

# Functional Verification

## Coverage

There are two important metrics to measure coverage:

Code coverage automatically measures whether, and potentially how many times, a line, statement, block or branch has executed. More advanced code coverage can measure whether, and potentially how many times, an expression term has a chance to control the outcome of the expression. Code coverage does not automatically check transitions between design states, as the transition space is extremely large.

Functional coverage, as the name suggests, checks whether the test exercises the functionality of the design. You identify the critical combinations and sequences that should be exercised to verify the design functionality. It is important to mention that Functional coverage complements but does not replace code coverage.

SystemVerilog has two types of functional coverage:

- Covergroups for data-oriented functional coverage
- Assertions for control-oriented functional coverage

### covergroup functional coverage

```
covergroup cg @(posedge clk);  
  Addr: coverpoint addr  
  { bins low   = { [0:'h0F], 19 };  
    bins mid[] = { 16, 17, 18 };  
    bins high  = { ['h14:'hFF] }; }  
  AddrXvalid : cross Addr, valid;  
endgroup  
cg cg1 = new;
```

### Property functional coverage

```
property req_gets_gnt;  
  @(posedge clk)  
  req |-> ##[1:$] gnt;  
endproperty  
cover (req_gets_gnt);
```

## Cover-groups:

Working with the design specification, it is developed a test plan highlighting those data values, combinations of values, and sequences of values, whose occurrence provides an indication of how thoroughly the test exercises the design. Then the translation of those test plan items into the SystemVerilog data-oriented functional coverage syntax is done. An example of the definition of cover-points inside a cover-group is continued:

```
module example;  
  logic clk;  
  logic [2:0] opcode;  
  logic [7:0] address;  
  
  covergroup cg1 @(posedge clk);  
    c1: coverpoint opcode;  
    c2: coverpoint address;  
  endgroup : cg1  
  
  cg1 cover_inst = new;  
  ...  
endmodule
```

SystemVerilog by default automatically creates a single bin for every value in the coverpoint variable range. These are called automatic, or implicit, bins. For an enumerated coverpoint, there is one bin for each valid value. For an integral coverpoint variable, the number of automatic bins is at most  $2^M$  where  $M$  is the number of bits required to represent the variable. When  $2^M$  is greater than the preset limit, the values are distributed as evenly as possible, with the last bin getting any extra values. Also, it can be explicitly declared the bins enclosed in curly braces ({}), immediately after the coverpoint identifier. Note that if the coverpoint bins or options are not explicitly specified, you terminate the coverpoint declaration with a semicolon, otherwise the termination of each bin or option specification itself should be with a semicolon.

Example of explicit bins:

```

logic [4:0] var1;

ce: coverpoint var1 {
  illegal_bins a = { 0, 15 };           // 1 bin for illegal values
  ignore_bins b = { [13:15] };         // 1 bin for ignored values
  bins c = { 2, 3 };                   // 1 bin for 2, 3
  bins d[2] = { [9:11], 9, [12:15] }; // 2 bins - d[0] = {9,10}
                                     // - d[1] = {11,9,12}
  bins e[] = { [0:2], 2, 6 };           // 3 bins e[1], e[2], e[6]
  bins f = default;                   // 1 bin for 4,5,7,8
}

```

When defining a coverpoint, some options can be set in order to manage the coverage, the most useful of them are listed in the table that appears below:

Field Type	Field Ident	Default Value	cover group	cover point	cover cross	Description
string	name	<i>unique</i>	✓			Instance name
int	weight	1	✓	✓	✓	Weight of this element for calculation of covergroup or overall coverage
int	goal	90	✓	✓	✓	Target goal
string	comment	""	✓	✓	✓	Comment to include in coverage report
int	at_least	1	✓	✓	✓	Target count to consider bin as "covered"
int	auto_bin_max	64	✓	✓		Max number of auto bins
int	cross_num_print_missing	0	✓		✓	Max number of uncovered cross bins to report
bit	detect_overlap	0	✓	✓		If true issue warning for values overlapping bins
bit	per_instance	0	✓			If true track coverage for each instance

## Assertions

### Sequence syntax / Property syntax:

```
sequence name_of_sequence;    property name_of_property;
    < test expression>;        < test expression >; or
endsequence                    < complex sequence expressions >;
                                endproperty
```

### Some built-in functions / Example:

**Srose** (*boolean expression or signal\_name*)

- This returns true if LSB of signal/expression changed to 1

**Sfell** (*boolean expression or signal\_name*)

- This returns true if LSB of signal/expression changed to 0

**Sstable** (*boolean expression or signal\_name*)

- This returns true if the value of the expression did not change

```
sequence s2;
@(posedge clk) $rose(a);
endsequence
```

By defining formal arguments in a sequence definition, the same sequence can be re-used on other signals of a design that have similar behavior. Example:

```
sequence s3_lib (a, b);
    a || b;
endsequence
```

In SVA, clock cycle delays are represented by a “###”. For example, ##3 means 3 clock cycles. Remember, a sequence or a property does not do anything by itself in a simulation. They have to be asserted to take effect as shown below. Note: the sequence in the example uses the clock signal, and there are several ways of relating a check to a clock, the ones are shown here

```
sequence s5;
  @(posedge clk) a ##2 b;
endsequence

sequence s5a;
  a ##2 b;
endsequence

property p5;
  s5;
endproperty

property p5a;
  @(posedge clk) s5a;
endproperty

sequence s5b;
  a ##2 b;
endsequence

a5b : assert property(@(posedge clk) s5b);

a5 : assert property(p5); a5a : assert property(p5a);
```

A property can also be forbidden from happening. In other words, we expect the property to be false always, if the property is true, the assertion fails.

```
sequence s6;
  @(posedge clk) a ##2 b;
endsequence

property p6;
  not s6;
endproperty

a6 : assert property(p6);
```

Some fails in assertions are due to the fact that it did not get a valid starting point for the checker at a certain clock. This error can be treated as benign. In order to get a solution (vacuous success will be triggered if antecedent does not success), we should use implications. The implications constructs can only be defined inside property definitions, not in sequence definitions. There are 2 types of implications: overlapped implications and non-overlapped implications.

**Overlapped implications:** its symbol is " $\mid\rightarrow$ ", if there is a match in the antecedent, then the consequent expression is evaluated in the same clock cycle.

**Non-Overlapped implications:** its symbol is " $\mid\Rightarrow$ ", if there is a match in the antecedent, then the consequent expression is evaluated in the next clock cycle.

If a fixed delay on the consequent is added:

```
property p10;
  @(posedge clk) a |-> ##2 b;
endproperty

a10 : assert property(p10);
```

It will behave just as the examples before, the only difference is that all false errors will be removed (not encountering a valid starting point).

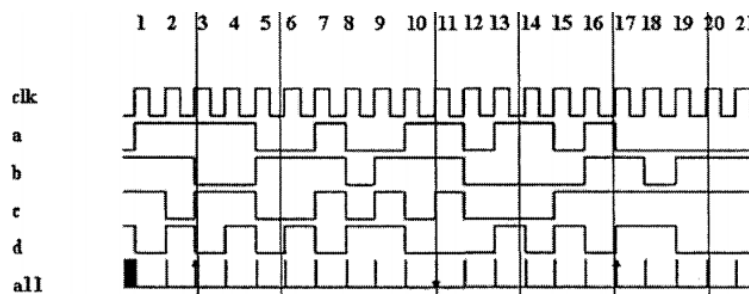
Antecedents and consequents can also be sequences:

```
sequence s11a;
  @(posedge clk) (a && b) ##1 c;
endsequence

sequence s11b;
  @(posedge clk) ##2 !d;
endsequence

property p11;
  s11a |-> s11b;
endproperty
```

Just remember that the evaluation of the second sequence will start only if the previous sequence encounters a valid starting point and get a succeeded result. Hence, the starting timing from where the second sequence will start counting the 2 clock cycle delays will be the time where the first sequence got a succeeded result. The waveform will clarify this.

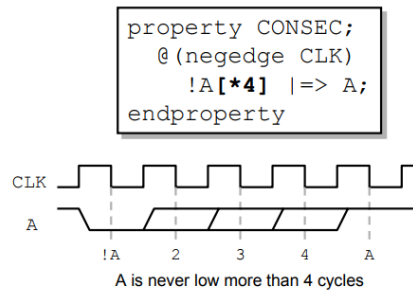


**Timing windows in SVA checkers:** a range of numbers of clock cycle delays can be specified in sequences, thus a variable number of cycle is analyzed looking for a condition to be true.

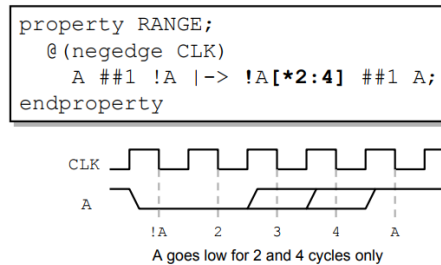
```
property p12;
  @(posedge clk) (a && b) |-> ##[1:3] c;
endproperty

a12 : assert property(p12);
```

**Consecutive sequence Repetition:** it can be specified multiple consecutive repetitions of a sequence.



Also, a range can be specified:



**Liveness assertions:**

```
property Eventually2;
  (bottom && call2 |->
    ##[1:$] top);
endproperty
```

**[1:\$]** means a range of 1 to infinite number of clock cycles. In the example: if the lift is at the bottom and the top call button is pressed the lift gets to the top floor eventually.