Even if you cast the result to be put in 5 bits, this will automatically create 32 bins, for the values from 0 to 31 but the values 23, 24, 25, 26, 27, 28, 29, 30, 31 will never be generated, so you will never get 100% coverage.

Sampling Expressions

13

```
COVERGROUP COVERAGE:
----------------------------------------------------------------------
Covergroup                Metric       Goal       Bins    Status

----------------------------------------------------------------------
Coverpoint CP1          100.00%        100          -    Covered
    covered/total bins:      16         16          -
    missing/total bins:       0         16          -
    % Hit:              100.00%        100          -
    bin auto[0]               3          1          -    Covered
    bin auto[1]               6          1          -    Covered
    bin auto[2]               3          1          -    Covered
    bin auto[3]               8          1          -    Covered
    bin auto[4]               3          1          -    Covered
    bin auto[5]               2          1          -    Covered
    bin auto[6]               3          1          -    Covered
    bin auto[7]               9          1          -    Covered
    bin auto[8]               7          1          -    Covered
    bin auto[9]               1          1          -    Covered
    bin auto[10]              8          1          -    Covered
    bin auto[11]              4          1          -    Covered
    bin auto[12]              1          1          -    Covered
    bin auto[13]              1          1          -    Covered
    bin auto[14]              7          1          -    Covered
    bin auto[15]              4          1          -    Covered
```

We got 100% coverage, however this is fake, because there is no guarantee that any of the values 16, 17, 18, 19, 20, 21, or 22 has been generated.

- A quick run with 70 random transactions showed that we got 100% coverage, but this is across only 16 bins.
- This is a fake coverage, as there is no guarantee that any of the values 16, 17, 18, 19, 20, 21, or 22 has been generated.

Sampling Expressions

14

```
23 module test(busifc.tb ifc);
24   class Transaction;
25     rand bit [31:0] d;
26     rand bit [2:0] p;        //Eight port numbers
27     rand bit [3:0] k;
28   endclass
29
30   Transaction tr;           // Create a handle
31
32   covergroup CovPort;
33     CP1 : coverpoint (tr.p + tr.k + 5'b00000); // 5-bit expression
34   endgroup
35
36   initial begin
37     CovPort xyz;
38     xyz = new();
39     tr = new();
40
41     repeat (2000) begin          // Run some cycles
42       assert (tr.randomize);    // Create a transaction
43       xyz.sample();
44       @ifc.clk;                 // Wait a cycle
45     end
46     running = 0;                // Flag to stop clock
47     $display ("Coverage = %.2f%%",xyz.get_coverage());
48
49   end
50 endmodule
```

Run so many cycles

We added the dummy value 5'b00000 to make the result of the expression composed of 5 bits. Consequently, there will be 32 bins.

Having 32 bins, you will never get 100% coverage, as shown in the next slide

Sampling Expressions

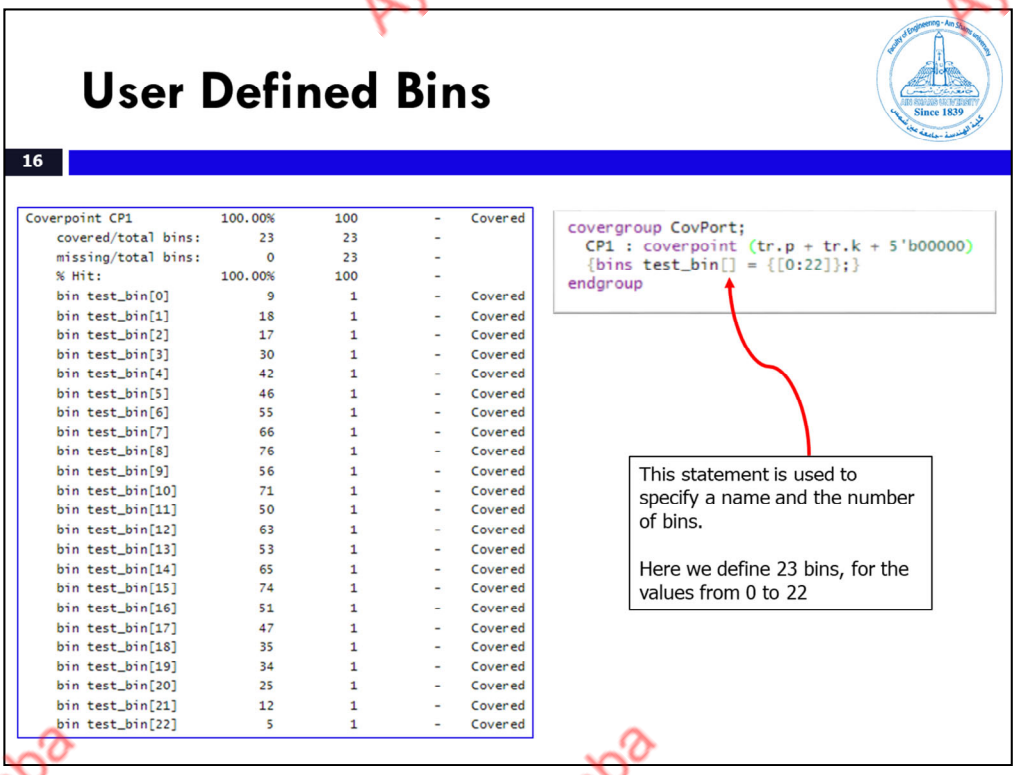
15

Coverpoint CP1	71.87%	100	-	Uncovered
covered/total bins:	23	32	-	
missing/total bins:	9	32	-	
% Hit:	71.87%	100		
bin auto[0]	17	1	-	Covered
bin auto[1]	36	1	-	Covered
bin auto[2]	42	1	-	Covered
bin auto[3]	64	1	-	Covered
bin auto[4]	77	1	-	Covered
bin auto[5]	88	1	-	Covered
bin auto[6]	107	1	-	Covered
bin auto[7]	132	1	-	Covered
bin auto[8]	148	1	-	Covered
bin auto[9]	125	1	-	Covered
bin auto[10]	141	1	-	Covered
bin auto[11]	106	1	-	Covered
bin auto[12]	119	1	-	Covered
bin auto[13]	107	1	-	Covered
bin auto[14]	130	1	-	Covered
bin auto[15]	136	1	-	Covered
bin auto[16]	105	1	-	Covered
bin auto[17]	102	1	-	Covered
bin auto[18]	70	1	-	Covered
bin auto[19]	60	1	-	Covered
bin auto[20]	45	1	-	Covered
bin auto[21]	26	1	-	Covered
bin auto[22]	17	1	-	Covered
bin auto[23]	0	1	-	ZERO
bin auto[24]	0	1	-	ZERO
bin auto[25]	0	1	-	ZERO
bin auto[26]	0	1	-	ZERO
bin auto[27]	0	1	-	ZERO
bin auto[28]	0	1	-	ZERO
bin auto[29]	0	1	-	ZERO
bin auto[30]	0	1	-	ZERO
bin auto[31]	0	1	-	ZERO

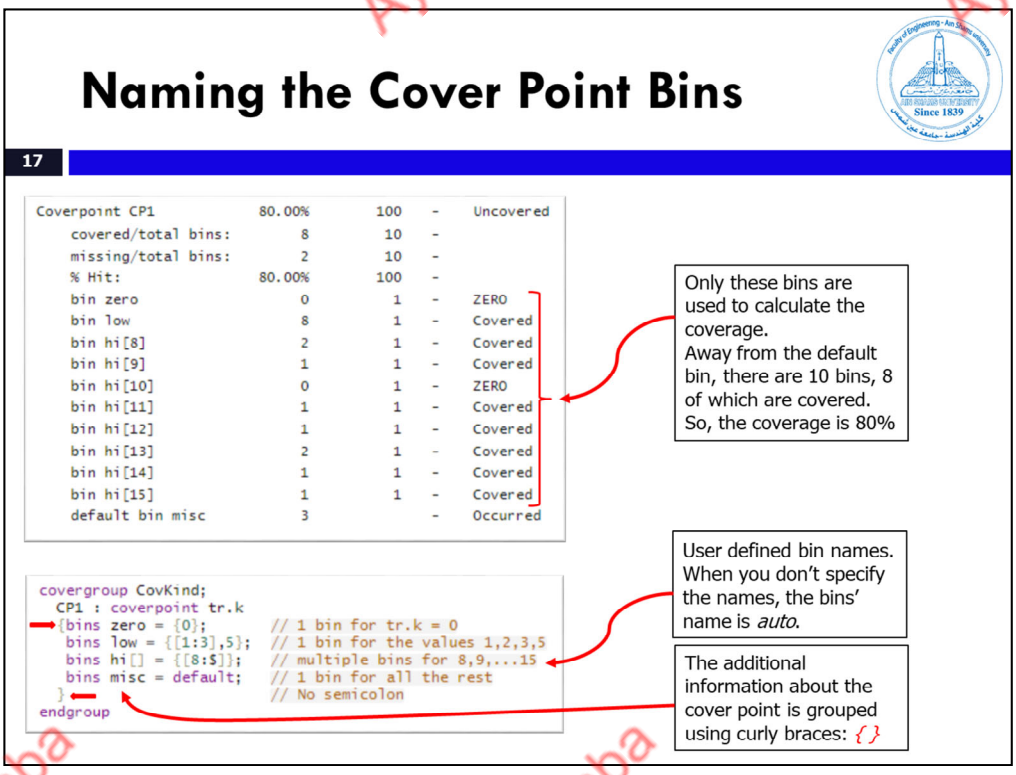
The maximum coverage you will get is 71.87%, because the values from 23 to 31 will never happen.

So having 16 bins will give a fake coverage report and having 32 bins will never give 100% coverage.

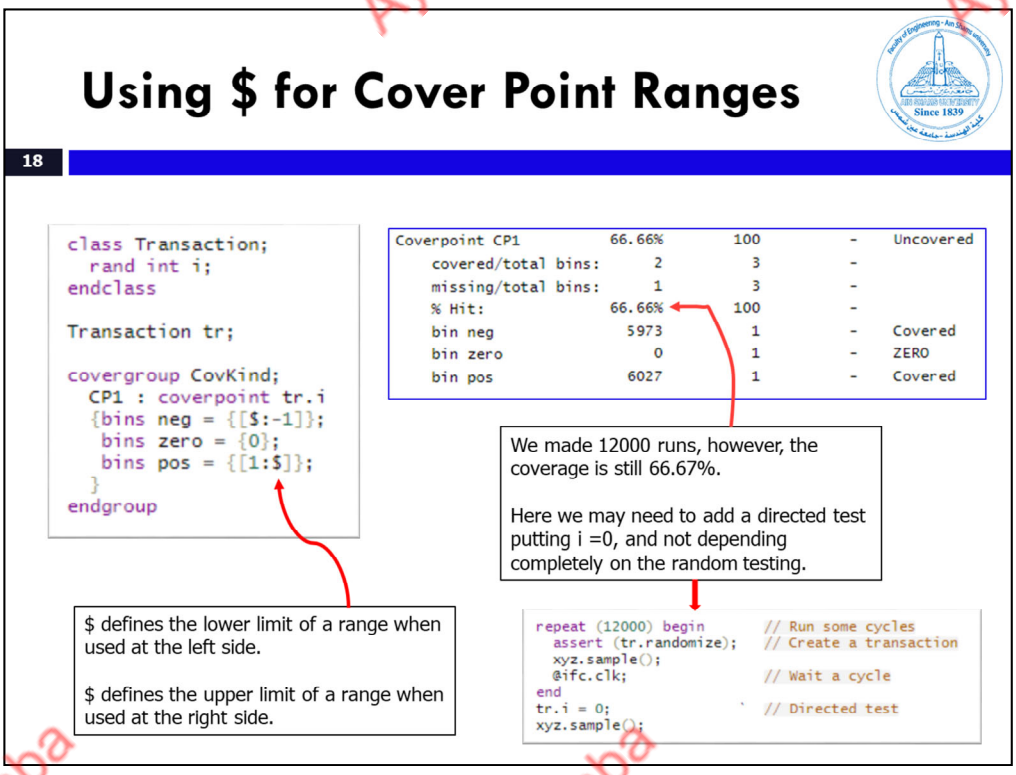
To get correct coverage report you must have 23 bins for the values from 0 to 22.



- Automatically generated bins are okay for anonymous data values, such as counter values, addresses, or values that are a power of 2.
- For other values, you should explicitly name the bins to improve accuracy and ease coverage report analysis.
- System Verilog automatically creates bin names for enumerated types, but for other variables you need to give names to the interesting states.
- The easiest way to specify bins is with the *[ ]* syntax, as shown in the code above.



- The above code samples a 4-bit variable, *tr.k* that has 16 possible values.
  - The first bin is called *zero*. It counts the number of times *tr.k* is 0.
  - The values, 1-3 and 5, are all grouped into a single bin, *low*.
  - The upper eight values. 8-15. are kept in separate bins: hi[8], hi[9], hi[10], hi[11], hi[12], hi[13], hi[14], and hi[15].
  - Note how $ in the hi bin expression is used as a shorthand notation for the largest value for the sampled variable.
  - Lastly, using *default*, "misc" holds all values that were not previously chosen: 4, 6, and 7.
- When you define the bins. you are restricting the values used for coverage to those that are interesting to you. In this case, SystemVerilog no longer automatically creates bins, and it ignores values that do not fall into a predefined bin.
- Note that the additional information about the cover point is grouped using curly braces: *{ }*. This is because the bin specification is *declarative* code, not procedural code that would be grouped with begin ... end.
- Lastly, the final curly brace is NOT followed by a semicolon. just as an end never has a semicolon after it.
- More importantly, only the bins you create are used to calculate functional coverage. You get 100% coverage only as long as you get a hit in every specified bin. Values that do not fall into any specified bin are ignored.
- In general, if you are specifying bins, always use the *default* bin specifier to catch values that you may have forgotten.



- In the previous example, the range for *hi* uses a dollar sign ($) on the right side to specify the upper value.
- This is a very useful shortcut, as now you can let the compiler calculate the limits for a range.
- You can use the dollar sign on the left side of a range to specify the lower limit as well.
- In the above code, the $ in the range for bin *neg* represents the negative number furthest from zero: 32'h8000 _0000. or -2.147.483.648. whereas the $ in bin *pos* represents the largest signed positive value, 32'h7FFF _ FFFF. or 2.147.483.647.
- We ran the test 20 times, and got only values in *neg* and *pos* bins. So the coverage is only 66.67%.
- We increased the number of runs till we reached 12000 run, and still we couldn't find values in the *zero* bin. That mean that the value 0 is very hard to be generated using the random testing.
- In this case we added a directed test where we set tr.i = 0, explicitly in the code.

Conditional Coverage

19

```
module test(busifc.tb ifc);
  class Transaction;
    bit reset;
    rand bit [2:0] p;
  endclass

  Transaction tr;

  covergroup CovKind;
    CP1 : coverpoint tr.p iff(!tr.reset);
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    tr.reset = 0;
    repeat (100) begin          // Run some cycles
      assert (tr.randomize); // Create a transaction
      xyz.sample();
      @ifc.clk;                 // Wait a cycle
    end
    tr.reset = 1;
    repeat (100) begin          // Run some cycles
      assert (tr.randomize); // Create a transaction
      xyz.sample();
      @ifc.clk;                 // Wait a cycle
    end
    running = 0;                // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

```
covergroup CovKind;
  CP1 : coverpoint tr.port iff (!tr.reset);
endgroup
```

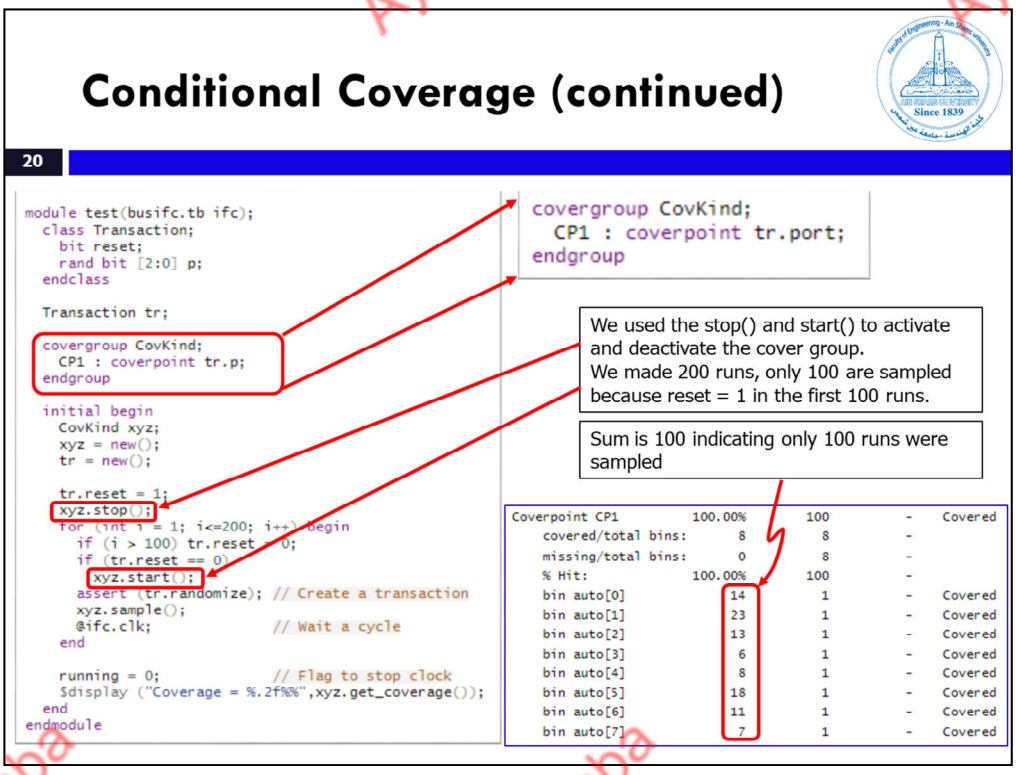
We made 200 runs, only 100 are sampled because reset = 1 in the last 100 runs.

Sum is 100 indicating only 100 runs were sampled

```
Coverpoint CP1              100.00%        100         -    Covered
    covered/total bins:           8          8         -
    missing/total bins:           0          8         -
    % Hit:                  100.00%        100         -
    bin auto[0]                  16          1         -    Covered
    bin auto[1]                  11          1         -    Covered
    bin auto[2]                  10          1         -    Covered
    bin auto[3]                  21          1         -    Covered
    bin auto[4]                  13          1         -    Covered
    bin auto[5]                  10          1         -    Covered
    bin auto[6]                  10          1         -    Covered
    bin auto[7]                   9          1         -    Covered
```

- You can use the *iff* keyword to add a condition to a cover point.
- The most common reason for doing so is to turn off coverage during reset so that stray triggers are ignored.
- The code shown on the left gathers only values of port when reset is 0. where reset is active-high.

Conditional Coverage (continued)

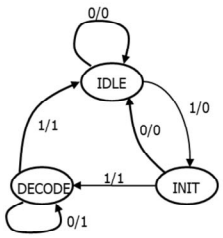


- Alternately. you can use the *start* and *stop* functions to control individual instances of cover groups, as illustrated in the above.

Creating Bins for Enumerated Types

21

```
module test(busifc.tb ifc);
  class Transaction;
    bit reset;
    rand bit [2:0] p;

    typedef enum {INIT, DECODE, IDLE} fsmstate_e;
    rand fsmstate_e pstate, nstate;
  endclass

  Transaction tr;

  covergroup CovKind;
    CP1 : coverpoint tr.pstate;
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (20) begin
      assert (tr.randomize); // Create a transaction
      xyz.sample();
      @ifc.clk;              // Wait a cycle
    end

    running = 0;             // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```



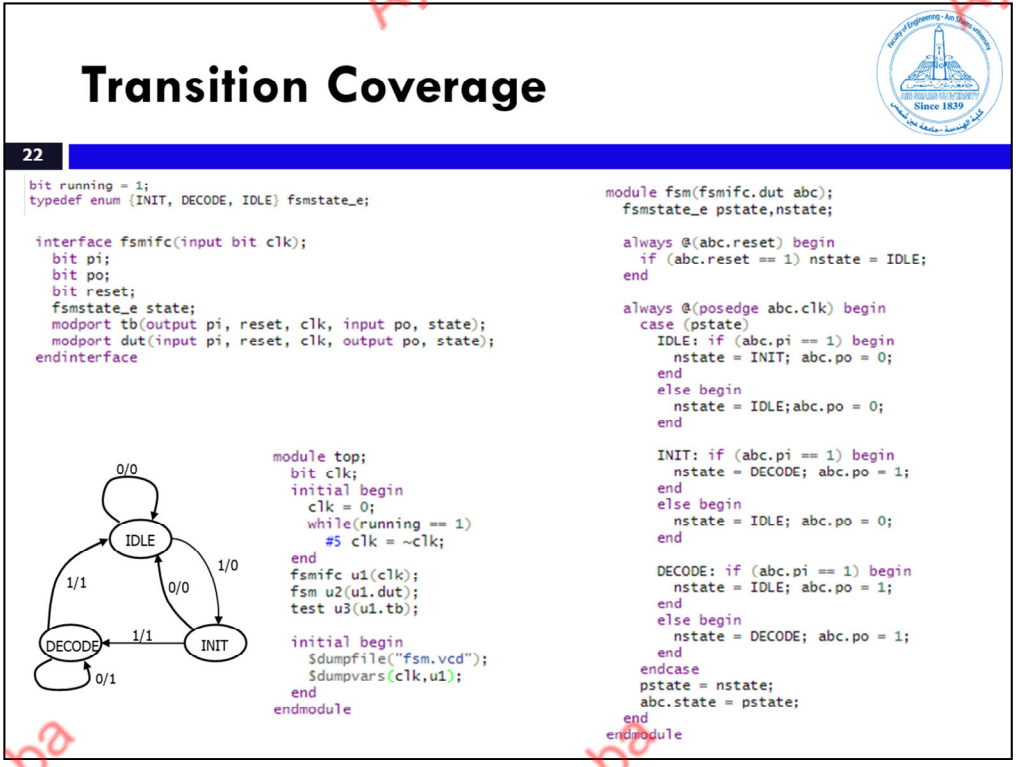

```
Coverpoint CP1          100.00%   100   -    Covered
    covered/total bins:       3     3   -
    missing/total bins:       0     3   -
    % Hit:                100.00%  100   -
    bin auto[INIT]            8     1   -    Covered
    bin auto[DECODE]          7     1   -    Covered
    bin auto[IDLE]            5     1   -    Covered
```

- For enumerated types, SystemVerilog creates a bin for each value.
- The coverage report generated for the shown code is showing the bins for the enumerated types.
- If you want to group multiple values into a single bin, you have to define your own bins

```
covergroup CovKind;
  CP1 : coverpoint tr.pstate
  {
   bins t1 = {0, 1}; // 0 for INIT, 1 for DECODE
   bins t2 = {2};    // 2 for IDLE
  }
 endgroup
```

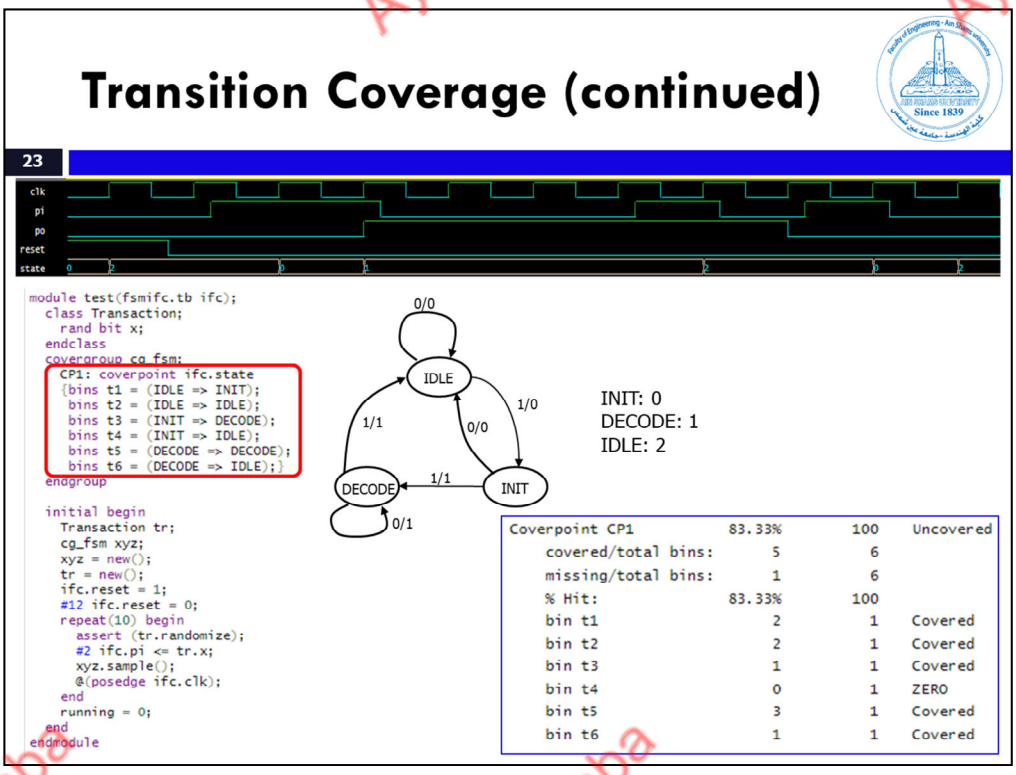
- Any bins outside the enumerated values are ignored unless you define a bin with the default specifier.
- When you gather coverage on enumerated types, *auto_bin_max* does not apply.

Transition Coverage



- Here we will design a finite state machine, that has three states.
- We need to generate a random input value and we want to make sure that all the transitions have been excited.

Transition Coverage (continued)



- You can specify state transitions for a cover point.
- In this way, you can tell not only what interesting values were seen but also the sequences.
- For example, you can check if he state ever went from INIT to IDLE, or from INIT to DECODE.
- The transition in a cover group is specified as follows: (INIT => IDLE) for example.
- You can quickly specify multiple transitions using ranges: the expression (1,2 => 3,4) creates the four transitions (1=>3), (1=>4), (2=>3), and (2=>4).
- You can specify transitions of any length. Note that you have to sample once for each state in the transition.
  So (0 = > 1 = > 2) is different from (0 = > 1 = > 1 = > 2) or
  (0 = > 1 = > 1 = > 1 = > 2).
- If you need to repeat values, as in the last sequence, you can use the shorthand form: (0 = > 1 [*3] = > 2). This is equivalet to (0 = > 1 = > 1 = > 1 = > 2)
- To repeat the value 1 for 3, 4, or 5 times. use 1 [* 3 : 5] .

Wild Cards in Cover Groups

24

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit[3:0] i;
  endclass

  Transaction tr;

  covergroup CovKind;
    CP1 : coverpoint tr.i
    {wildcard bins even = {4'bxzx0};
     wildcard bins odd = {4'b???1};
    }
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (10) begin          // Run a few cycles
      assert (tr.randomize);   // Create a transaction
      xyz.sample();
      @ifc.clk;                // Wait a cycle
    end
    running = 0;               // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

Any x, z, or ? in the expression is treated as a wildcard for 0 or 1

You use the wildcard keyword to create multiple states and transitions

- You use the wildcard keyword to create multiple values for bins.
- Any x, z, or ? in the expression is treated as a wildcard for 0 or 1.
- The shown code creates a cover point with a bin for even values and one for odd values.

Ignoring values

25

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit[3:0] only_0_to_5;
  endclass

  Transaction tr;

  covergroup CovKind;
    CP1 : coverpoint tr.only_0_to_5
    {ignore_bins high = {[6:15]};
    }
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (10) begin          // Run a few cycles
      assert (tr.randomize);   // Create a transaction
      xyz.sample();
      @ifc.clk;                // Wait a cycle
    end
    running = 0;               // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

```
Coverpoint CP1          83.33%   100   Uncovered
    covered/total bins:      5     6
    missing/total bins:      1     6
    % Hit:               83.33%   100
    ignore_bin high          5         Occurred
    bin auto[0]              1     1   Covered
    bin auto[1]              0     1   ZERO
    bin auto[2]              1     1   Covered
    bin auto[3]              1     1   Covered
    bin auto[4]              1     1   Covered
    bin auto[5]              1     1   Covered
```

Ignore the auto bin 6 to 15

- With some cover points, you never get all possible values. For instance, a 4-bit variable may be used to store just six values, 0-5. If you use automatic bin creation, you never get beyond 37.5% coverage. (6/16).
- There are two ways to solve this problem:
  - You can explicitly define the bins that you want to cover as shows before
  - Alternatively, you can let SystemVerilog automatically create bins, and then use *ignore_bins* to tell which values to exclude from functional coverage calculation.
- The original range of *only_0_to_5*, a four-bit variable is 0:15. The *ignore_bins* excludes the last 10 bins, which reduces the range to 0:5. So total coverage for this group is the number of bins with samples, divided by the total number of bins, which is 6 in this case.

Ignoring values (contined)

26

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit[3:0] only_0_to_5; //Only the values 0:5 are allowed
  endclass

  Transaction tr;

  covergroup CovKind;
    CP1 : coverpoint tr.only_0_to_5
    {option.auto_bin_max = 8;      //[0:1][2:3][4:5], ... [14:15]
     ignore_bins high = {[6:15]}; //ignore the upper 5 bins
    }
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (10) begin          // Run a few cycles
      assert (tr.randomize);   // Create a transaction
      xyz.sample();
      @ifc.clk;                // Wait a cycle
    end
    running = 0;               // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

```
Coverpoint CP1            100.00%  100   Covered
    covered/total bins:         3    3
    missing/total bins:         0    3
    % Hit:                100.00%  100
    ignore_bin high             5        Occurred
    bin auto[0:1]               1    1   Covered
    bin auto[2:3]               2    1   Covered
    bin auto[4:5]               2    1   Covered
```

We have only 8 bins, and we ignore values: 6 to 15 which are contained in the upper 5 bins, leaving only 3 bins.

- If you define bins either explicitly or by using the *auto_bin_max* option, and then ignore them, the ignored bins do not contribute to the calculation of coverage. In the above code sample, eight bins are initially created using the *auto_bin_max* option: [0:1], [2:3], [4:5], [6:7], [8:9], [10:11], [12:13], [14:15],
- However, then the 5 uppermost bins are eliminated by *ignore_bins,* and so at the end only three bins are created. This cover point can have coverage of 0%, 33%, 66%, or 100%.

Illegal Bins

27

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit[3:0] only_0_to_5; //Only 0:5 are allowed
  endclass

  Transaction tr;

  covergroup CovKind;
    CP1 : coverpoint tr.only_0_to_5 {
      illegal_bins high = {[6:15]}; // Give an error if seen
    }
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (10) begin          // Run a few cycles
      assert (tr.randomize);   // Create a transaction
      xyz.sample();
      @ifc.clk;                // Wait a cycle
    end
    running = 0;               // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

```
# ** Error: (vsim-8565) Illegal state bin was hit at value=11.
#    Time: 5 ns  Iteration: 1  Instance: /top/u3
# ** Error: (vsim-8565) Illegal state bin was hit at value=7.
#    Time: 15 ns  Iteration: 1  Instance: /top/u3
# ** Error: (vsim-8565) Illegal state bin was hit at value=13.
#    Time: 35 ns  Iteration: 1  Instance: /top/u3
# ** Error: (vsim-8565) Illegal state bin was hit at value=7.
#    Time: 40 ns  Iteration: 1  Instance: /top/u3
# ** Error: (vsim-8565) Illegal state bin was hit at value=6.
#    Time: 45 ns  Iteration: 1  Instance: /top/u3
# Coverage = 83.33%
```

If the testbench generates these values, an error will be issued

- Some sampled values not only should be ignored but also should cause an error if they are seen.
- This is best done in the testbench's monitor code, but can also be done by labeling a bin with *illegal_bins*.
- Use *illegal_bins* to catch states that were missed by the test's error checking.
- This also double-checks the accuracy of your bin creation: if an illegal value is found by the cover group, it is a problem either with the testbench or with your bin definitions.

Cross Coverage

28

- A cover point records the observed values of a single variable or expression.
- You may want to know the relation between two or more cover points to discover errors and their source and destination.
- Note that when you measure cross coverage of a variable with N values, and of another with M values, System Verilog needs N × M cross bins to store all the combinations.

- A cover point records the observed values of a single variable or expression.
- You may want to know not only which transactions have happened, but also which values were generated due to the occurrence of these transactons.
- For this you need cross coverage that measures what values were seen for two or more cover points at the same time.
- Note that when you measure cross coverage of a variable with N values, and of another with M values, System Verilog needs N × M cross bins to store all the combinations.

Basic Cross Coverage Example

29

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit [2:0] p;
    rand bit [3:0] k;
  endclass

  Transaction tr;

  covergroup CovKind;
    CP1 : coverpoint tr.p;
    CP2: coverpoint tr.k;
    cross CP1,CP2;
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (30) begin
      assert (tr.randomize); // Create a transaction
      xyz.sample();
      @ifc.clk;              // Wait a cycle
    end
    running = 0;             // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

```
                                  Metric       Goal
-------------------------------------------------------
Coverpoint CP1                   100.00%        100
    covered/total bins:                8          8
    missing/total bins:                0          8
    % Hit:                       100.00%        100
Coverpoint CP2                    87.50%        100
    covered/total bins:               14         16
    missing/total bins:                2         16
    % Hit:                        87.50%        100
Cross #cross__0#                  20.31%        100
    covered/total bins:               26        128
    missing/total bins:              102        128
    % Hit:                        20.31%        100
```

```
                                  Metric Goal Bins    Status
-------------------------------------------------------------
bin <auto[12],auto[0]>             3     1    -     Covered
bin <auto[5],auto[0]>              1     1    -     Covered
bin <auto[2],auto[0]>              3     1    -     Covered
bin <auto[0],*>                    0     1    8     ZERO
bin <auto[14],auto[7]>             0     1    1     ZERO
bin <auto[13],auto[7]>             0     1    1     ZERO
bin <auto[12],auto[7]>             0     1    1     ZERO
bin <auto[9],auto[7]>              0     1    1     ZERO
bin <auto[8],auto[7]>              0     1    1     ZERO
bin <auto[7],auto[7]>              0     1    1     ZERO
bin <auto[6],auto[7]>              0     1    1     ZERO
```

Sample of the generated cross bins

- Some previous examples have measured coverage of the transaction *kind*, and *port number*, but what about the two combined?
- Did you try every kind of transaction into every port?
- The *cross* construct in SystemVerilog records the combined values of two or more cover points in a group.
- The *cross* statement takes only cover points or a simple variable name.
- If you want to use expressions, hierarchical names or variables in an object such as *handle.variable*, you must first specify the expression in a cover point with a label and then use the label in the *cross* statement.
- The above code creates cover points for *tr.k* and *tr.p,* then the two points are crossed to show all combinations. System Verilog creates a total of 128 (8 × 16) bins.
- Even a simple cross can result in a very large number of bins.
- A random testbench created 30 transactions and produced the coverage report shown above.
- Note that even though 100% port values were generated, and 87.5% kind values are generated, only 21.31% of the cross combinations were seen.

Labelling Cross Coverage Bins

30

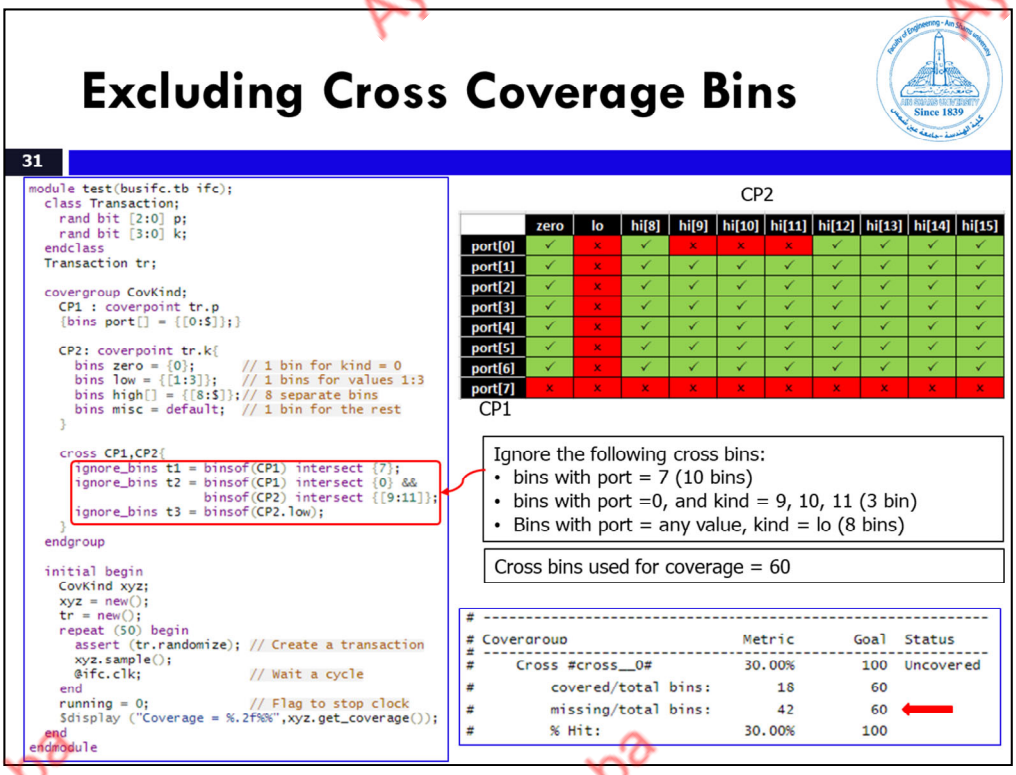
```
module test(busifc.tb ifc);
  class Transaction;
    rand bit [2:0] p;
    rand bit [3:0] k;
  endclass

  Transaction tr;

  covergroup CovKind;
    CP1 : coverpoint tr.p
    {bins port[] = {[0:$]};}

    CP2: coverpoint tr.k{
      bins zero = {0};          // 1 bin for kind = 0
      bins low = {[1:3]};       // 1 bins for values 1:3
      bins high[] = {[8:$]};    // 8 separate bins
      bins misc = default;      // 1 bin for the rest
    }

    cross CP1,CP2;
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (50) begin
      assert (tr.randomize); // Create a transaction
      xyz.sample();
      @ifc.clk;              // Wait a cycle
    end
    running = 0;             // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

	Metric	Goal	Status
bin <port[1],high[15]>	2	1	Covered
bin <port[0],high[15]>	4	1	Covered
bin <port[7],high[14]>	4	1	Covered
bin <port[6],high[14]>	4	1	Covered
bin <port[4],zero>	9	1	Covered
bin <port[3],zero>	4	1	Covered
bin <port[2],zero>	1	1	Covered
bin <port[1],zero>	6	1	Covered
bin <port[0],zero>	6	1	Covered
bin <port[4],high[13]>	0	1	ZERO
bin <port[1],high[10]>	0	1	ZERO
bin <port[0],high[8]>	0	1	ZERO

Sample of the generated cross bins

- If you want more readable cross coverage bin names. you can label the individual cover point bins, and System Verilog will use these names when creating the cross bins.
- If you define bins that contain multiple values, the coverage statistics change.
- In the generated report, the number of cross bins has dropped from 128 to 80. This is because kind has 10 bins: zero, lo, hi[8], hi[9], hi[10], hi[11], hi[12], hi[13], hi[14], hi[15]. The bin *misc* is not used in coverage calculation.

Excluding Cross Coverage Bins



- To reduce the number of bins, use *ignore_bins*. With cross coverage, you specify the cover point with *binsof* and the set of values with *intersect* so that a single *ignore_bins* construct can sweep out many individual bins.
- The first *ignore_bins* (t1) just excludes bins where *port* is 7 and any value of *kind*. Since *kind* has 10 bins, this statement excludes 10 bins.
- The second *ignore_bins* (t2) is more selective, ignoring bins where port is 0 and kind is 9, 10, or 11, for a total of 3 bins.
- The *ignore_bins* can use the bins defined in the individual cover points.
- The *ignore_bins* (t3) uses bin names to exclude *CP2.low*
- The bins must be names defined at compile-time, such as zero and low.
- The bins hi[11], hi[9], hi[10] ... hi[15], and any automatically generated bins do not have names that can be used at compile-time in other statements such as *ignore_bins*; these names are created at run-time or during the report generation.
- Note that *binsof* uses parentheses (), while *intersect* specifies a range and therefore uses curly braces {}

Generic Coverage Groups

32

- Sometimes you may have several cover groups which are very similar.
- System Verilog allows you to create a generic cover group, which can then be customized when you instantiate it.
- This is done by passing the arguments into the constructor, as will be seen in the following.

- As you start writing cover groups, you will find that some are very similar to one another.
- System Verilog allows you to create a generic cover group so that you can specify a few unique details when you instantiate it.

Passing Arguments to Cover Groups

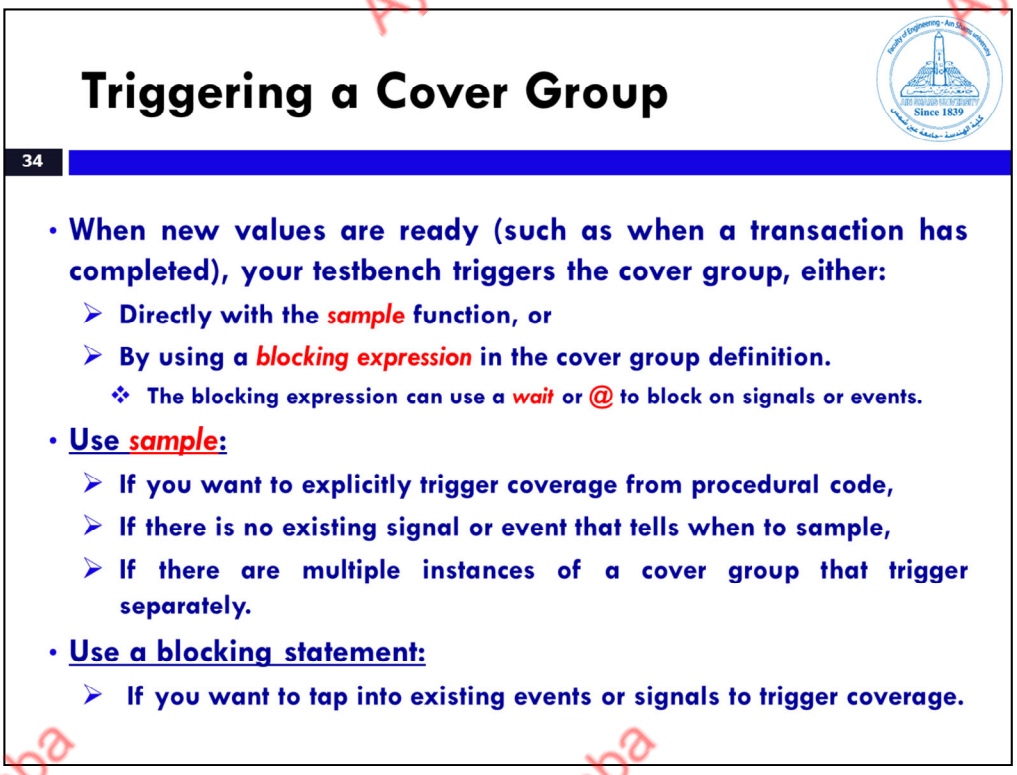
33

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit[2:0] port_a, port_b;
  endclass

  covergroup CovPort(ref bit[2:0] port, input int mid);
    coverpoint port{
      bins lo = {[0:mid-1]};
      bins hi = {[mid:$]};
    }
  endgroup

  initial begin
    Transaction tr;
    CovPort cpa, cpb;
    tr = new();
    cpa = new(tr.port_a,4); // port_a, lo=0:3, hi=4:7
    cpb = new(tr.port_b,2); // port_b, lo=0:1, hi=2:7

    repeat (5) begin        // Run a few cycles
      tr.randomize;         // Create a transaction
      cpa.sample();
      cpb.sample();
      @ifc.cb;              // Wait a cycle
    end
    running = 0;            // Flag to stop clock
  end
endmodule
```

- The cover points are passed by reference, while the other arguments are passed by value.

```
Covergroup instance cpa      100.00%   100   Covered
   covered/total bins:             2     2
   missing/total bins:             0     2
   % Hit:                    100.00%   100
   Coverpoint port           100.00%   100   Covered
      covered/total bins:          2     2
      missing/total bins:          0     2
      % Hit:                 100.00%   100
      bin lo                       1     1   Covered
      bin hi                       4     1   Covered
Covergroup instance cpb       50.00%   100   Uncovered
   covered/total bins:             1     2
   missing/total bins:             1     2
   % Hit:                     50.00%   100
   Coverpoint port            50.00%   100   Uncovered
      covered/total bins:          1     2
      missing/total bins:          1     2
      % Hit:                  50.00%   100
      bin lo                       0     1   ZERO
      bin hi                       5     1   Covered
```

- The above code shows a cover group that uses an argument to split the range into two halves. Just pass the midpoint value to the cover groups' new function.
- The cover points are passed by reference, while the other arguments are passed by value.



- The two major parts of functional coverage are the sampled data values and the time when they are sampled.
- When new values are ready (such as when a transaction has completed), your testbench triggers the cover group. This can be done directly with the *sample* function, as shown in the previous examples, or by using a *blocking expression* in the cover group definition (as will be seen in the next few slides).
- The blocking expression can use a *wait* or @ to block on signals or events.
- Use sample if you want to explicitly trigger coverage from procedural code, if there is no existing signal or event that tells when to sample, or if there are multiple instances of a cover group that trigger separately.
- Use the blocking statement in the cover group declaration if you want to tap into existing events or signals to trigger coverage.

Cover Group with an Event Trigger

35

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit [2:0] p;        //Eight port numbers
    rand bit [3:0] k;
  endclass

  Transaction tr;            // Create a handle

  covergroup CovKind @ifc.clk;
    CP1 : coverpoint tr.k;
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (20) begin          // Run some cycles
      assert (tr.randomize);   // Create a transaction
//      xyz.sample();
      @ifc.clk;                // Wait a cycle
    end
    running = 0;               // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());

  end
endmodule
```

Here the sampling occurs based on the clock edge of the clocking block.

Here we don't use sample

- In the above code the cover group CovPort is sampled when the testbench triggers the *ifc.clk* event (i.e at the positive edge of the clock)
- The advantage of using an event over calling the sample method directly is that you may be able to use an existing event (even if this event is triggered by an assertion)

Triggering on a SV Assertion

36

```
module test(busifc.tb ifc);
  event ready;
  class Transaction;
    rand bit [2:0] p;        //Eight port numbers
    rand bit [3:0] k;
  endclass

  Transaction tr;            // Create a handle

  covergroup CovKind @(ready);
    CP1 : coverpoint tr.k;
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (20) begin        // Run some cycles
      assert (tr.randomize) -> ready;
//      xyz.sample();
//      @(posedge ifc.clk);    // Wait a cycle
    end
    running = 0;             // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

Ready is an event that will be triggered by an assertion

Here the sampling occurs based on the event triggered by the assertion.

The assertion triggers the event when the randomization succeeds

- If you already have an SVA that looks for useful events like a complete transaction, you can add an event trigger to wake up the cover group.
- An identifier declared as an event data type is called a named event.
- Named event is a data type which has no storage.
- A named event can be triggered explicitly using "->".
- Event triggering occurrence can be recognized by using the event control "@".