

Randomization in system verilog refered from chip verify and verificaion guide

How is randomization done in SystemVerilog ?

Introduction :

Randomization is the process of making something random; SystemVerilog randomization is the process of generating random values to a variable. Verilog has a \$random method for generating the random integer values. This is good for randomizing the variables alone, but it is hard to use in case of class object randomization. for easy randomization of class properties, SystemVerilog provides rand keyword and randomize() method.

To enable randomization on a variable, you have to declare variables as either **rand** or **randc**. The difference between the two is that **randc** is cyclic in nature, and hence after randomization, the same value will be picked again only after all other values have been applied. If randomization succeeds, **randomize()** will return 1, else 0.

We can ensure that randomization has succeeded by using **assert()** function. This is will avoid running simulations junk values that we may not figure until we look closer.

Example:

```
class myPacket;
    // Declare two variables for randomization
    // mode is of type rand and hence any random value between 0 and 3 can be picked each time
    // key is of type randc and hence random values between 0 and 7 can be picked and
    // values will be repeated only after all other values have been already taken
    rand bit [1:0] mode;
    randc bit [2:0] key;

    // These statements are called constraints that help us to limit
    // the randomness within specified ranges
    // mode is constrained to have a value less than 3 (excluding)
    // key is constrained to have a value between 2 and 7 (excluding)
    constraint c_mode1 { mode < 3; }
    constraint c_key1 { key > 2;key < 7; }

    // This is just a function to display current values of these variables
    function display ();
        $display ("Mode : 0x%0h Key : 0x%0h", mode, key);
    endfunction
endclass

module tb_top;
    // Create a class object handle
    myPacket pkt;

    initial begin

        // Instantiate the object, and allocate memory to this variable
        pkt = new ();

        // Let's just randomize the class object 15 times and display all the
        // values randomization yielded each time
        for (int i = 0; i < 15; i++) begin
    
```

```

        // By using assert(), we are ensuring that randomization is successful.
        assert (pkt.randomize ());
        pkt.display ();
    end
end
endmodule

```

Example 2:

In the example below,

Two variables addr1 and addr2 of same bit type are declared as rand and randc respectively, observe the randomized values of addr1 and addr2.

addr1 – takes the random value on every randomization

addr2 – takes the random value on every randomization, but takes random value until every possible value has been assigned

```

//class
class packet;
    rand  bit [2:0] addr1;
    randc bit [2:0] addr2;
endclass

```

```

module rand_methods;
initial begin
    packet pkt;
    pkt = new();
repeat(10) begin
    pkt.randomize();
    $display("\taddr1 = %0d \t addr2 = %0d",pkt.addr1,pkt.addr2);
end
end
endmodule
outputs:
addr1 = 6 addr2 = 4
addr1 = 4 addr2 = 3
addr1 = 2 addr2 = 0
addr1 = 6 addr2 = 6
addr1 = 1 addr2 = 7
addr1 = 3 addr2 = 5
addr1 = 7 addr2 = 1
addr1 = 6 addr2 = 2
addr1 = 4 addr2 = 7
addr1 = 4 addr2 = 6

```

```
sample output :  
Mode : 0x0 Key : 0x3  
Mode : 0x0 Key : 0x5  
Mode : 0x0 Key : 0x4  
Mode : 0x0 Key : 0x6  
Mode : 0x2 Key : 0x4  
Mode : 0x1 Key : 0x6  
Mode : 0x2 Key : 0x5  
Mode : 0x0 Key : 0x3  
Mode : 0x1 Key : 0x3  
Mode : 0x2 Key : 0x6  
Mode : 0x0 Key : 0x4  
Mode : 0x0 Key : 0x5  
Mode : 0x0 Key : 0x5  
Mode : 0x1 Key : 0x6  
Mode : 0x0 Key : 0x3
```

Notice that randomization of Mode has resulted in repetitive values, while for Key, the values are cyclic in nature (3,4,5,6 is a complete set).

What are the different constraint styles ?

You can write constraints in a variety of ways. Constraints should not contradict each other, else randomization will fail at run-time.

Disable randomization

Introduction:

it is possible to disable the randomization of a variable by using the systemverilog randomization method rand_mode.

rand_mode method

The rand_mode() method is used to disable the randomization of a variable declared with the rand/randc keyword.

- rand_mode(1) means randomization enabled
- rand_mode(0) means randomization disabled
- The default value of rand_mode is 1, i.e enabled
- Once the randomization is disabled, it is required to make rand_mode(1) enable back the randomization
- rand_mode can be called as SystemVerilog method, the randomization enables/disable status of a variable can be obtained by calling variable.rand_mode().
- the rand_mode method returns 1 if randomization is enabled else returns 0

rand_mode syntax

```
<object_hanlde>.<variable_name>.rand_mode(enable);  
//enable = 1, randomization enable  
//enable = 0, randomization disable
```

randomization disable for a class variable

In the below example,

The class packet has random variables addr and data, randomization is disabled for a variable addr, on randomization only data will get random value. The addr will not get any random value.
rand_mode() method is called in a display to know the status.

```
class packet;  
    rand byte addr;  
    rand byte data;  
endclass  
  
module rand_methods;  
    initial begin  
        packet pkt;  
        pkt = new();  
  
        //disable rand_mode of addr variable of pkt  
        pkt.addr.rand_mode(0);  
  
        //calling randomize method  
        pkt.randomize();  
  
        $display("\taddr = %0d \t data = %0d",pkt.addr,pkt.data);  
  
        $display("\taddr.rand_mode() = %0d \t data.rand_mode() =  
%0d",pkt.addr.rand_mode(),pkt.data.rand_mode());  
    end  
endmodule  
Simulator Output  
  
addr = 0 data = 110  
addr.rand_mode() = 0 data.rand_mode() = 1
```

randomization disable for all class variable

In the below example,

randomization for all the class variable is disabled by calling obj.rand_mode(0);

```
class packet;
```

```

rand byte addr;
rand byte data;
endclass

module rand_methods;
initial begin
  packet pkt;
  pkt = new();

$display("\taddr.rand_mode() = %0d \t data.rand_mode() =
%0d",pkt.addr.rand_mode(),pkt.data.rand_mode());

//disable rand_mode of object
pkt.rand_mode(0);

//calling randomize method
pkt.randomize();

$display("\taddr = %0d \t data = %0d",pkt.addr,pkt.data);

$display("\taddr.rand_mode() = %0d \t data.rand_mode() =
%0d",pkt.addr.rand_mode(),pkt.data.rand_mode());
end
endmodule
Simulator Output

```

```

addr.rand_mode() = 1 data.rand_mode() = 1
addr = 0 data = 0
addr.rand_mode() = 0 data.rand_mode() = 0

```

Randomization Methods:

pre randomize and post randomize methods:

On calling randomize(), pre_randomize() and post_randomize() functions will get called before and after the randomize call respectively

Users can override the pre_randomize() and post_randomize() functions

In the below example,

pre and post randomized methods are implemented in the class, on calling obj.randomize() pre and post will get called.

```
class packet;
    rand bit [7:0] addr;
    randc bit [7:0] data;

    //pre randomization function
    function void pre_randomize();
        $display("Inside pre_randomize");
    endfunction

    //post randomization function
    function void post_randomize();
        $display("Inside post_randomize");
        $display("value of addr = %0d, data = %0d",addr,data);
    endfunction
endclass

module rand_methods;
    initial begin
        packet pkt;
        pkt = new();
        pkt.randomize();
    end
endmodule
```

Simulator Output

Inside pre_randomize

Inside post_randomize

value of addr = 110, data = 129

pre_randomize

the pre_randomize function can be used to set pre-conditions before the object randomization. For example, Users can implement randomization control logic in pre_randomize function. i.e randomization enable or disable by using rand_mode() method.

post_randomize

the post_randomization function can be used to check and perform post-conditions after the object randomization.

For example, Users can override the randomized values or can print the randomized values of variables.

randomization control from pre_randomize method

In the example below,

Paket has two variables, addr, and wr_rd.
assuming wr_rd = 0 read operation.
wr_rd = 1 write operation.

In order to perform write followed by reading to the same addr, randomization of addr is controlled based on the previous randomization value of wr_rd. this controlling is done in pre_randomize() function.

```
//class
class packet;
    rand bit [7:0] addr;
    randc bit      wr_rd;
    bit      tmp_wr_rd;

//pre randomization function - disabling randomization of addr,
//if the previous operation is write.
function void pre_randomize();
    if(tmp_wr_rd==1)
        addr.rand_mode(0);
    else
        addr.rand_mode(1);
endfunction

//post randomization function - store the wr_rd value to tmp_wr_rd
//and display randomized values of addr and wr_rd
function void post_randomize();
    tmp_wr_rd = wr_rd;
    $display("POST_RANDOMIZATION:: Addr = %0h,wr_rd = %0h",addr,wr_rd);
endfunction
endclass

module rand_methods;
    initial begin
        packet pkt;
        pkt = new();
    end
```

```
repeat(4) pkt.randomize();
end
endmodule
```

If the class is a derived class and no user-defined implementation of the two methods exist, then both methods will automatically call its **super** function.

Note that **pre_randomize()** and **post_randomize()** are not virtual, but behave as virtual methods. In case you try to manually make them virtual, you'll probably hit a compiler error as shown next.

```
class parent ;
function void pre_randomize();
$display("pre randomize in parent class s");
endfunction
```

```
function void post_randomize();
$display("post randomize in parent class");
endfunction
endclass
```

```
class child extends parent ;
rand byte a ;
endclass
```

```
module test;
```

```
initial begin
child h1 ;
h1 = new();
```

```

    h1.randomize();
    $display("%0d",h1.a);
end
endmodule

```

output :

```

# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: pre randomize in parent class s
# KERNEL: post randomize in parent class
# KERNEL: -39

```

```

class Beverage;
rand bit [7:0] beer_id;

```

```

virtual function void pre_randomize ();
    $display ("This will be called just before randomization");
endfunction

```

```
endclass
```

Output

```
virtual function void pre_randomize ();
```

```
|
```

ncvlog: *E,CLSMNV (testbench.sv,7|36): The pre_randomize() method cannot be declared virtual.

If randomization fails, then the variables retain their original values and are not modified

note :If **randomize()** fails, then **post_randomize()** is not called

SystemVerilog randcase

Syntax

Example

Sometimes we come across scenarios where we want the solver to randomly pick one out of the many statements. The keyword randcase introduces a case statement that randomly selects one of its branches. The case item expressions are positive integer values that represent the weights associated with each item. Probability of selecting an item is derived by the division of that item's weight divided by the sum of all weights.

Syntax

randcase

```
item   :      statement;
```

...

endcase

Example

The sum of all weights is 9, and hence the probability of taking the first branch is 1/9 or 11.11%, the probability of taking the second branch is 5/9 or 55.56% and the probability of taking the last branch is 3/9 or 33.33%.

```
module tb;
    initial begin
        for (int i = 0; i < 10; i++)
            randcase
                1      : $display ("Wt 1");
                5      : $display ("Wt 5");
                3      : $display ("Wt 3");
            endcase
    end
endmodule
```

Note that 5 appeared maximum number of times, while 1 appeared least number of times and 3 somewhere in between.

Output

```
ncsim> run
```

```
Wt 5
Wt 5
Wt 3
Wt 5
Wt 1
Wt 3
Wt 5
Wt 3
Wt 3
Wt 5
```

```
ncsim: *W,RNQUIE: Simulation is complete.
```

If a branch specifies a zero weight, then that branch is not taken.

```
module tb;
    initial begin
        for (int i = 0; i < 10; i++)
            randcase
                0      : $display ("Wt 1");
                5      : $display ("Wt 5");
                3      : $display ("Wt 3");
            endcase
    end
endmodule
```

Output

```
ncsim> run
```

```
Wt 5
```

```
Wt 5
```

```
Wt 3
```

```
Wt 5
```

```
Wt 5
```

```
Wt 3
```

```
Wt 5
```

```
Wt 3
```

```
Wt 3
```

```
Wt 5
```

```
Wt 5
```

```
Wt 3
```

```
Wt 5
```

```
ncsim: *W,RNQUIE: Simulation is complete.
```

If all randcase_items specify zero weights, even though it doesn't make any sense to do so, then no branch will be taken and might result in a run-time warning.

```
module tb;
    initial begin
        for (int i = 0; i < 10; i++)
            randcase
                0      :      $display ("Wt 1");
                0      :      $display ("Wt 5");
                0      :      $display ("Wt 3");
            endcase
        end
    endmodule
Output
```

```
ncsim> run
```

ncsim: *W,RANDNOB: The sum of the weight expressions in the randcase statement is 0.
No randcase branch was taken.

File: ./testbench.sv, line = 4, pos = 14

Scope: tb.unmblk1

Time: 0 FS + 0

ncsim: *W,RANDNOB: The sum of the weight expressions in the randcase statement is 0.
No randcase branch was taken.

File: ./testbench.sv, line = 4, pos = 14

Scope: tb.unmblk1

Time: 0 FS + 0

ncsim: *W,RANDNOB: The sum of the weight expressions in the randcase statement is 0.
No randcase branch was taken.

File: ./testbench.sv, line = 4, pos = 14

Scope: tb.unmblk1

Time: 0 FS + 0

...

Each call to randcase retrieves one random number in the range of 0 to the sum of the weights. The weights are then selected in declaration order: small random numbers correspond to the first (top) weight statements.