SVA System Functions

Function	Use
$rose	Returns true if the LSB of the expression changed to 1. Otherwise, it returns false.
$fell	Returns true if the LSB of the expression changed to 0. Otherwise, it returns false.
$stable	Returns true if the value of the expression did not change. Otherwise, it returns false.
$past(expression, num_cycles)	Returns value of expression from num_cycles ago
$countones	Returns the number of 1s in an expression
$onehot	Returns true if exactly one bit is 1. if no bits or more than one bit is 1, it returns false.
$onehot0	Returns true is no bits or just 1 bit in the expression is 1
$isunknown	Returns true if any bit in the expression is 'X' or 'Z'

- T

A. Wahba

SVA Operators

151

Operator	Use
##n ##[m:n]	Delay operators - Fixed time interval and Time interval range
!, ||, &&	Boolean operators
|->	Overlapping implication
|=>	Nonoverlapping implication
not, or, and	Property operators

- T

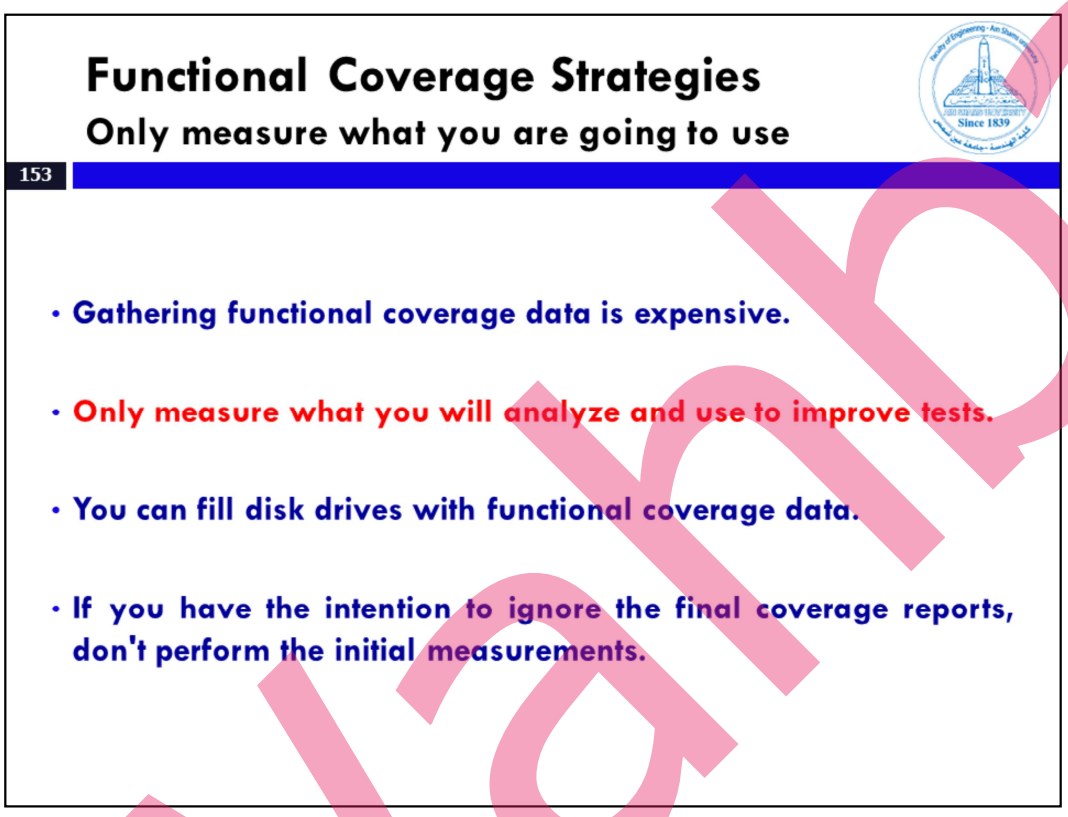
A. Wanba

SVA Repetition Operators

Operator	Use
[*n] [*m:n]	Continuous repetition operator. The expression repeats continuously for the specified range of cycles.
[->n] [->m:n]	Go to repetition operator. Indicates there's one or more delay cycles between each repetition of the expression. a[->3] is equivalent to (!a[*0:$] ##1 a) [*3]
[=n] [=m:n]	Non-consecutive implication. a[=3] is equivalent to (!a[*0:$] ##1 a ##1 !a[*0:$]) [*3]

- T

A. Wanba



- Let’s return back to functional coverage, on which we will focus more.
- Gathering functional coverage data can be expensive, and so only measure what you will analyze and use to improve your tests.
- Your simulations may run slower as the simulator monitors signals for functional coverage, but yet, this approach has lower overhead than gathering waveform traces and analyzing them.
- Once a simulation completes, the database is saved to disk. With multiple testcases and multiple random number generator seeds, you can fill disk drives with functional coverage data and reports.
- If you have the intention to ignore the coverage reports, don't perform the measurements.

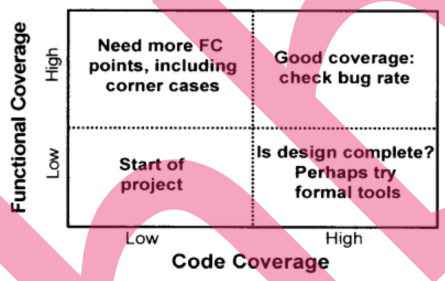
A.Wanba

Functional Coverage Strategies

Measure Completeness

154

- How can you tell if you have fully tested a design?
- Look at the coverage and consider the bug rate to see if you have reached your destination.



- At the beginning, both code and functional coverage are low.
- High functional coverage, and low code coverage:
  - Your are not exercising the full design. Update your verification plan.
- High code coverage but low functional coverage:
  - The design is missing some specs. Check the implementation.
- The goal is both high code and functional coverage.

- Your manager constantly asks you, "Are we there yet?" How can you tell if you have fully tested a design? You need to look at all coverage measurements and consider the bug rate to see if you have reached your destination.
- At the start of a project, both code and functional coverage are low.
- As you develop tests, run them over and over with different random seeds until you no longer see increasing values of functional coverage.
- Create additional constraints and tests to explore new areas. Save test/seed combinations that give high coverage, so that you can use them in regression testing.

- **What if the functional coverage is high but the code coverage is low?** Your tests are not exercising the full design, perhaps from an inadequate verification plan. It may be time to go back to the hardware specifications and update your verification plan. Then you need to add more functional coverage points to locate untested functionality.
- **A more difficult situation is high code coverage but low functional coverage.** Even though your testbench is giving the design a good workout, you are unable to put it in all the interesting states. First, see if the design implements all the specified functionality. If the functionality is there, but your tests can't reach it, you might need a formal veritication tool that can extract the design's states and create appropriate stimulus.
- The goal is both high code and functional coverage. However, don't plan your vacation yet. What is the trend of the bug rate? Are significant bugs still popping up?
- On the other hand, a low bug rate may mean that your existing strategies have run out of steam, and you should look into different approaches.

A-Wahba

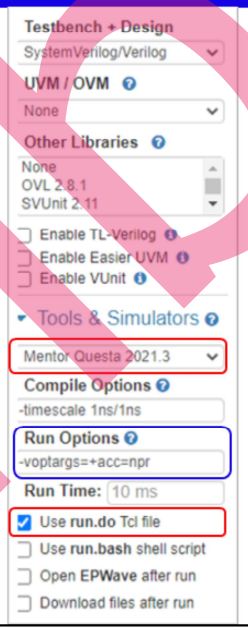
Functional Coverage Example

155

- 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
```



- T

A. Wahba

Functional Coverage Example

156

```
bit running = 1;

interface busifc(input bit clk);
  bit [31:0] data;
  bit [2:0] port;

  modport tb(output data, port, input clk);
  modport dut(input data, port);
endinterface
```

```
module bus(busifc.dut abc);
  /*
  code of the dut to be written here
  */
endmodule
```

```
module top;
  bit clk;
  initial begin
    clk = 0;
    while(running == 1)
      #5 clk = ~clk;
  end
  busifc u1(clk);
  bus u2(u1.dut);
  test u3(u1.tb);
endmodule
```

```
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;
  endgroup

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

    repeat (8) 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
```

- 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 siulator.

- **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 will talk about the differences between *modules* and *programs* later)

A-Wahba

Functional Coverage Example

157

```
# ----------------------------------------------------------------------------------------
# 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
#
```

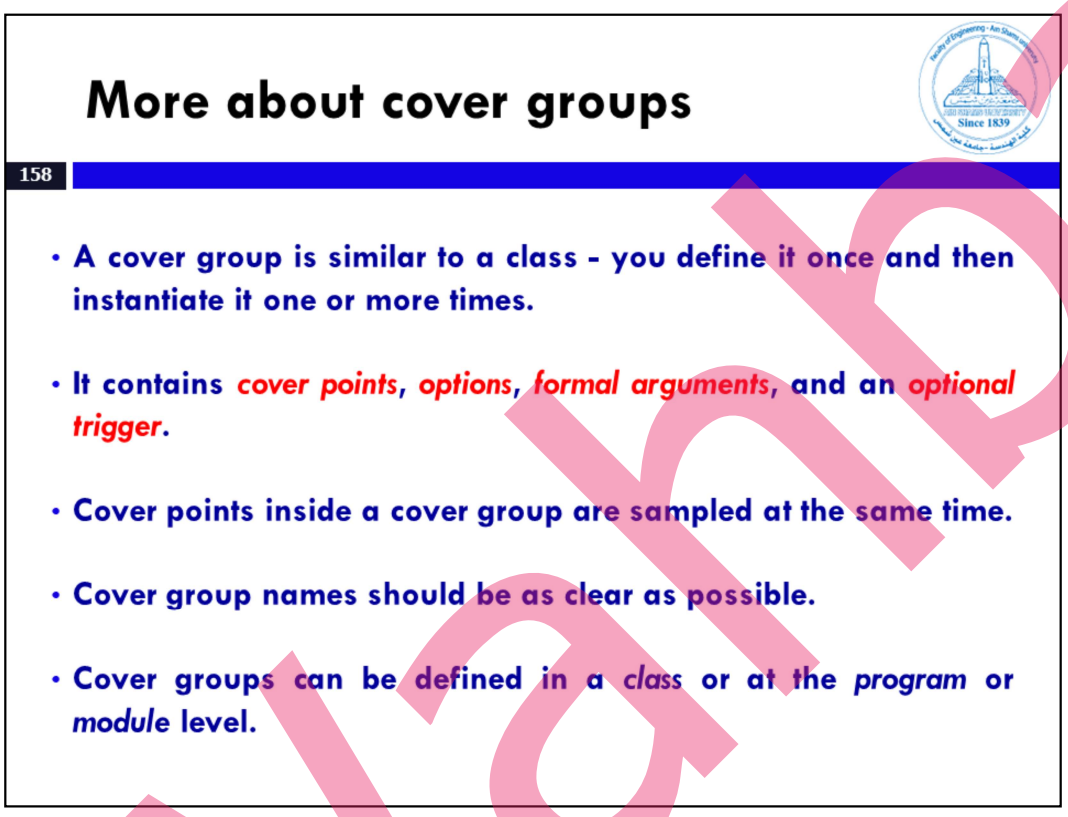
- The simulator automatically generates a bin for every value for the signal to be sampled,
- If a certain value happened, the contents for the corresponding bin will be incremented.
- Bins are the basic units of measurement for functional coverage. When you specify a variable or expression in a cover 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

A-Wahba

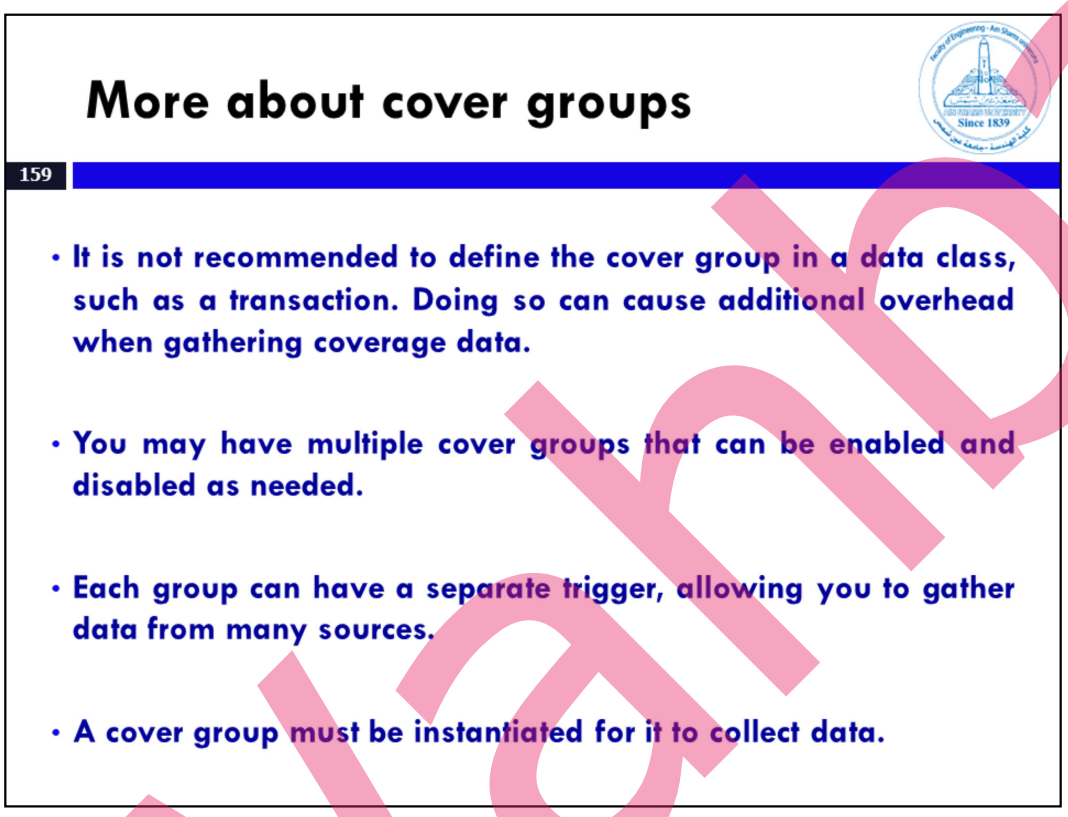
testbench is not creating the proper stimulus.

A. Wahba



- 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.
- 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-Wahba



- **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.

A.Wanda

Data Sampling

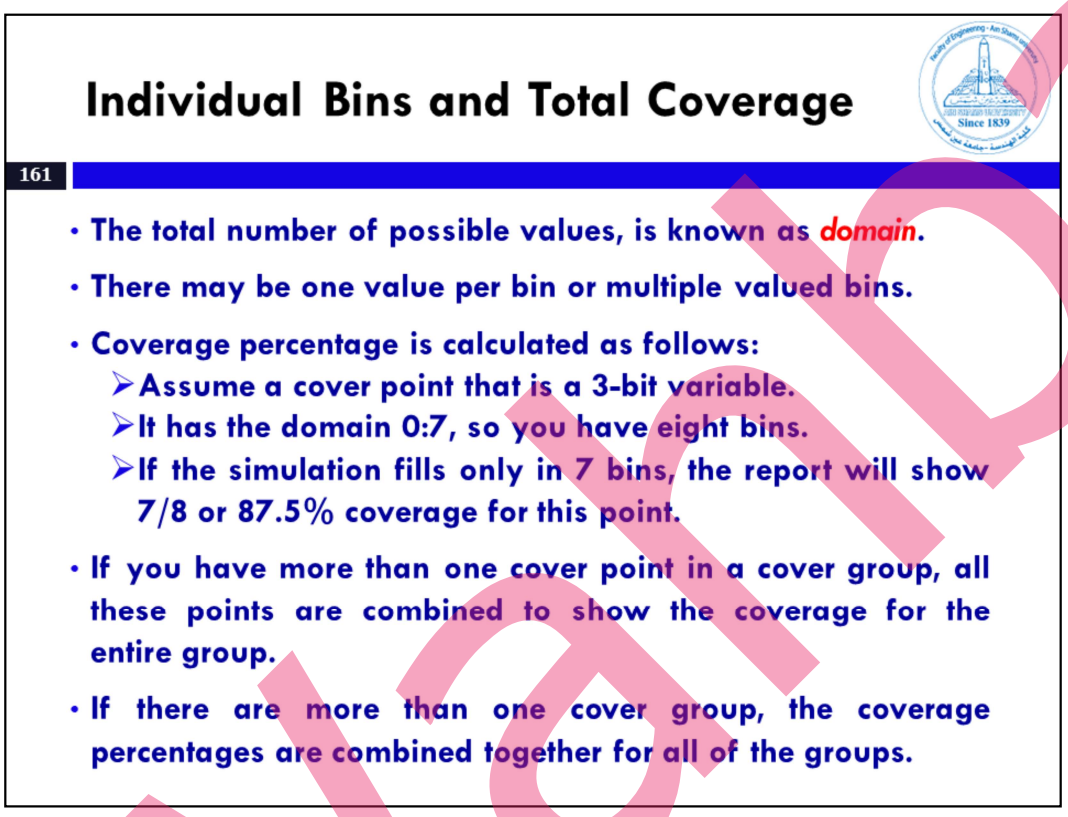
160

- 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 are then used that reads all databases and generate a report with the coverage for each part of the design and for the total coverage.

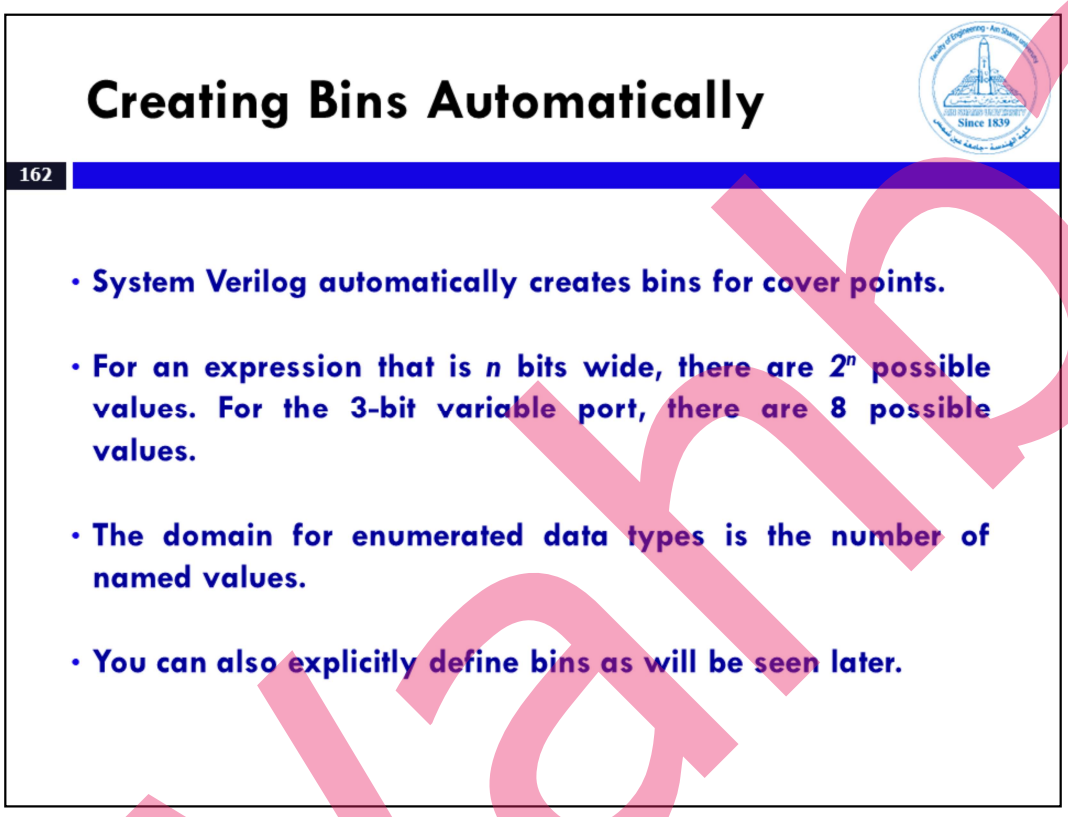
- How is coverage information gathered? When you specify a variable or expression in a cover point, SystemVeriiog creates a number of "bins" to record how many times each value has been seen.
- These bins are the basic units of measurement for functional coverage.
- If you sample a one-bit variable, 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.

A.Wahba



- 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 2, … 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.

A.Wahba



- 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.

A.Wahba

Limiting the Number of Automatic Bins Created

163

```
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 bis (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.

Alaa

Limiting the Number of Automatic Bins Created - continued

164

```
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.

A. Wahba

Sampling Expressions

165

```
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.

A-Wahba

Sampling Expressions

166

```
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.

A. Wanda

Sampling Expressions

167

```
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

A. Wahba

Sampling Expressions

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.

A. Wahba

User Defined Bins

169

```
Coverpoint CP1                100.00%        100          -    Covered
    covered/total bins:         23             23          -
    missing/total bins:         0              23          -
    % Hit:                    100.00%          100         -
    bin test_bin[0]             9              1           -    Covered
    bin test_bin[1]             18             1           -    Covered
    bin test_bin[2]             17             1           -    Covered
    bin test_bin[3]             30             1           -    Covered
    bin test_bin[4]             42             1           -    Covered
    bin test_bin[5]             46             1           -    Covered
    bin test_bin[6]             55             1           -    Covered
    bin test_bin[7]             66             1           -    Covered
    bin test_bin[8]             76             1           -    Covered
    bin test_bin[9]             56             1           -    Covered
    bin test_bin[10]            71             1           -    Covered
    bin test_bin[11]            50             1           -    Covered
    bin test_bin[12]            63             1           -    Covered
    bin test_bin[13]            53             1           -    Covered
    bin test_bin[14]            65             1           -    Covered
    bin test_bin[15]            74             1           -    Covered
    bin test_bin[16]            51             1           -    Covered
    bin test_bin[17]            47             1           -    Covered
    bin test_bin[18]            35             1           -    Covered
    bin test_bin[19]            34             1           -    Covered
    bin test_bin[20]            25             1           -    Covered
    bin test_bin[21]            12             1           -    Covered
    bin test_bin[22]            5              1           -    Covered
```

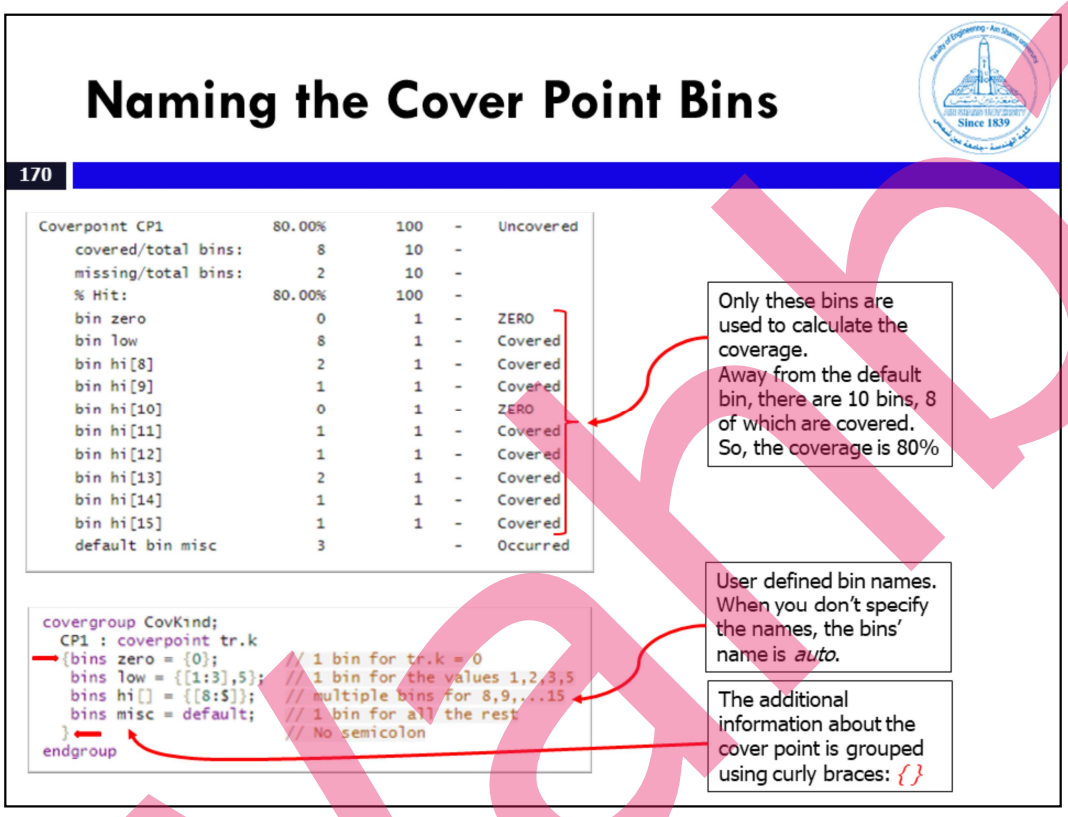
```
covergroup CovPort;
  CP1 : coverpoint (tr.p + tr.k + 5'b00000)
   {bins test_bin[] = {[0:22]};}
endgroup
```

This statement is used to specify a name and the number of bins.

Here we define 23 bins, for the values from 0 to 22

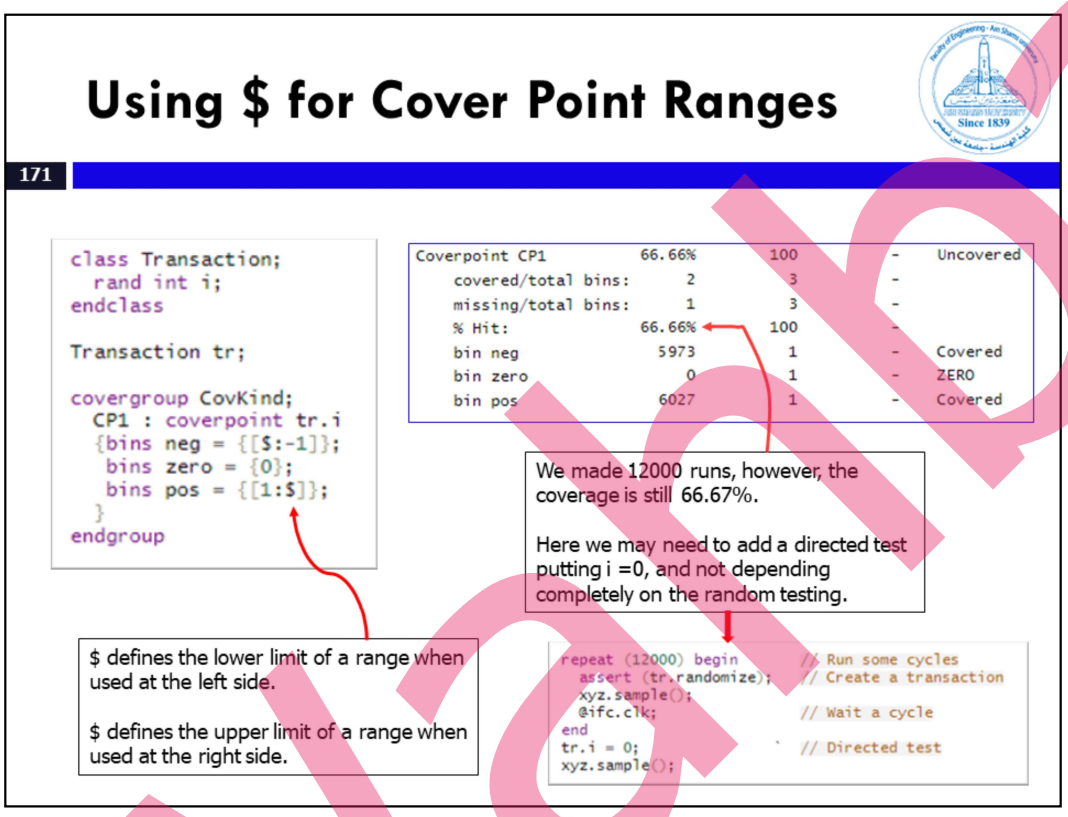
- Automatically generated bins are okay for anonymous data values, such as counter values, addresses, or values that are a power of 2.
- For other values, you should explicitly name the bins to improve accuracy and ease coverage report analysis.
- System Verilog automatically creates bin names for enumerated 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.

A.Wahba



- 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.

Amira



- 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.

Alaa

Conditional Coverage

172

```
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
```

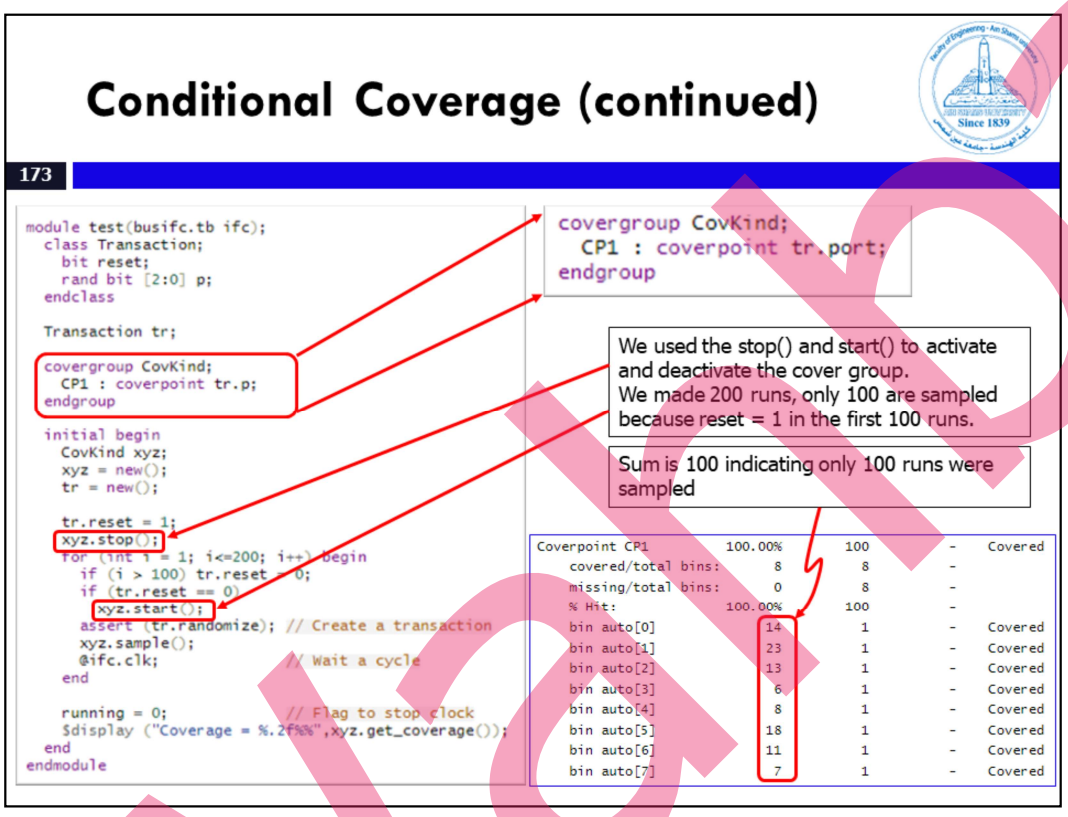
We made 200 runs, only 100 are sampled because reset = 1 in the last 100 runs.

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.

A. Wahba



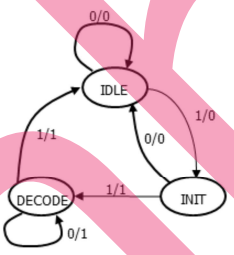
- Alternately. you can use the *start* and *stop* functions to control individual instances of cover groups, as illustrated in the above.

A. Wahba

Creating Bins for Enumerated Types

174

```
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.

A. Wahba