Functional Coverage

//**//

经典功能覆盖点资料

整理：曾义和

联系：zengyihe@ict.ac.cn

//**//

Introduction

Traditionally quality of verification was measured with help of code coverage tool. Code coverage reflects how thorough the HDL code was exercised. A code coverage tool traces the code execution, usually by instrumenting. The set of features provided by code coverage tools usually includes line/block coverage, arc coverage for state machines, expression coverage, event coverage and toggle coverage.
There are some limitiation with this approach. they are.

- Overlooking non-implemented features.
- The inability to measure the interaction between multiple modules.
- The ability to measure simultaneous events or sequences of events.

Functional coverage perceives the design from a user's or a system point of view. Have you covered all of your typical scenarios? Error cases? Corner cases? Protocols? Functional coverage also allows relationships, "OK, I've covered every state in my state machine, but did I ever have an interrupt at the same time? When the input buffer was full, did I have all types of packets injected? Did I ever inject two erroneous packets in a row?"

Functional coverage was provied in Vera, Specman (E) and now in SystemVerilog. SystemVerilog functional coverage features are below.

- Coverage of variables and expressions, as well as cross coverage between them
- Automatic as well as user-defined coverage bins
- Associate bins with sets of values, transitions, or cross products
- Filtering conditions at multiple levels
- Events and sequences to automatically trigger coverage sampling
- Procedural activation and query of coverage
- Optional directives to control and regulate coverage

Functional Coverage Types

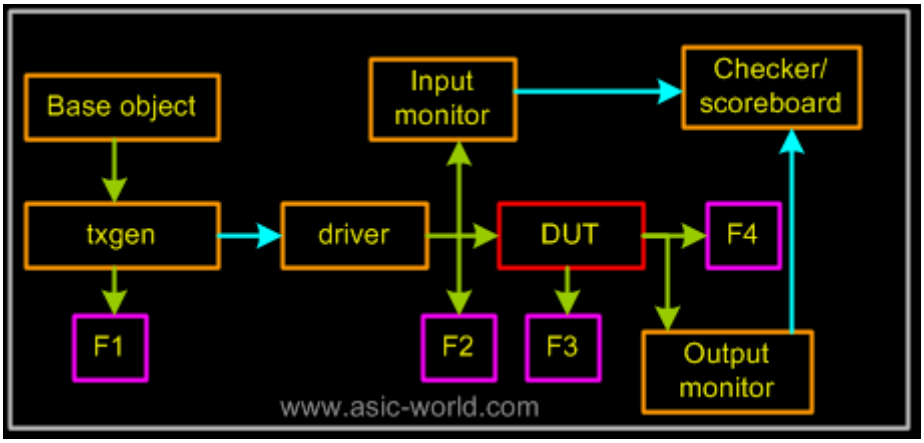
Typically when we start documenting test cases in test plan document, we also start documenting functional coverage plan. So when we start coding testbench, we code coverage along testbench. Systemverilog language

gives very good set of features that are useful in measuring coverage.

There are 4 places where functional coverage points can be coded in a verification enviroment, and they can be classfied as

- F1 : Functional coverage points are very near the randomization
- F2 : Functional coverage points are sampled at input interface of DUT
- F3 : Functional coverage points which sample internal DUT states
- F4 : Functional coverage points which sample output interface of DUT

Below figure shows all the 4 types:



- F1 Coverage points

These set of coverage points are coded in class which is instantiated very near to the randomization and it is before actual BFM/driver to DUT. There is problem with F1 type coverage points. Assume for some reason, BFM/Driver is not able to send randomizated object to DUT, even then functiona coverage gets updated. This is not acceptable.

- F2 Coverage points

These set of coverage points are coded in class which is instantiated inside a input monitor or coverage class itself will have ablitiy to sample the DUT input signals. This is perfect for having functional coverage on stimulus that DUT is being driven with,

- F3 Coverage points

These set of coverage points are coded in class which is instantiated as standalone class, and it monitors the internal states of DUT, like FSM states, or some registers. I have rarely come across these kind of coverage points. Also with advent of SVA cover properties, I see little use of F3 functional coverage types.

- F4 Coverage points

These set of coverage points are coded in class which is instantiated inside a output monitor or coverage class itself will have ablitiy to sample the DUT output signals. This is perfect for having functional coverage on output of DUT

```
1  //+++++++++++++++++++++++++++++++++++++++++++++++++++
2  //     DUT With Coverage
3  //+++++++++++++++++++++++++++++++++++++++++++++++++++
4  module simple_coverage();
5
6  logic [7:0]    addr;
7  logic [7:0]    data;
8  logic          par;
9  logic          rw;
10 logic          en;
11 //=================================================
12 // Coverage Group
13 //=================================================
14 covergroup memory @ (posedge en);
15     address : coverpoint addr {
16         bins low       = {0,50};
17         bins med       = {51,150};
18         bins high      = {151,255};
19     }
20     parity : coverpoint    par {
21         bins even    = {0};
22         bins odd     = {1};
23     }
24     read_write : coverpoint rw {
25         bins    read    = {0};
26         bins    write = {1};
27     }
28 endgroup
29 //=================================================
30 // Instance of covergroup memory
31 //=================================================
32 memory mem = new();
33 //=================================================
34 // Task to drive values
35 //=================================================
36 task drive (input [7:0] a, input [7:0] d, input r);
37       #5    en <= 1;
```

```
38      addr   <= a;
39      rw      <= r;
40      data   <= d;
41      par     <= ^d;
42      $display ("@%2tns Address :%d data %x, rw %x, parity %x",
43          $time,a,d,r, ^d);
44       #5        en <= 0;
45      rw      <= 0;
46      data   <= 0;
47      par     <= 0;
48      addr   <= 0;
49      rw      <= 0;
50  endtask
51  //================================================
52  // Testvector generation
53  //================================================
54  initial begin
55      en = 0;
56      repeat (10) begin
57          drive ($random,$random,$random);
58      end
59       #10    $finish;
60  end
61
62  endmodule
```

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. Similar to a class, once defined, a covergroup instance can be created via the new() operator. A covergroup can be defined in a package, module, program, interface, or class.

A covergroup can contain following constructs.

- clocking event : Defines the event at which coverage points are sampled. If the clocking event is omitted, users must procedurally trigger the coverage sampling.
- coverage points : A coverage point can be a variable or an expression.
- cross coverage : Coverage group can also specify cross coverage between two or more coverage points or variables.

- coverage options : This are used to control the behaviour of the covergroup.
- Optional formal arguments : This is arguments that are passed when a instance of covergroup is created.

If a clocking event is specified, it defines the event at which coverage points are sampled. If the clocking event is omitted, users must procedurally trigger the coverage sampling. This is done via the built-in sample method.

The identifier associated with the covergroup declaration defines the name of the coverage model. A covergroup can specify an optional list of arguments. When the covergroup specifies a list of formal arguments, its instances must provide to the new operator all the actual arguments that are not defaulted. This is same as in the case of constructor of a class. Actual arguments are evaluated when the new operator is executed.

A covergroup can contain one or more coverage points. A coverage point can be a variable or an expression. Each coverage point includes a set of bins associated with its sampled values or its value transitions.

A coverage group can also specify cross coverage between two or more coverage points or variables. Any combination of more than two variables or previously declared coverage points is allowed.

A coverage group can also specify one or more options to control and regulate how coverage data are structured and collected. Coverage options can be specified for the coverage group as a whole or for specific items within the coverage group, that is, any of its coverage points or crosses.

```
1  //+++++++++++++++++++++++++++++++++++++++++++++++++++++++
2  // Define the interface with coverage
3  //+++++++++++++++++++++++++++++++++++++++++++++++++++++++
4  interface mem_if (input wire clk);
5      logic          reset;
6      logic          we;
7      logic          ce;
8      logic [7:0] datai;
9      logic [7:0] datao;
10     logic [7:0] addr;
11     //=============================================
```

```
12      // Clocking block for testbench
13      //=================================================
14      clocking cb @ (posedge clk);
15         output reset, we, ce, datai,addr;
16         input    datao;
17      endclocking
18      //=================================================
19      // Coverage Group in interface
20      //=================================================
21      covergroup memory @ (posedge ce);
22         address : coverpoint addr {
23            bins low       = {0,50};
24            bins med       = {51,150};
25            bins high      = {151,255};
26         }
27         data_in : coverpoint datai {
28            bins low       = {0,50};
29            bins med       = {51,150};
30            bins high      = {151,255};
31         }
32         data_out : coverpoint datao {
33            bins low       = {0,50};
34            bins med       = {51,150};
35            bins high      = {151,255};
36         }
37         read_write : coverpoint we {
38            bins    read    = {0};
39            bins    write = {1};
40         }
41      endgroup
42      //=================================================
43      // Instance of covergroup
44      //=================================================
45      memory mem = new();
46
47   endinterface
48   //+++++++++++++++++++++++++++++++++++++++++++++++++++++
49   //     DUT With interface
50   //+++++++++++++++++++++++++++++++++++++++++++++++++++++
51   module simple_if (mem_if mif);
52   // Memory array
53   logic [7:0] mem [0:255];
54
55   //=================================================
```

```
56 // Read logic
57 //================================================
58 always @ (posedge mif.clk)
59   if (mif.reset) mif.datao <= 0;
60   else if (mif.ce &&   ! mif.we) mif.datao <= mem[mif.addr];
61
62 //================================================
63 // Write Logic
64 //================================================
65 always @ (posedge mif.clk)
66   if (mif.ce && mif.we) mem[mif.addr] <= mif.datai;
67
68 endmodule
69
70 //++++++++++++++++++++++++++++++++++++++++++++++++
71 //    Testbench
72 //++++++++++++++++++++++++++++++++++++++++++++++++
73 module coverage_covergroup();
74
75 logic clk = 0;
76 always   #10    clk++;
77 //================================================
78 // Instianciate Interface and DUT
79 //================================================
80 mem_if miff(clk);
81 simple_if U_dut(miff);
82 //================================================
83 // Default clocking
84 //================================================
85 default clocking dclk @ (posedge clk);
86
87 endclocking
88 //================================================
89 // Test Vector generation
90 //================================================
91 initial begin
92     miff.reset        <= 1;
93     miff.ce            <= 1'b0;
94     miff.we            <= 1'b0;
95     miff.addr          <= 0;
96     miff.datai         <= 0;
97      ##1    miff.reset <= 0;
98     for (int i = 0; i < 3; i ++ ) begin
99         ##1    miff.ce    <= 1'b1;
```

```
100         miff.we          <= 1'b1;
101         miff.addr        <= i;
102         miff.datai      <= $random;
103          ##3    miff.ce    <= 1'b0;
104         $display ("@%0dns Write access address %x, data %x",
105           $time,miff.addr,miff.datai);
106       end
107       for (int i = 0; i < 3; i ++ ) begin
108          ##1    miff.ce    <= 1'b1;
109         miff.we          <= 1'b0;
110         miff.addr        <= i;
111          ##3    miff.ce    <= 1'b0;
112         $display ("@%0dns Read access address %x, data %x",
113           $time,miff.addr,miff.datao);
114       end
115        #10    $finish;
116 end
117
118 endmodule
```

CoverGroup Inside a Class

As said earlier, covergroup can be embedded inside a class, interface, or module. When emebedded inside a class, it allows to generate coverage on subset of class properties. Important difference between a covergroup in module and covergroup in class is that, it is optional to create the instance of covergroup in class. as this is kind of automatic i.e the coverage group is implicitly declared. An embedded covergroup can define a coverage model for protected and local class properties without any changes to the class data encapsulation. Class members can become coverage points or can be used in other coverage constructs, such as conditional guards or option initialization. A class can have more than one covergroup.

Below example shows all this.

```
1 //+++++++++++++++++++++++++++++++++++++++++++++++++++
2 // Define the interface with coverage
3 //+++++++++++++++++++++++++++++++++++++++++++++++++++
4 interface mem_if (input wire clk);
5     logic            reset;
6     logic            we;
7     logic            ce;
```

```
 8     logic    [7:0] datai;
 9     logic [7:0] datao;
10     logic [7:0] addr;
11     //=================================================
12     // Clocking block for testbench
13     //=================================================
14     clocking cb @ (posedge clk);
15        inout reset, we, ce, datai,addr;
16        input    datao;
17     endclocking
18     //=================================================
19     // Modport for testbench
20     //=================================================
21     modport    tb (clocking cb, input clk);
22
23 endinterface
24 //+++++++++++++++++++++++++++++++++++++++++++++++++++++
25 //      DUT With interface
26 //+++++++++++++++++++++++++++++++++++++++++++++++++++++
27 module simple_if (mem_if mif);
28 // Memory array
29 logic [7:0] mem [0:255];
30
31 //=================================================
32 // Read logic
33 //=================================================
34 always @ (posedge mif.clk)
35   if (mif.reset) mif.datao <= 0;
36   else if (mif.ce &&    ! mif.we) mif.datao <= mem[mif.addr];
37
38 //=================================================
39 // Write Logic
40 //=================================================
41 always @ (posedge mif.clk)
42   if (mif.ce && mif.we) mem[mif.addr] <= mif.datai;
43
44 endmodule
45
46 //+++++++++++++++++++++++++++++++++++++++++++++++++++++
47 //    Testbench
48 //+++++++++++++++++++++++++++++++++++++++++++++++++++++
49 module coverage_class();
50
51 logic clk = 0;
```

```
52 always   #10   clk++;
53 //================================================
54 // Instianciate Interface and DUT
55 //================================================
56 mem_if miff(clk);
57 simple_if U_dut(miff);
58 //================================================
59 // Default clocking
60 //================================================
61 default clocking dclk @ (posedge clk);
62
63 endclocking
64
65 //================================================
66 // Test Vector generation
67 //================================================
68 class   mem_driver;
69    virtual   mem_if.tb cif;
70
71    //================================================
72    // Coverage Group in class
73    //================================================
74    covergroup memory @ (negedge cif.cb.ce);
75       address : coverpoint cif.cb.addr {
76          bins low      = {0,50};
77          bins med      = {51,150};
78          bins high     = {151,255};
79       }
80     endgroup
81
82     covergroup datac @ (negedge cif.cb.ce);
83        data_in : coverpoint cif.cb.datai {
84           bins low      = {0,50};
85           bins med      = {51,150};
86           bins high     = {151,255};
87        }
88        data_out : coverpoint cif.cb.datao {
89           bins low      = {0,50};
90           bins med      = {51,150};
91           bins high     = {151,255};
92        }
93        read_write : coverpoint cif.cb.we {
94           bins   read   = {0};
95           bins   write = {1};
```

```
96          }
97      endgroup
98
99      function new (virtual mem_if.tb cif);
100         this.cif = cif;
101         this.datac = new();
102         this.memory = new();
103     endfunction
104
105     task automatic drive ();
106         cif.cb.reset        <= 1;
107         cif.cb.ce           <= 1'b0;
108         cif.cb.we           <= 1'b0;
109         cif.cb.addr         <= 0;
110         cif.cb.datai        <= 0;
111         @ (cif.cb) cif.cb.reset <= 0;
112         for (int i = 0; i < 3; i ++ ) begin
113             ##1   cif.cb.ce   <= 1'b1;
114           cif.cb.we          <= 1'b1;
115           cif.cb.addr        <= i;
116           cif.cb.datai       <= $random;
117           repeat (3) @ (cif.cb) cif.cb.ce   <= 1'b0;
118           $display ("@%0dns Write access address %0x, data %x",
119               $time,i,cif.cb.datai);
120       end
121       for (int i = 0; i < 3; i ++ ) begin
122           @ (cif.cb) cif.cb.ce   <= 1'b1;
123           cif.cb.we          <= 1'b0;
124           cif.cb.addr        <= i;
125           repeat (4) @ (cif.cb) cif.cb.ce   <= 1'b0;
126           $display ("@%0dns Read access address %0x, data %x",
127               $time,i,cif.cb.datao);
128       end
129   endtask
130
131 endclass
132
133 mem_driver driver = new(miff);
134
135 initial begin
136     driver.drive();
137      #10   $finish;
138 end
139 endmodule
```

Coverage Points

A covergroup can contain one or more coverage points. A coverage point can be an integral variable or an integral expression. Each coverage point includes a set of bins associated with its sampled values or its value transitions. The bins can be explicitly defined by the user or automatically created by SystemVerilog. A coverage point creates a hierarchical scope and can be optionally labeled. If the label is specified, then it designates the name of the coverage point. This name can be used to add this coverage point to a cross coverage specification or to access the methods of the coverage point. If the label is omitted and the coverage point is associated with a single variable, then the variable name becomes the name of the coverage point. Otherwise, an implementation can generate a name for the coverage point only for the purposes of coverage reporting, that is, generated names cannot be used within the language.

Coverage point can contain optional iff statement to disable the coverage collection, when condition is active, say reset is asserted and we don't want to collect coverage during reset.

A coverage point bin associates a name and a count with a set of values or a sequence of value transitions. If the bin designates a set of values, the count is incremented every time the coverage point matches one of the values in the set. If the bin designates a sequence of value transitions, the count is incremented every time the coverage point matches the entire sequence of value transitions.

The bins for a coverage point can be automatically created by SystemVerilog or explicitly defined using the bins construct to name each bin. If the bins are not explicitly defined, they are automatically created by SystemVerilog. The number of automatically created bins can be controlled using the auto_bin_max coverage option.

The bins construct allows creating a separate bin for each value in the given range list or a single bin for the entire range of values. To create a separate bin for each value (an array of bins), the square brackets, [], must follow the bin name. To create a fixed number of bins for a set of values, a number can be specified inside the square brackets. The open_range_list used to specify the set of values associated with a bin

shall be constant expressions, instance constants (for classes only), or non-ref arguments to the coverage group. It shall be legal to use the $ primary in an open_value_range of the form [expression : $] or [$: expression].

If a fixed number of bins is specified and that number is smaller than the specified number of values, then the possible bin values are uniformly distributed among the specified bins.

The expression within the iff construct at the end of a bin definition provides a per-bin guard condition. If the expression is false at a sampling point, the count for the bin is not incremented.

The default specification defines a bin that is associated with none of the defined value bins. The default bin catches the values of the coverage point that do not lie within any of the defined bins. However, the coverage calculation for a coverage point shall not take into account the coverage captured by the default bin. The default bin is also excluded from cross coverage The default sequence form can be used to catch all transitions (or sequences) that do not lie within any of the defined transition bins. The default sequence specification does not accept multiple transition bins (i.e., the [] notation is not allowed).

We will be covering following next few pages

- Generic coverage group
- Implicit bins creation
- Explicity bins creation
- Array of bins creation
- Default bins creation
- Transition bins creation
  - ✧ Single value transition
  - ✧ Sequence of transition
  - ✧ Set of transition
  - ✧ Consecutive repetion
  - ✧ Range of repetition
  - ✧ Goto repetition
  - ✧ Non consecutive repetition
- Wildcard bin creation

- Ignore bin creation
- Illegal bin creation

Basic example

```
1  //+++++++++++++++++++++++++++++++++++++++++++++++++++++++
2  // Define the interface with coverage
3  //+++++++++++++++++++++++++++++++++++++++++++++++++++++++
4  interface mem_if (input wire clk);
5      logic           reset;
6      logic           we;
7      logic           ce;
8      logic [7:0] datai;
9      logic [7:0] datao;
10     logic [7:0] addr;
11     //=================================================
12     // Clocking block for testbench
13     //=================================================
14     clocking cb @ (posedge clk);
15        output reset, we, ce, datai,addr;
16        input   datao;
17     endclocking
18     //=================================================
19     // Coverage Group in interface
20     //=================================================
21     covergroup memory @ (posedge ce);
22        address : coverpoint addr {
23           bins low      = {0,50};
24           bins med      = {51,150};
25           bins high     = {151,255};
26        }
27        data_in : coverpoint datai {
28           bins low      = {0,50};
29           bins med      = {51,150};
30           bins high     = {151,255};
31        }
32        data_out : coverpoint datao {
33           bins low      = {0,50};
34           bins med      = {51,150};
35           bins high     = {151,255};
36        }
37        read_write : coverpoint we {
38           bins    read   = {0};
39           bins    write = {1};
```

```
40        }
41      endgroup
42      //=============================================
43      // Instance of covergroup
44      //=============================================
45      memory mem = new();
46
47  endinterface
48  //+++++++++++++++++++++++++++++++++++++++++++++++++++
49  //      DUT With interface
50  //+++++++++++++++++++++++++++++++++++++++++++++++++++
51  module simple_if (mem_if mif);
52  // Memory array
53  logic [7:0] mem [0:255];
54
55  //=============================================
56  // Read logic
57  //=============================================
58  always @ (posedge mif.clk)
59    if (mif.reset) mif.datao <= 0;
60    else if (mif.ce &&    ! mif.we) mif.datao <= mem[mif.addr];
61
62  //=============================================
63  // Write Logic
64  //=============================================
65  always @ (posedge mif.clk)
66    if (mif.ce && mif.we) mem[mif.addr] <= mif.datai;
67
68  endmodule
69
70  //+++++++++++++++++++++++++++++++++++++++++++++++++++
71  //    Testbench
72  //+++++++++++++++++++++++++++++++++++++++++++++++++++
73  module coverage_covergroup();
74
75  logic clk = 0;
76  always    #10    clk++;
77  //=============================================
78  // Instianciate Interface and DUT
79  //=============================================
80  mem_if miff(clk);
81  simple_if U_dut(miff);
82  //=============================================
83  // Default clocking
```

```
84 //================================================
85 default clocking dclk @ (posedge clk);
86
87 endclocking
88 //================================================
89 // Test Vector generation
90 //================================================
91 initial begin
92     miff.reset        <= 1;
93     miff.ce           <= 1'b0;
94     miff.we           <= 1'b0;
95     miff.addr         <= 0;
96     miff.datai        <= 0;
97      ##1    miff.reset <= 0;
98     for (int i = 0; i < 3; i ++ ) begin
99         ##1    miff.ce    <= 1'b1;
100        miff.we          <= 1'b1;
101        miff.addr        <= i;
102        miff.datai       <= $random;
103         ##3    miff.ce    <= 1'b0;
104        $display ("@%0dns Write access address %x, data %x",
105          $time,miff.addr,miff.datai);
106    end
107    for (int i = 0; i < 3; i ++ ) begin
108        ##1    miff.ce    <= 1'b1;
109        miff.we          <= 1'b0;
110        miff.addr        <= i;
111         ##3    miff.ce    <= 1'b0;
112        $display ("@%0dns Read access address %x, data %x",
113          $time,miff.addr,miff.datao);
114    end
115     #10    $finish;
116 end
117
118 endmodule
```

Generic Coverage Group

Normally coverage groups are coded to work on known variables, Like addres, data, or response. There are times when we don't want the coverage group to be generic, so that same coverage group can be instanciated multiple times, and each instance working on its own set of variables.

Generic coverage group is created by passing variable traits as arguments, just like we do for function and task

```
1  //+++++++++++++++++++++++++++++++++++++++++++++++++++
2  // Define the interface with coverage
3  //+++++++++++++++++++++++++++++++++++++++++++++++++++
4  interface mem_if (input wire clk);
5      logic           reset;
6      logic           we;
7      logic           ce;
8      logic [7:0] datai;
9      logic [7:0] datao;
10     logic [7:0] addr;
11     //=================================================
12     // Clocking block for testbench
13     //=================================================
14     clocking cb @ (posedge clk);
15        output reset, we, ce, datai,addr;
16        input    datao;
17     endclocking
18     //=================================================
19     // Coverage Group in interface
20     //=================================================
21     covergroup address_cov (ref logic [7:0] address,
22           input int low, int high) @ (posedge ce);
23        ADDRESS : coverpoint address {
24           bins low       = {0,low};
25           bins med       = {low,high};
26        }
27     endgroup
28     //=================================================
29     // Instance of covergroup
30     //=================================================
31     address_cov acov_low    = new(addr,0,10);
32     address_cov acov_med    = new(addr,11,20);
33     address_cov acov_high = new(addr,21,30);
34
35 endinterface
36 //+++++++++++++++++++++++++++++++++++++++++++++++++++
37 //     DUT With interface
38 //+++++++++++++++++++++++++++++++++++++++++++++++++++
39 module simple_if (mem_if mif);
40 // Memory array
41 logic [7:0] mem [0:255];
42
```

```
43 //================================================
44 // Read logic
45 //================================================
46 always @ (posedge mif.clk)
47   if (mif.reset) mif.datao <= 0;
48   else if (mif.ce &&   ! mif.we) mif.datao <= mem[mif.addr];
49
50 //================================================
51 // Write Logic
52 //================================================
53 always @ (posedge mif.clk)
54   if (mif.ce && mif.we) mem[mif.addr] <= mif.datai;
55
56 endmodule
57
58 //++++++++++++++++++++++++++++++++++++++++++++++++
59 //    Testbench
60 //++++++++++++++++++++++++++++++++++++++++++++++++
61 module coverage_covergroup();
62
63 logic clk = 0;
64 always   #10    clk++;
65 //================================================
66 // Instianciate Interface and DUT
67 //================================================
68 mem_if miff(clk);
69 simple_if U_dut(miff);
70 //================================================
71 // Default clocking
72 //================================================
73 default clocking dclk @ (posedge clk);
74
75 endclocking
76 //================================================
77 // Test Vector generation
78 //================================================
79 initial begin
80     miff.reset        <= 1;
81     miff.ce            <= 1'b0;
82     miff.we            <= 1'b0;
83     miff.addr          <= 0;
84     miff.datai         <= 0;
85      ##1    miff.reset <= 0;
86     for (int i = 0; i < 3; i ++ ) begin
```

```
87          ##1    miff.ce    <= 1'b1;
88        miff.we          <= 1'b1;
89        miff.addr        <= i;
90        miff.datai      <= $random;
91          ##3    miff.ce    <= 1'b0;
92        $display ("@%0dns Write access address %x, data %x",
93            $time,miff.addr,miff.datai);
94      end
95      for (int i = 0; i < 3; i ++ ) begin
96          ##1    miff.ce    <= 1'b1;
97        miff.we          <= 1'b0;
98        miff.addr        <= i;
99          ##3    miff.ce    <= 1'b0;
100       $display ("@%0dns Read access address %x, data %x",
101           $time,miff.addr,miff.datao);
102     end
103      #10    $finish;
104 end
105
106 endmodule
```

Implicit bins creation

In any covergroup definition, bins needs to be defined, when they are not defined, simulator infers the coverage bins automatically, these bins are called implicit bins.

```
1  module test();
2
3  logic [7:0] addr;
4  reg ce;
5
6  covergroup address_cov () @ (posedge ce);
7      ADDRESS : coverpoint addr {
8          // Set this option to limit number of auto bins created
9          option.auto_bin_max = 10;
10        // See no bins are declared here, Not a good idea
11     }
12 endgroup
13
14 address_cov my_cov = new();
15
16 initial begin
17     ce      <= 0;
18     addr <= 0;
```

```
19      $monitor("ce %b addr 8'h%x",ce,addr);
20      repeat (10) begin
21          addr = $random();
22          ce <= 1;
23           #10 ;
24          ce <= 0;
25           #10 ;
26      end
27  end
28
29  endmodule
```

Explicity bins creation

In the last page we saw, how compile creates auto bins, There are mutiple problems with that aproach. So it recommended that user specify all the coverage bins manually.

```
 1  module test();
 2
 3  logic [7:0] addr;
 4  reg ce;
 5
 6  covergroup address_cov () @ (posedge ce);
 7      ADDRESS : coverpoint addr {
 8          // Bins are explicity declared, This is prefered way
 9          bins low                = {0,10};
10          bins med                = {11,20};
11          bins value_255      = {255};
12      }
13  endgroup
14
15  address_cov my_cov = new();
16
17  initial begin
18      ce      <= 0;
19      addr <= 0;
20      $monitor("ce %b addr 8'h%x",ce,addr);
21      repeat (10) begin
22          addr = $random();
23          ce <= 1;
24           #10 ;
25          ce <= 0;
26           #10 ;
27      end
```

```
28 end
29
30 endmodule
```

Array of bins creation

Systemverilog provides syntax to create array of bins in automatic way, but in explicity way for values

```
1  module test();
2
3  logic [7:0] addr;
4  reg ce;
5
6  covergroup address_cov () @ (posedge ce);
7      ADDRESS : coverpoint addr {
8          // This should create 11 bins
9          bins low[]                  = {[0:10]};
10         // This should create 10 bins
11         bins med[]                  = {[11:20]};
12     }
13 endgroup
14
15 address_cov my_cov = new();
16
17 initial begin
18     ce      <= 0;
19     addr <= 0;
20     $monitor("ce %b addr 8'h%x",ce,addr);
21     repeat (10) begin
22         addr = $random();
23         ce <= 1;
24          #10 ;
25         ce <= 0;
26          #10 ;
27     end
28 end
29
30 endmodule
```

Default bins creation

Like default branch in a case statement, default bin captures all the values that are not covered by implicity bin definition. Coverage values hit in default bin are not taken account while reporting coverage.

```
1  module test();
2
3  logic [7:0] addr;
4  reg ce;
5
6  covergroup address_cov () @ (posedge ce);
7      ADDRESS : coverpoint addr {
8          bins low                = {0,10};
9          // All other values are not counted in coverage calculation
10         bins lazy               = default;
11     }
12 endgroup
13
14 address_cov my_cov = new();
15
16 initial begin
17     ce      <= 0;
18     addr <= 0;
19     $monitor("ce %b addr 8'h%x",ce,addr);
20     repeat (10) begin
21         addr = $random();
22         ce <= 1;
23          #10 ;
24         ce <= 0;
25          #10 ;
26     end
27 end
28
29 endmodule
```

Transition bins creation

Transition coverage is used for checking if required transition happened, it is also used for checking if legal/illegal transition of values happened.

In any coverage plan, it is very important that importance in given to transition coverage. In any testbench, which does not implement transition coverage, it is at best a poorly done functional coverage. Transition coverage has got ability to create scenarios which can not be captured by RTL coverage.

Some of the examples of a transition coverage are

- Write followed by read to same address of memory

- Jump instruction execution after a zero flag test in CPU
- High priority frames followed by low priorit frames
- cache miss followed by cache hit for a cache controller

Note : It is very important that transition coverage be studied in detail and applied all verification env.

Transition bins creation : Sequence

This is basic type of transition coverage bin. In this bin are created for one value transition to another value.

WRITE => READ;

WRITE => READ => WRITE;

```
1  module test();
2
3  logic [7:0] addr;
4  reg ce;
5
6  covergroup address_cov () @ (posedge ce);
7      ADDRESS : coverpoint addr {
8          // simple transition bin
9          bins adr_0_to_1              = (0=>1);
10         bins adr_1_to_0              = (1=>0);
11         bins adr_1_to_2              = (1=>2);
12         bins adr_2_to_1              = (1=>0);
13         bins adr_0_1_2_3             = (0=>1=>2=>3);
14         bins adr_1_4_7               = (1=>4=>7);
15     }
16 endgroup
17
18 address_cov my_cov = new();
19
20 initial begin
21     ce      <= 0;
22     addr <= 0;
23     $monitor("ce %b addr 8'h%x",ce,addr);
24     repeat (10) begin
25         ce <= 1;
26          #10 ;
27         ce <= 0;
28         addr ++;
29          #10 ;
30     end
31 end
```

```
32
33 endmodule
```

Transition bins creation : Default sequence

Like default branch in a case statement, default sequence bin captures all the values that are not covered by implicity bin definition. Coverage values hit in default bin are not taken account while reporting coverage.

```
1  module test();
2
3  logic [7:0] addr;
4  reg ce;
5
6  covergroup address_cov () @ (posedge ce);
7      ADDRESS : coverpoint addr {
8          // simple transition bin
9          bins adr_0_to_1              = (0=>1);
10         bins adr_1_to_0              = (1=>0);
11         bins adr_1_to_2              = (1=>2);
12         bins adr_2_to_1              = (1=>0);
13         bins allother                = default sequence;
14     }
15 endgroup
16
17 address_cov my_cov = new();
18
19 initial begin
20     ce      <= 0;
21     addr <= 0;
22     $monitor("ce %b addr 8'h%x",ce,addr);
23     repeat (10) begin
24         ce <= 1;
25          #10 ;
26         ce <= 0;
27         addr ++;
28          #10 ;
29     end
30 end
31
32 endmodule
```

Transition bins creation : Set of transition

Like in the case of normal array of bin creation, Systemverilog provides syntax to declare array of transitions. This is called set of transition

```
1  module test();
2
3  logic [2:0] addr;
4  reg ce;
5
6  covergroup address_cov () @ (posedge ce);
7      ADDRESS : coverpoint addr {
8          // simple transition bin
9          bins adr_low[]              = (0,1=>2,3);
10         bins adr_med[]              = (1,2=>3,4);
11         bins adr_high[]             = (3,4=>5,6);
12     }
13 endgroup
14
15 address_cov my_cov = new();
16
17 initial begin
18     ce      <= 0;
19     addr <= 0;
20     $monitor("ce %b addr 8'h%x",ce,addr);
21     repeat (10) begin
22         ce <= 1;
23         addr <= $random;
24          #10 ;
25         ce <= 0;
26          #10 ;
27     end
28 end
29
30 endmodule
```

Transition bins creation : Consecutive repetion

Sometime it is required to check if same values repeated N number of time. This can be done by manually typing the value N times as in below.

WRITE=>WRITE=>WRITE=>WRITE

It works, but the problem is if you have to measure long sequence, then it is time waste. Systemverilog provides syntax to do it.

4[*WRITE]

```
 1 module test();
 2
 3 logic [2:0] addr;
 4 reg ce;
 5
 6 covergroup address_cov () @ (posedge ce);
 7     ADDRESS : coverpoint addr {
 8         bins adr_0_2times              = (0[*2]);
 9         bins adr_1_3times              = (1[*3]);
10         bins adr_2_4times              = (2[*4]);
11     }
12 endgroup
13
14 address_cov my_cov = new();
15
16 initial begin
17     ce      <= 0;
18     addr <= 2;
19     $monitor("ce %b addr 8'h%x",ce,addr);
20     repeat (10) begin
21         ce <= 1;
22          #10 ;
23         ce <= 0;
24          #10 ;
25     end
26 end
27
28 endmodule
```

Transition bins creation : Range of repetition

Sometime it is required to cover atleat one of the these repetition of values in transition bins are hit. We can manually specify them, but then Systemverilog provides the synax to write this.

WRITE=>WRITE or

WRITE=>WRITE=>WRITE or

WRITE=>WRITE=>WRITE=>WRITE or

WRITE=>WRITE=>WRITE=>WRITE=>WRITE

```
1  module test();
2
3  logic [2:0] addr;
4  reg ce;
5
6  covergroup address_cov () @ (posedge ce);
7      ADDRESS : coverpoint addr {
8          bins adr0[]              = (0[*1:4]);
9          bins adr1[]              = (1[*1:2]);
10     }
11 endgroup
12
13 address_cov my_cov = new();
14
15 initial begin
16     ce      <= 0;
17     $monitor("ce %b addr 8'h%x",ce,addr);
18     repeat (10) begin
19         ce <= 1;
20         addr <= $urandom_range(0,1);
21          #10 ;
22         ce <= 0;
23          #10 ;
24     end
25 end
26
27 endmodule
```

Transition bins creation : repetition with nonconsecutive

The repetition with nonconsecutive occurrence of a value is specified using: trans_item [-> repeat_range]. Here, the occurrence of a value is specified with an arbitrary number of sample points where the value does not occur.

2[->WRITE]

This is expanded as

.....=>WRITE.....=>WRITE

Here dots can be any other transition which does not contain command WRITE like IDLE, READ etc

We can mix repetition with nonconsecutive with other transition types to create complex transition patterns. Below example shows one.

```
1  module test();
2
3  logic [2:0] addr;
4  reg ce;
5
6  covergroup address_cov () @ (posedge ce);
7      ADDRESS : coverpoint addr {
8          bins adr    = (0=>2[->2]=>1);
9      }
10 endgroup
11
12 address_cov my_cov = new();
13
14 initial begin
15     ce      <= 0;
16     $monitor("ce %b addr 8'h%x",ce,addr);
17     repeat (10) begin
18         ce <= 1;
19         addr <= $urandom_range(0,2);
20          #10 ;
21         ce <= 0;
22          #10 ;
23     end
24 end
25
26 endmodule
```

Transition bins creation : Nonconsecutive repetition

Nonconsecutive repetition is where a sequence of transitions continues until the next transition.A trans_list specifies one or more sets of ordered value transitions of the coverage point. If the sequence of value transitions of the coverage point matches any complete sequence in the trans_list, the coverage count of the corresponding bin is incremented.

2[=WRITE]

....=>WRITE.....=>WRITE.....=>WRITE

```
1  module test();
2
3  logic [2:0] addr;
4  reg ce;
5
6  covergroup address_cov () @ (posedge ce);
```

```
 7     ADDRESS : coverpoint addr {
 8       bins adr    = (0=>2[=2]=>1);
 9     }
10 endgroup
11
12 address_cov my_cov = new();
13
14 initial begin
15     ce      <= 0;
16     $monitor("ce %b addr 8'h%x",ce,addr);
17     repeat (10) begin
18       ce <= 1;
19       addr <= $urandom_range(0,2);
20        #10 ;
21       ce <= 0;
22        #10 ;
23     end
24 end
25
26 endmodule
```

Wildcard bin creation

By default, a value or transition bin definition can specify 4-state values. When a bin definition includes an X or Z, it indicates that the bin count should only be incremented when the sampled value has an X or Z in the same bit positions, i.e., the comparison is done using ===. The wildcard bins definition causes all X, Z, or ? to be treated as wildcards for 0 or 1 (similar to the ==? operator).

wildcard bins abc = {2'b1?};

This will cover following values

- 2'b10
- 2'b11

transition bin :Wildcard bin can be created for transition bin, You can use X or ? based on your need.

wildcard bins abc = (2'b1x => 2'bx0};

This will cover following values

- 2'b10 => 2'b10
- 2'b10 => 2'b00
- 2'b11 => 2'b10

- 2'b11 => 2'b00

```
1  module test();
2
3  logic [2:0] addr;
4  reg ce;
5
6  covergroup address_cov () @ (posedge ce);
7      ADDRESS : coverpoint addr {
8          // Normal transition bibs
9          wildcard bins adr0    = {3'b11?};
10         // We can use wildcard in transition bins also
11         wildcard bins adr1    = (3'b1x0 => 3'bx00);
12         wildcard bins adr2    = (3'b1?0 => 3'b?00);
13     }
14 endgroup
15
16 address_cov my_cov = new();
17
18 initial begin
19     ce      <= 0;
20     $monitor("ce %b addr 8'h%x",ce,addr);
21     repeat (10) begin
22         ce <= 1;
23         addr <= $urandom_range(0,2);
24          #10 ;
25         ce <= 0;
26          #10 ;
27     end
28 end
29
30 endmodule
```

Ignore bin

A set of values or transitions associated with a coverage point can be explicitly excluded from coverage by specifying them as ignore_bins. Ignore bin syntax can be used for dynamic coverage bin disabling and can be used for modelling coverage shape syntax of VERA.

```
1  module test();
2
3  logic [2:0] addr;
4  reg ce;
```

```
 5
 6 covergroup address_cov () @ (posedge ce);
 7     ADDRESS : coverpoint addr {
 8         ignore_bins ignore_tran = (0=>2=>1);
 9         ignore_bins ignore_vals = {0,1,2,3};
10     }
11 endgroup
12
13 address_cov my_cov = new();
14
15 initial begin
16     ce      <= 0;
17     $monitor("ce %b addr 8'h%x",ce,addr);
18     repeat (10) begin
19         ce <= 1;
20         addr <= $urandom_range(0,7);
21          #10 ;
22         ce <= 0;
23          #10 ;
24     end
25 end
26
27 endmodule
```

Illegal bin creation

A set of values or transitions associated with a coverage point can be marked as illegal by specifying them as illegal_bins. Hitting a illegal bin can cause simulator to terminate simulation.

Normally illegal bin syntax should be used on coverage points on variables inside DUT or on ports which are output of DUT. Having illegal bin syntax on testbench stimulus could prevent error injection.

```
 1 module test();
 2
 3 logic [2:0] addr;
 4 reg ce;
 5
 6 covergroup address_cov () @ (posedge ce);
 7     ADDRESS : coverpoint addr {
 8         ignore_bins ignore_tran = (0=>2=>1);
 9         ignore_bins ignore_vals = {0,1,2,3};
10         illegal_bins ignore         = {5};
```

```
11      }
12  endgroup
13
14  address_cov my_cov = new();
15
16  initial begin
17      ce      <= 0;
18      $monitor("ce %b addr 8'h%x",ce,addr);
19      repeat (10) begin
20          ce <= 1;
21          addr <= $urandom_range(0,7);
22           #10 ;
23          ce <= 0;
24           #10 ;
25      end
26  end
27
28  endmodule
```

Cross Coverage

A coverage group can specify cross coverage between two or more coverage points or variables. Cross coverage is specified using the cross construct. When verifying complex systems it is important that combination of functional points are verified.

Example

- cross cmd (READ/WRITE) and addres for memory component
- cross packet type and packet length for a networking component

Auto cross coverage points

Typically cross coverage is defined like auto bin points. Syntax for doing so is as below

```
1  module test();
2
3  logic [2:0] addr;
4  logic [1:0] cmd;
5  reg ce;
6
7  covergroup address_cov () @ (posedge ce);
8      ADDRESS : coverpoint addr {
```

```
 9         bins addr0 = {0};
10         bins addr1 = {1};
11         bins addr2 = {2};
12         bins addr3 = {3};
13       }
14       CMD : coverpoint cmd {
15         bins READ = {0};
16         bins WRITE = {1};
17         bins IDLE    = {2};
18       }
19       CRS_ADDR_CMD : cross ADDRESS, CMD;
20   endgroup
21
22   address_cov my_cov = new();
23
24   initial begin
25       ce      <= 0;
26       $monitor("ce %b addr 8'h%x cmd %x",ce,addr,cmd);
27       repeat (10) begin
28         ce <= 1;
29         addr <= $urandom_range(0,5);
30         cmd    <= $urandom_range(0,2);
31          #10 ;
32         ce <= 0;
33          #10 ;
34       end
35   end
36
37   endmodule
```

User defined cross coverage points

Normally creating auto cross bin results in lot of coverage holes and it could lead to lot of fancy ignore bin syntax. So sometime user define syntax is used. User-defined bins for cross coverage are defined using binsof and intersect. To create a ignore bin on cross one needs to use binsof and intersect.

```
1  module test();
2
3  logic [2:0] addr;
4  logic [1:0] cmd;
5  reg ce;
6
```

```
 7 covergroup address_cov () @ (posedge ce);
 8     ADDRESS : coverpoint addr {
 9       bins addr0 = {0};
10       bins addr1 = {1};
11     }
12     CMD : coverpoint cmd {
13       bins READ = {0};
14       bins WRITE = {1};
15       bins IDLE    = {2};
16     }
17     CRS_USER_ADDR_CMD : cross ADDRESS, CMD {
18       bins USER_ADDR0_READ = binsof(CMD) intersect {0};
19     }
20     CRS_AUTO_ADDR_CMD : cross ADDRESS, CMD {
21       ignore_bins AUTO_ADDR_READ = binsof(CMD) intersect {0};
22       ignore_bins AUTO_ADDR_WRITE = binsof(CMD) intersect {1} &&
         binsof(ADDRESS) intersect{0};
23     }
24
25 endgroup
26
27 address_cov my_cov = new();
28
29 initial begin
30     ce      <= 0;
31     $monitor("ce %b addr 8'h%x cmd %x",ce,addr,cmd);
32     repeat (10) begin
33       ce <= 1;
34       addr <= $urandom_range(0,5);
35       cmd    <= $urandom_range(0,2);
36        #10 ;
37       ce <= 0;
38        #10 ;
39     end
40 end
41
42 endmodule
```

Coverage Options

Like in Vera, Systemverilog provides ways to control the behavior of the covergroup, coverpoint and cross. One of the most common usage of these coverage options is setting weightage of a covergroup. In a

advanced testbench there could be many covergroups, and from the verification point of view some covergroups have high priority, and they migh be is less number, on other there hand there could be covergroups which are of low priority, and they are in large number. There is no way simulator is going to know this priority information, so Systemverilog provides options to communicate this to simulator. This way, even if don't have good coverage on low priority covegroup, it won't effect overall coverage in big way.

There are two types of options

- covergroup instance specific options
- All instance specific options

Below table shows all the options available in systemverilog to control behaviour of coverage group

Options Name	Default	Description
weight=number	1	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 (or cross) syntactic level, it specifies the weight of a coverpoint (or cross) for computing the instance coverage of the enclosing covergroup.
goal=number	90	Specifies the target goal for a covergroup instance or for a coverpoint or a cross of an instance.
name=string	unique name	Specifies a name for the covergroup instance.
comment=string		A comment that appears with a covergroup instance or with a coverpoint or cross of the covergroup instance
at_least=number	1	Minimum number of times a bin needs to hit before it is declared as hit
detect_overlap=boolean	0	When true, a warning is issued if there is an overlap between the range list (or transition list) of two bins of a coverpoint.
auto_bin_max=number	64	Maximum number of automatically created bins when no bins are explicitly defined for a coverpoint.

cross_num_print_missing = number	0	Number of missing (not covered) cross product bins that must be saved to the coverage database and printed in the coverage report.
per_instance=boolean	0	Each instance contributes to the overall coverage information for the covergroup type. When true, coverage information for this covergroup instance is tracked as well.

Out of above, below list of options could improve quality of coverag convergence.

- weight=number
- goal=number
- at_least=number
- per_instance=boolean

Note : Refer to LRM if you need more information on this

```
1  module test();
2
3  logic [2:0] addr;
4  wire [2:0] addr2;
5  reg ce;
6
7  assign addr2 = addr + 1;
8
9  covergroup address_cov () @ (posedge ce);
10     option.name              = "address_cov";
11     option.comment           = "This is cool";
12     option.per_instance = 1;
13     option.goal              = 100;
14     option.weight            = 50;
15     ADDRESS : coverpoint addr {
16        option.auto_bin_max = 100;
17     }
18     ADDRESS2 : coverpoint addr2 {
19        option.auto_bin_max = 10;
20     }
21 endgroup
22
23 address_cov my_cov = new();
24
25 initial begin
26     my_cov.ADDRESS.option.at_least = 1;
```

```
27      my_cov.ADDRESS2.option.at_least = 2;
28      ce      <= 0;
29      $monitor("ce %b addr 8'h%x addr2 8'h%x",ce,addr,addr2);
30      repeat (10) begin
31        ce <= 1;
32        addr <= $urandom_range(0,7);
33         #10 ;
34        ce <= 0;
35         #10 ;
36      end
37  end
38
39  endmodule
```

Coverage Methods

Sytemverilog provides set of pre defined methods to work on covergroup. Below list of methods.

- void sample() : Triggers the sampling of covergroup
- real get_coverage() : Calculate coverage number, return value will be 0 to 100
- real get_inst_coverage() :Calculate coverage number for given instance, return value will be 0 to 100
- void set_inst_name(string) :Set name of the instance with given string
- void start() :Start collecting coverage
- void stop() :Stop collecting coverage

```
1  module test();
2
3  logic [2:0] addr;
4  wire [2:0] addr2;
5
6  assign addr2 = addr + 1;
7
8  covergroup address_cov;
9      ADDRESS : coverpoint addr {
10       option.auto_bin_max = 10;
11     }
12     ADDRESS2 : coverpoint addr2 {
13       option.auto_bin_max = 10;
14     }
15 endgroup
16
```

```
17 address_cov my_cov = new;
18
19 initial begin
20     my_cov.ADDRESS.option.at_least = 1;
21     my_cov.ADDRESS2.option.at_least = 2;
22     // start the coverage collection
23     my_cov.start();
24     // Set the coverage group name
25     my_cov.set_inst_name("ASIC-WORLD");
26     $monitor("addr 8'h%x addr2 8'h%x",addr,addr2);
27     repeat (10) begin
28        addr = $urandom_range(0,7);
29        // Sample the covergroup
30        my_cov.sample();
31         #10 ;
32     end
33     // Stop the coverage collection
34     my_cov.stop();
35     // Display the coverage
36     $display("Instance coverage is %e",my_cov.get_coverage());
37 end
38
39 endmodule
```

Coverage System Tasks

Systemverilog provides set of system tasks to help manage coverage data collection as shown below

- $set_coverage_db_name(name) : sets the filename of the coverage database into which coverage
- information is saved at the end of a simulation run.
- $load_coverage_db(name) :loads from the given filename the cumulative coverage information for all coverage group types.
- $get_coverage() :returns as a real number in the range of 0 to 100 the overall coverage of all coverage group types. This number is computed as described above.

```
1 module test();
2
3 logic [2:0] addr;
4 wire [2:0] addr2;
5
```

```
 6 assign addr2 = addr + 1;
 7
 8 covergroup address_cov;
 9     ADDRESS : coverpoint addr {
10         option.auto_bin_max = 10;
11     }
12     ADDRESS2 : coverpoint addr2 {
13         option.auto_bin_max = 10;
14     }
15 endgroup
16
17 address_cov my_cov = new;
18
19 initial begin
20     // Set the database name
21     $set_coverage_db_name("asic_world");
22     $monitor("addr 8'h%x addr2 8'h%x",addr,addr2);
23     repeat (10) begin
24         addr = $urandom_range(0,7);
25         my_cov.sample();
26          #10 ;
27     end
28     // Get the final coverage
29     $display("Total coverage %e",$get_coverage());
30 end
31
32 endmodule
```