The American University in Cairo

CENTER OF NANOELECTRONICS & DEVICES

CND212: Digital Testing and Verification

Coverage Driven Verification (CDV)

- CDV is one of the main critical criteria for Judging the effectiveness of applied verification methodology and helps us identify what “Hasn’t been tested.”
- Classified into two main types:

    1. Code coverage
    2. Functional coverage.

- Both types of coverage metrics are essential for assessing the completeness and quality of the verification.

CENTER OF NANOELECTRONICS & DEVICES

Code Coverage

- Code coverage is a completion metric that indicates how much of the code of the Design Under Test (DUT) has been executed.
- It does not indicate that the code is correct.
- Types of RTL code coverage:
    - Statement coverage
    - Conditional coverage
    - Branch coverage
    - Path coverage
    - Toggle coverage
    - FSM coverage

Code Coverage: Statement coverage

- Measures the percentage of code statements that have been executed during the simulation.
- From N lines of code and according to the applied stimulus how many statements (lines) are covered in the simulation.
- Line coverage includes only the executable statements.
- Statements which are not executable like module, end module, comments, timescale etc are not covered.

Code Coverage: Statement coverage

EXAMPLE

```
1
2 module dut();
3 reg a,b,c,d,e,f;
4
5 initial
6 begin
7 #5 a = 0;
8 #5 a = 1;
9 end
10
11 always @(posedge a)
12 begin
13 c = b && a;
14 if(c && f)
15 b = e;
16 else
17 e = b;
18
19 case(c)
20 1:f = 1;
21 0:f = 0;
22 default : f = 0;
23 endcase
24
25 end
26 endmodule
```

There are total 12 statements at lines

5, 7, 8, 11, 13, 14, 15, 17, 19, 20, 21, 22

Covered 9 statements:

5, 7, 8, 11, 13, 14, 17, 19, 21

Uncovered 3 statements:

15, 20, 22

Statement coverage percentage: 75.00 (9/12)

Code Coverage: Conditional coverage

- Conditional coverage is the ratio of no. of cases checked to the total no. of cases present.
- Suppose one expression having Boolean expression like AND or OR, so entries which is given to that expression to the total possibilities is called expression coverage.

CENTER OF NANOELECTRONICS & DEVICES

Code Coverage: Conditional coverage

EXAMPLE

```
1
2 module dut();
3 reg a,b,c,d,e,f;
4
5 initial
6 begin
7 #5 a = 0;
8 #5 a = 1;
9 end
10
11 always @(posedge a)
12 begin
13 c = b && a;
14 if(c && f)
15 b = e;
16 else
17 e = b;
18
19 case(c)
20 1:f = 1;
21 0:f = 0;
22 default : f = 0;
23 endcase
24
25 end
26 endmodule
```

At LINE 13: Combinations of STATEMENT c = (b && a)

B = 0 and a = 0 is Covered

B = 0 and a = 1 is Covered

B = 1 and a = 0 is Not Covered

b = 1 and a = 1 is Not Covered

At LINE 14: combinations of STATEMENT if ((c && f))

C = 0 and f = 0 is Covered

C = 0 and f = 1 is Not Covered

C = 1 and f = 0 is Not Covered

C = 1 and f = 1 is Not Covered

Total possible combinations: 8

Total combinations executed: 3

Code Coverage: Branch coverage

EXAMPLE

```
1
2 module dut();
3 reg a,b,c,d,e,f;
4
5 initial
6 begin
7 #5 a = 0;
8 #5 a = 1;
9 end
10
11 always @(posedge a)
12 begin
13 c = b && a;
14 if(c && f)
15 b = e;
16 else
17 e = b;
18
19 case(c)
20 1:f = 1;
21 0:f = 0;
22 default : f = 0;
23 endcase
24
25 end
26 endmodule
```

At line 15 branch b = e; not covered

At line 17 branch e = b; covered

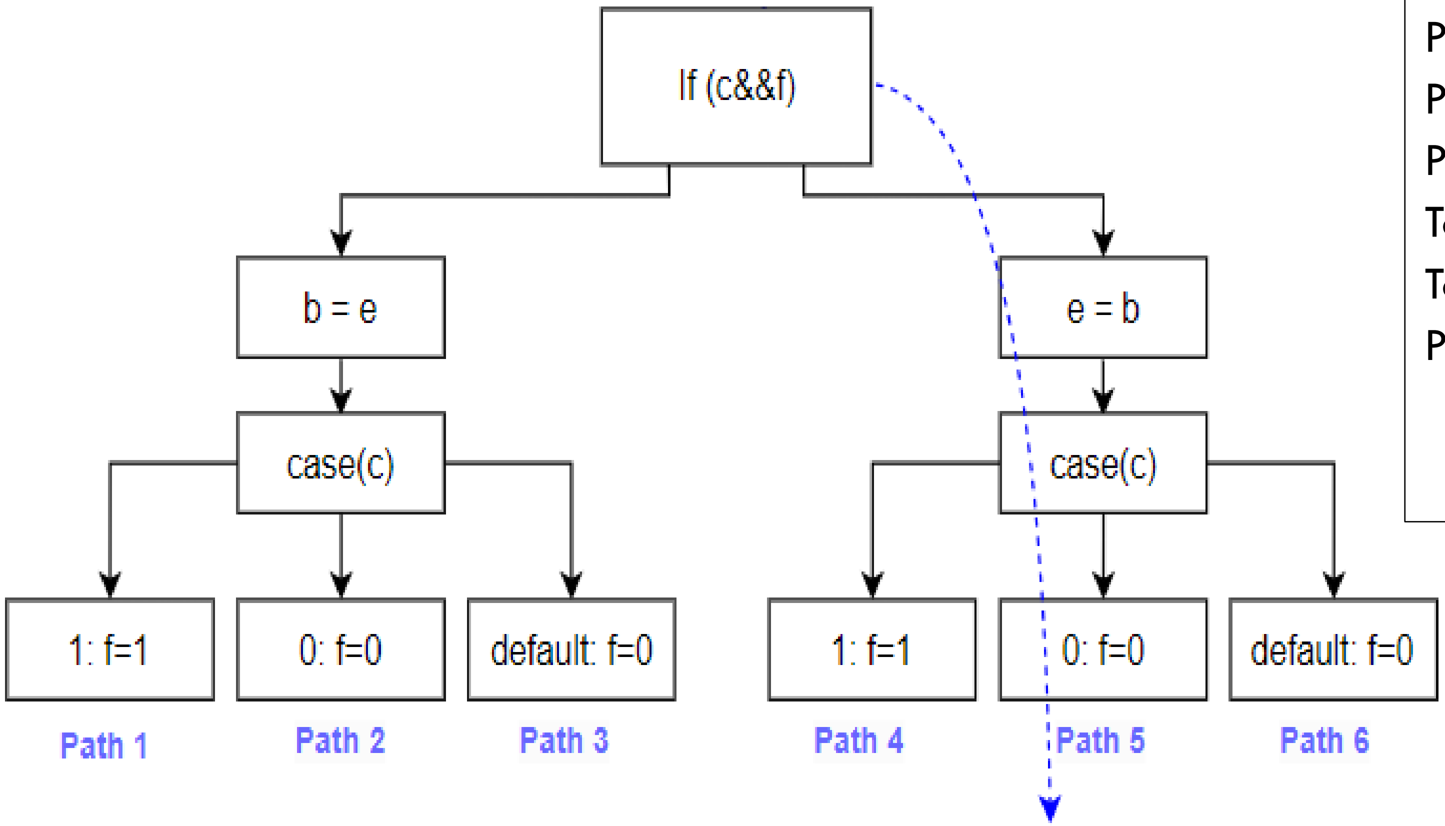
At line 20 branch 1: f = 1; not covered

At line 21 branch 0: f = 0; covered

At line 22 branch default: f = 0; not covered

Coverage percentage: 40.00 (2/5)

Code Coverage: Path coverage



Path 1 : Not Covered
Path 2 : Not Covered
Path 3: Not Covered
Path 4: Not Covered
Path 5 : Covered
Path 6 : Not Covered
Total possible paths : 6
Total covered path : 1
Path coverage Percentage : 16.67 (1/6)

Code Coverage: Toggle coverage

EXAMPLE

```

module dut();
reg a,b,c,d,e,f;

initial
begin
#5 a = 0;
#5 a = 1;
end

always @(posedge a)
begin
c = b && a;
if(c && f)
b = e;
else
e = b;

case(c)
1:f = 1;
0:f = 0;
default : f = 0;
endcase

end
endmodule
```

Name Toggled	1->0	0->1
a	No	Yes
b	No	No
c	No	No
d	No	No
e	No	No
f	No	No

Code Coverage: FSM coverage

- In this coverage we look for how many times states are visited, transited and how many sequence are covered in a Finite state machine.

- Two Types:

  1. State Coverage:

     It gives the coverage of no. of states visited over the total no. of states.

  2. Transition Coverage:

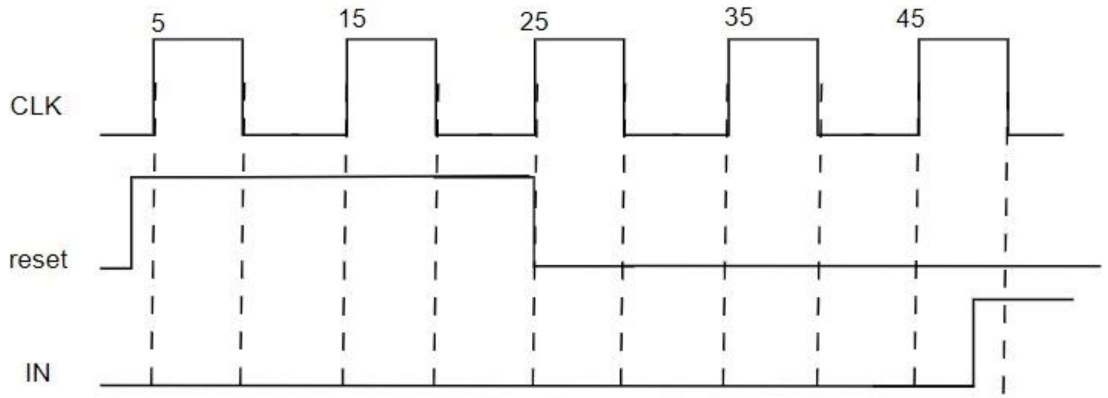
     It will count the no. of transition from one state to another and it will compare it with other total no. of transition.

Code Coverage: FSM coverage

EXAMPLE of FSM:

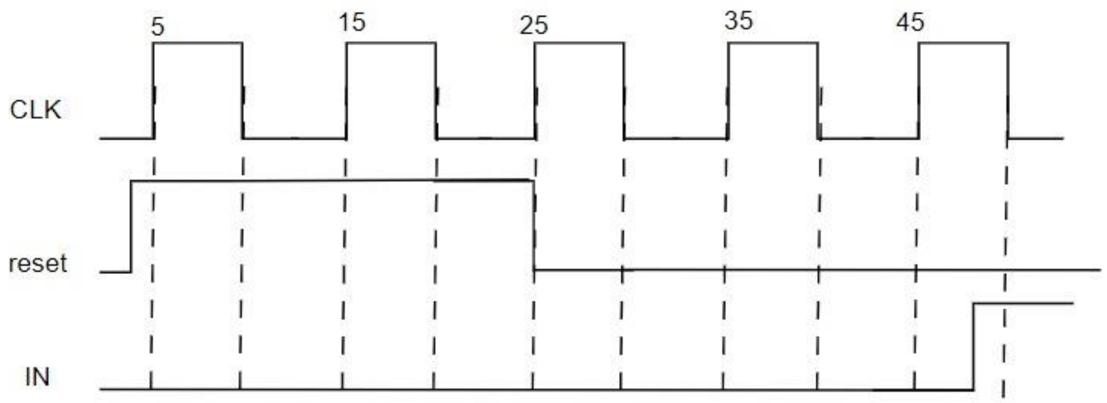
```
module fsm (clk, reset, in);
input clk, reset, in;
reg [1:0] state;

parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;

always @(posedge clk or posedge reset)
begin
if (reset) state <= s1;
else case (state)
s1:if (in == 1'b1) state <= s2;
else state <= s3;
s2: state <= s4;
s3: state <= s4;
s4: state <= s1;
endcase
end
endmodule
```

```
module testbench();
reg clk,reset,in;

fsm dut(clk,reset,in);

initial
forever #5 clk = ~clk;

initial
begin
clk =0;in = 0;
#2 reset = 0;#2 reset = 1;
#21 reset = 0;#9 in = 0;
#9 in = 1;#10 $finish;
end

endmodule
```



Code Coverage: FSM coverage

@4 → S1
@5 → S1
@15 → S1
@25 → S3
@35 → S4
@45 → S1



FSM coverage report for the above example:

// state coverage

s1 | Covered

s2 | Not Covered

s3 | Covered

s4 | Covered

// state transition

s1->s2 | Not Covered

s1->s3 | Covered

s2->s1 | Not Covered

s2->s4 | Not Covered

s3->s1 | Not Covered

s3->s4 | Covered

s4->s1 | Covered

Code Coverage: Summary

- Achieving high code coverage does not guarantee the correctness of the design or the absence of bugs.
- It is possible to have high code coverage but still have functional bugs that were not exercised during testing.
- In addition, code coverage does not take into account the design's corner cases and error handling.

Functional Coverage

- There are two types of functional coverage:

  1. Cover property

     Checks whether sequences of behaviors have occurred.

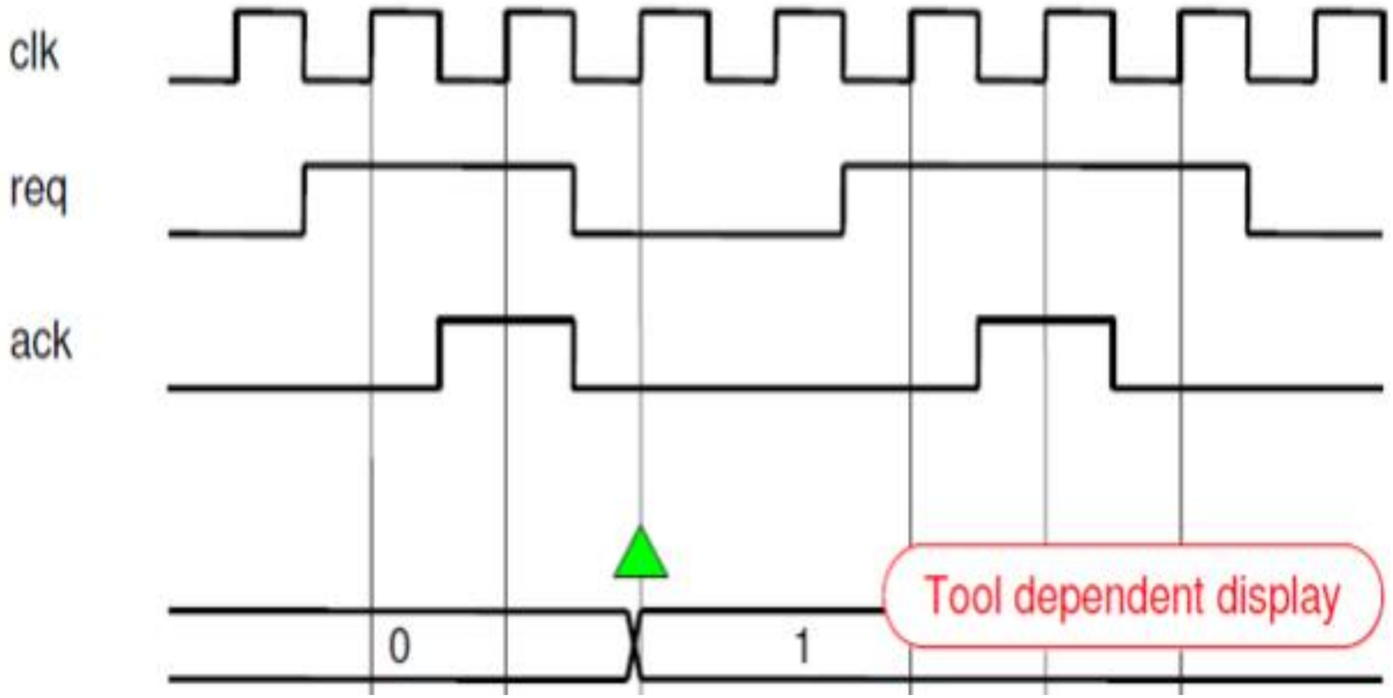
     Using SystemVerilog Assertions.

  2. Covergroup

     Checks combinations of data values have occurred. We can get Data-oriented coverage by writing Coverage groups, coverage points and also by cross coverage

Functional Coverage: Cover property

```
cover property ( @(posedge clk)
  $rose(req) |=> ((req && ack)[*0:$] ##1 !req) );
```



Functional Coverage: Covergroup

- The covergroup construct is a user-defined type. The type definition is written once, and multiple instances of that type can be created in different contexts.

- A covergroup instance can be created via the new() method.
- A covergroup can be defined in a module, program, interface, or class.
- Each covergroup specification can include the following components:
    - Clocking event
    - Coverage points
    - Cross coverage
    - Arguments(optional)
    - Coverage Options

Covergroup: Clocking event

- A clocking event that synchronizes the sampling of coverage points

```
covergroup cov_grp @(posedge clk);
    cov_p1: coverpoint a;
endgroup

cov_grp g1 = new();
```

```
covergroup cov_grp;
    cov_p1: coverpoint a;
endgroup

cov_grp g2 = new();
g2.sample();
```

Covergroup: Clocking event

- The expression within the iff construct specifies an optional condition that disables coverage for that cover point.

```
covergroup cg;
    coverpoint cp_varib iff(!reset);
endgroup
```

- In the preceding example, cover point "cp_varib" is covered only if the value reset is low.

Functional Coverage: Coverage Points

- Each coverage point is associated with "bin". On each sample clock simulator will increment the associated bin value.

- Bins can be created implicitly or explicitly

- **Implicit Bins**

  - While define cover point, if you do not specify any bins, then Implicit bins are created.
  - For an "n" bit integral coverpoint variable, a 2^n number of automatic bins will get created.
  - The number of bins creating can be controlled by auto_bin_max parameter.

Coverage Points: Implicit Bins

```
program main;
bit [0:2] y;
bit [0:2] values[$]= '{3,5,6};

covergroup cg;
cover_point_y : coverpoint y
{ option.auto_bin_max = 4 ; }
endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Coverage report:
--------------------
VARIABLE : cover_point_y
Expected : 4
Covered : 3
Percent: 75.00

Uncovered bins
------------------
auto[0:1]

Covered bins
------------------
auto[2:3]
auto[4:5]
auto[6:7]
```

Bin[0] for 0 and 1
Bin[1] for 2 and 3
Bin[2] for 4 and 5
Bin[3] for 6 and 7

Coverage Points: Implicit Bins

```
module cov;
  logic       clk;
  logic [7:0] addr;
  logic       wr_rd;

  covergroup cg @(posedge clk);
    c1: coverpoint addr;
    c2: coverpoint wr_rd;
  endgroup : cg
  cg cover_inst = new();

endmodule
```

for addr: c1.auto[0] c1.auto[1]
c1.auto[2] … c1.auto[255]

for wr_rd: c2.auto[0]

CENTER OF NANOELECTRONICS & DEVICES

Coverage Points: Explicit Bins

- Explicit bin creation is recommended method.
- Not all values are interesting or relevant in a cover point, so when the user knows the exact values he is going to cover, he can use explicit bins.
- You can also name the bins.
- “bins” keyword is used to declare the bins explicitly to a variable.
- Two types:
    1. Value Bins
    2. Transition Bins

Coverage Points: Explicit Bins

```
program main;
bit [0:2] y;
bit [0:2] values[$]= '{3,5,6};

covergroup cg;
cover_point_y : coverpoint y {
bins a = {0,1};
bins b = {2,3};
bins c = {4,5};
bins d = {6,7};
}

endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Coverage report:
--------------------
VARIABLE : cover_point_y
Expected : 4
Covered : 3
Percent: 75.00

Uncovered bins
--------------------
a

Covered bins
--------------------
b
c
d
```

Coverage Points: Explicit Bins

Array of Bins

```
program main;
bit [0:2] y;
bit [0:2] values[$]= '{3,5,6};

covergroup cg;
cover_point_y : coverpoint y {
bins a[] = {[0:7]};
}

endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Coverage report:
--------------------
VARIABLE : cover_point_y
Expected : 8
Covered : 3
Percent: 37.50

Uncovered bins
-------------------
a_0
a_1
a_2
a_4
a_7

Covered bins
-------------------
a_3
a_5
a_6
```

Coverage Points: Explicit Bins

Fixed number of bins

```
program main;
bit [0:3] y;
bit [0:2] values[$]= '{3,5,6};

covergroup cg;
cover_point_y : coverpoint y {
bins a[4] = {[0:7]};
}

endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Coverage report:
--------------------
VARIABLE : cover_point_y
Expected : 4
Covered : 3
Percent: 75.00

Uncovered bins
-------------------
a[0:1]

Covered bins
------------------
a[2:3]
a[4:5]
a[6:7]
```

Coverage Points: Explicit Bins

- SystemVerilog allows specifying one or more sets of ordered value transitions of the coverage point.

- Type of Transitions:

  - Single Value Transition
  - Sequence Of Transitions
  - Set Of Transitions
  - Consecutive Repetitions
  - Range Of Repetition
  - Goto Repetition
  - Non Consecutive Repetition

CENTER OF NANOELECTRONICS & DEVICES

Coverage Points: Explicit Bins

Single Value Transition

```
program main;
bit [0:3] y;
bit [0:2] values[$]= '{3,5,6};

covergroup cg;
cover_point_y : coverpoint y {
bins tran_34 = (3=>4);
bins tran_56 = (5=>6);
}

endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Coverage report:
--------------------
VARIABLE : cover_point_y
Expected : 2
Covered : 1
Percent: 50.00

Uncovered bins
------------------
tran_34

Covered bins
----------------
tran_56
```

Coverage Points: Explicit Bins

Sequence Of Transitions

```
program main;
bit [0:3] y;
bit [0:2] values[$]= '{3,5,6};

covergroup cg;
cover_point_y : coverpoint y {
bins tran_345 = (3=>4>=5);
bins tran_356 = (3=>5=>6);
}

endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Coverage report:
--------------------
VARIABLE : cover_point_y
Expected : 2
Covered : 1
Percent: 50.00

Uncovered bins
------------------
tran_345

Covered bins
-----------------
tran_356
```

Coverage Points: Explicit Bins

Set Of Transitions

```
program main;
bit [0:3] y;
bit [0:2] values[$]= '{3,5,6};

covergroup cg;
cover_point_y : coverpoint y {
bins trans[] = (3,4=>5,6);
}

endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Coverage report:
--------------------
VARIABLE : cover_point_y
Expected : 4
Covered : 1
Percent: 25.00

Uncovered bins
------------------
tran_34_to_56:3->6
tran_34_to_56:4->5
tran_34_to_56:4->6


Covered bins
----------------
tran_34_to_56:3->5
```

Coverage Points: Explicit Bins

Consecutive Repetitions

```
program main;
bit [0:3] y;
bit [0:2] values[$]= '{3,3,3,4,4};

covergroup cg;
cover_point_y : coverpoint y {
bins trans_3 = (3[*5]);
bins trans_4 = (4[*2]);
}

endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Coverage report:
--------------------
VARIABLE : cover_point_y
Expected : 2
Covered : 1
Percent: 50.00

Uncovered bins
------------------
trans_3

Covered bins
----------------
trans_4
```

CENTER OF NANOELECTRONICS & DEVICES

Coverage Points: Explicit Bins

Range Of Repetition

```
program main;
bit [0:3] y;
bit [0:2] values[$]= '{4,5,3,3,3,3,6,7};

covergroup cg;
cover_point_y : coverpoint y {
bins trans_3[] = (3[*3:5]);
}

endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Coverage report:
--------------------
VARIABLE : cover_point_y
Expected : 3
Covered : 1
Percent: 33.33

Uncovered bins
------------------

tran_3:3[*5]

Covered bins
----------------
tran_3:3[*4]
tran_3:3[*3]
```

Coverage Points: Explicit Bins

Goto Repetition

```
program main;
bit [0:3] y;
bit [0:2] values[$]= '{1,6,3,6,3,6,3,5};

covergroup cg;
cover_point_y : coverpoint y {
bins trans_3 = (1=>3[->3]=>5);
}

endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Coverage report:
--------------------
VARIABLE : cover_point_y
Expected : 1
Covered : 1
Percent: 100.00
```

Coverage Points: Explicit Bins

Non Consecutive Repetition

```
program main;
bit [0:3] y;
bit [0:2] values[$]= '{1,6,3,6,3,6,5};

covergroup cg;
cover_point_y : coverpoint y {
bins trans_3 = (1=>3[=2]=>5);
}

endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Coverage report:
--------------------
VARIABLE : cover_point_y
Expected : 1
Covered : 1
Percent: 100.00
```

Coverage Points: Explicit Bins

Ignore Bins

```
program main;
bit [0:2] y;
bit [0:2] values[$]= '{1,6,3,7,3,4,3,5};

covergroup cg;
cover_point_y : coverpoint y {
ignore_bins ig = {1,2,3,4,5};
}

endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Coverage report:
--------------------
VARIABLE : cover_point_y
Expected : 3
Covered : 2
Percent: 66.66

Uncovered bins
------------------
auto[0]

Excluded/Illegal bins
-------------------------
ig
auto[1]
auto[2]
auto[3]
auto[4]
auto[5]

Covered bins
----------------
auto[6]
auto[7]
```

Coverage Points: Explicit Bins

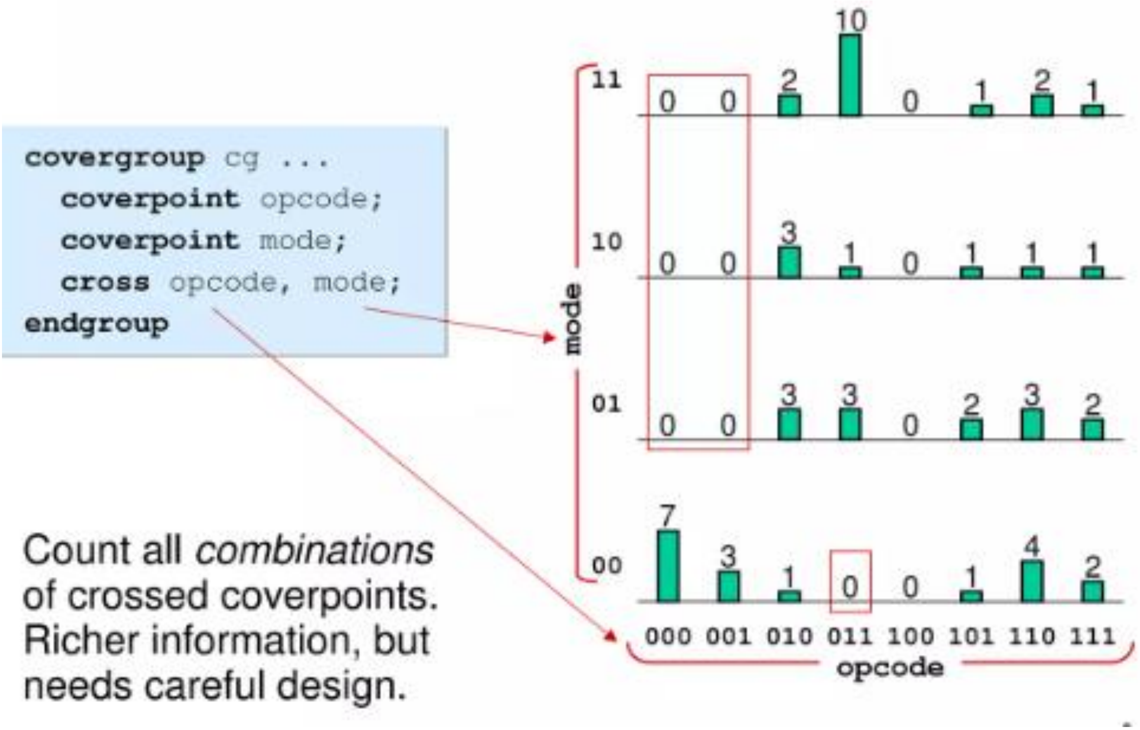
Illegal Bins:

```
program main;
bit [0:2] y;
bit [0:2] values[$]= '{1,6,3,7,3,4,3,5};

covergroup cg;
cover_point_y : coverpoint y {
illegal_bins ib = {7};
}

endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

```
Result:
------------
** ERROR **
Illegal state bin ib of coverpoint cover_point_y in
covergroup cg got hit with value 0x7
```

Functional Coverage: Cross Coverage

- Cross allows keeping track of information which is received simultaneous on more than one cover point.
- 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, SystemVerilog needs N*M cross bins to store all the combinations.



Functional Coverage: Cross Coverage

```
program main;
bit [0:1] y;
bit [0:1] y_values[$]= '{1,3};

bit [0:1] z;
bit [0:1] z_values[$]= '{1,2};

covergroup cg;
cover_point_y : coverpoint y ;
cover_point_z : coverpoint z ;
cross_yz : cross cover_point_y,cover_point_z ;
endgroup

cg cg_inst = new();
initial
foreach(y_values[i])
begin
y = y_values[i];
z = z_values[i];
cg_inst.sample();
end

endprogram
```

```
Covered bins
------------------
cover_point_y cover_point_z
auto[3] auto[2]
auto[1] auto[1]
```

Functional Coverage: Arguments

- Generic coverage groups can be written by passing their traits as arguments to the coverage constructor. This allows creating a reusable coverage group which can be used in multiple places.

```
covergroup cg (ref int v, input string comment);
  coverpoint v;

  option.per_instance = 1;
  option.weight = 5;
  option.goal = 90;
  option.comment = comment;
endgroup

int a, b;

cg cg_inst1 = new(a, "This is cg_inst1 - variable a");
cg cg_inst2 = new(b, "This is cg_inst2 - variable b");
```

Same definition - multiple uses

Functional Coverage: Options

Coverage Options			
Option	Syntax	Description	Default
Weight	weight= number	If set at the covergroup syntactic level, it specifies the weight of this covergroup instance for computing the overall instance coverage of the simulation. If set at the coverpoint syntactic level, it specifies the weight of a coverpoint for computing the instance coverage of the enclosing covergroup	1
Goal	goal=number	Specifies the target goal for a covergroup instance.	100
Name	name=string	Specifies a name for the covergroup instance.	

Functional Coverage: Options

Coverage Options			
Option	Syntax	Description	Default
Comment	comment=string	A comment that appears with a covergroup instance.	
At_least	at_least=number	Minimum number of hits for each bin.	1
Auto_bin_max	auto_bin_max=number	Maximum number of automatically created bins.	64
Per_instance	per_instance =Boolean	Each instance contributes to the overall coverage information for the covergroup type. If true, coverage information considered per each instance.	0

CENTER OF NANOELECTRONICS & DEVICES

Functional Coverage: Methods

Coverage Methods	
Method	Description
void sample()	Triggers sampling of the covergroup
real get_coverage()	Method returns the cumulative (or type) coverage, which considers the contribution of all instances of a particular coverage item, and it is a static method.
real get_inst_coverage()	Method returns the coverage of the specific instance on which it is invoked, thus, it can only be invoked via the "." operator.
void start()	Starts collecting coverage information
void stop()	Stops collecting coverage information

CENTER OF NANOELECTRONICS & DEVICES

Coverage Reports on VCS

- Home > Synopsys > Open in terminal > gedit filename.sv
- Write your code and save.
- In the terminal write the next commands:
- Compilation command:

  vcs -sverilog -cm line+cond+fsm+tgl filename.sv

- Simulation Command:

  ./simv -cm line+cond+fsm+tgl

- Command to generate Coverage report: Coverage report in html format will be in the ./urgReport directory

  urg -dir simv.vdb

- Open interactive Verdi to track coverage

  Verdi –cov –covdir simv.vdb &

Assignment

- Your task is to design a testbench for a given RTL (Register Transfer Level) code of a finite state machine and create a set of cover groups. The goal is to cover the states, state transitions, and output signals of the finite state machine.