SystemVerilog Assertions

Type	Description

Immediate assertions	Syntax:
	[label:]assert (boolean_expresion)
	// pass block
	else
	//fail block
	
	SIGNAL_ASSERTED_ERR: assert(signal) // pass block omitted
	else $error("Assertion failed");

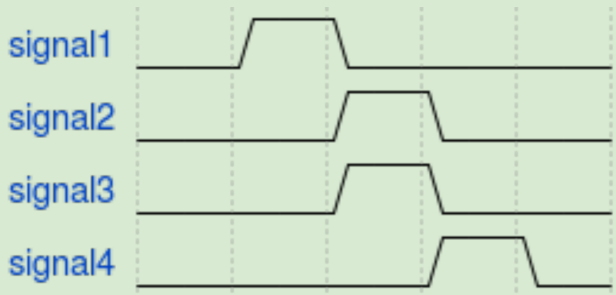
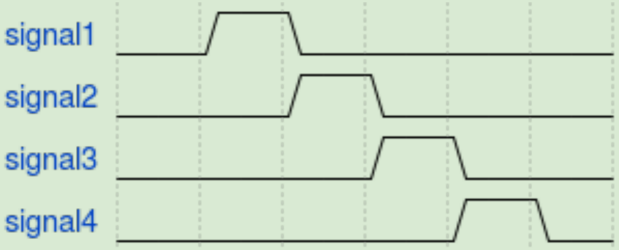
Concurrent assertions	Are SVA directives used to verify that a property holds. Syntax: [label:] assert *property* (property_name); property my_property; @(posedge clk) (rst_n !== x && rst_n != Z) endproperty assert *property* (my_property);
Cover	Is SVA directive used to verify that a property occurs during simulation. Syntax: [label:] cover *property* (property_name);

	property my_property; @(posedge clk) (rst_n !== x && rst_n != Z) **endproperty** **cover** *property* (my_property);

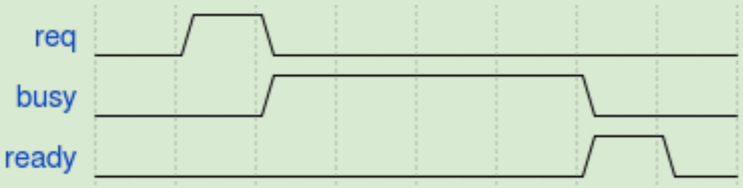
SVA Syntax

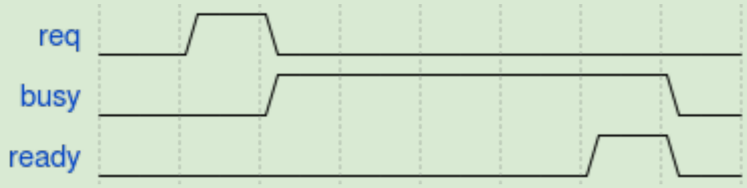
Property

Declaration	**property** my_property[(port0, port1, ...)]; <assertion variable declarations> // AVD property_statement **endproperty**
Operator	**Description**
Overlapping sequence implication operator |->	Syntax: sequence1 **|->** sequence2 sequence2 will start in the same clock cycle in which sequence1 will end Example: signal1 **##1** signal2 **|->** signal3 **##1** signal4 Antecedent Consequent

	 Figure 1
Non-overlapping sequence implication operator |=>	Syntax: sequence1 **|=>** sequence2 sequence2 will start one clock cycle after sequence1 has ended Example: signal1 **##1** signal2 **|=>** signal3 **##1** signal4  Figure 2 Observation! Seq1 **|=>** Seq2 is the same as Seq1 **|-> ##1** Seq2
not	Syntax: not my_property; It is not recommended to negate properties that contain an implication operator. The result of such a negation might be hard

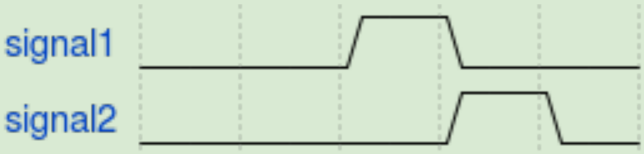
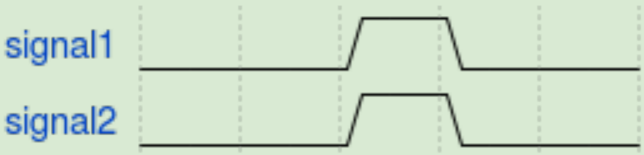
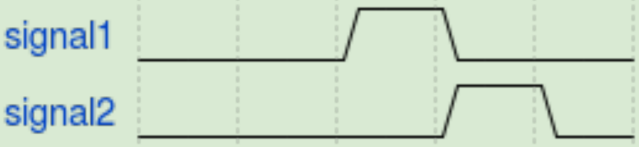
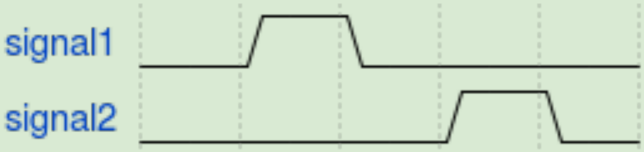
	to predict. Example: **property** my_propery; @(**posedge** clk) signal1 **|->** signal2; **endproperty** **not** my_propery is equivalent to signal1 && !signal2. The correct negation should be signal1 |-> not signal2
and	Syntax: **property** my_property; property1 **and** property2; **endproperty** Both properties should start at the same evaluation time and may end at different cycles. The assertion will pass only if both properties hold.
or	Syntax: **property** my_property; property1 **or** property2; **endproperty** Both properties should start at the same evaluation time and may end at different cycles. The assertion will pass only if at least one property holds.
until	Syntax: **property** my_property; property1 **until** property2; **endproperty**

	my_property evaluates to true if property1 evaluates to true for every clock cycle beginning with the starting point, and finishing one clock cycle before property2 starts to evaluate to true. Example: property my_property; req **|=>** busy **until** ready; endproperty  Figure 3
until_with	Syntax: **property** my_property; property1 **until_with** property2; **endproperty** my_property evaluates to true if property1 evaluates to true for every clock cycle beginning with the starting point, and finishing the same cycle when property2 starts to evaluate to true. Example: **property** my_property; req **|=>** busy **until_with** ready; **endproperty**

	 Figure 4
disable iff	Syntax: **property** my_property; **disable iff**(boolean_condition) property_statement **endproperty** Causes the assertion checking to be terminated if the boolean condition is evaluated to TRUE. Example: **property** my_property(**input** rst_n); @(**posedge** clk) **disable iff**(rst_n == 0) signal1 |=> signal2; **endproperty**

Sequences

Declaration	**sequence** my_sequence [(port0, port1, ...)] <assertion variable declarations> //AVD <sequence expressions> **endsequence**
Operators	**Description**

Temporal delay ## with integer	signal1 **##1** signal 2  Figure 5
Temporal delay ## with range	signal1 **##[0:2]** signal 2  Figure 6. 0 - delay  Figure 7 1-delay  Figure 8. 2-delay
Consecutive repetition [*m] or range [*n:m], [*],[+]	Where n,m re natural numbers, m>n>=1. The $ sign can be used to represent infinity Example: signal1[*1:2] **##1** signal2

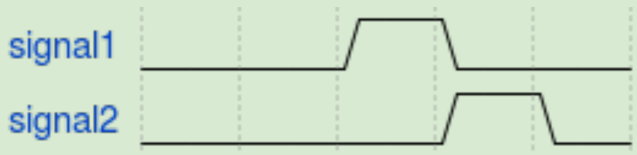


Figure 9

The length of signal1 is 1 clock cycle

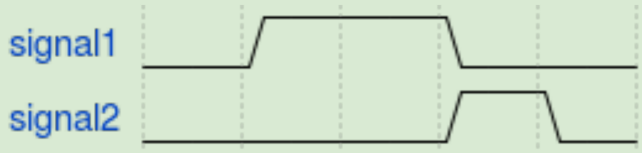
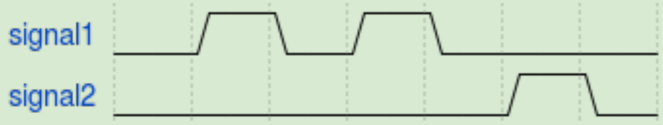
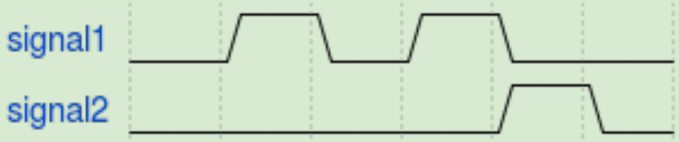


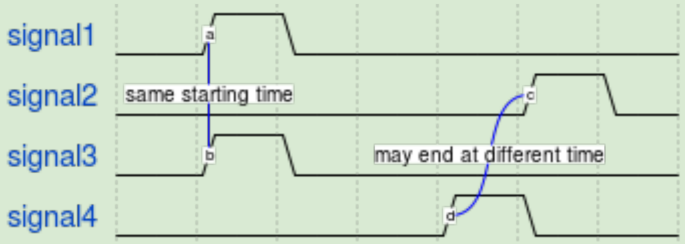
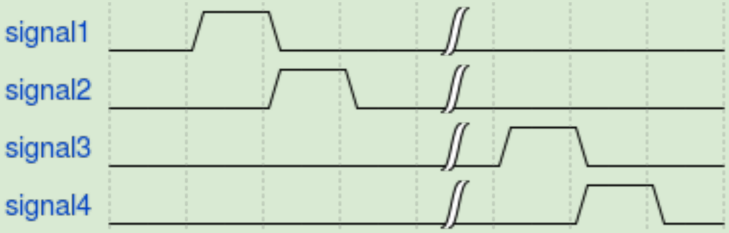
Figure 10

The length of signal1 is 2 clock cycle

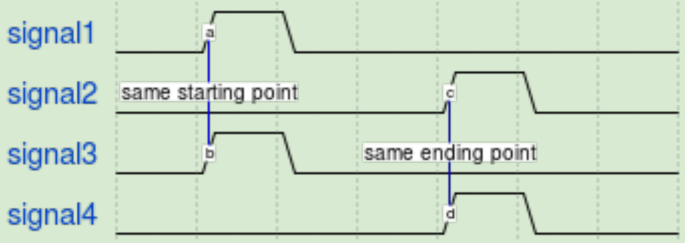
Abbreviations:

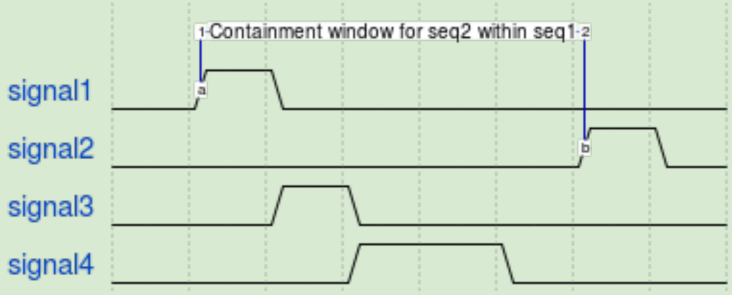
- [*] is the same as [*0:$]
- [+] is the same as [*1:$]

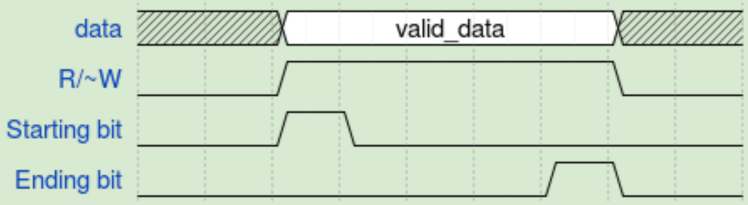
Non-consecutive repetition [=n], [=n:m]	Example1: signal1**[=2]** This is a shortcut for the following sequence !start[*0:$] ##1 start ##1 !start[*0:$] ##1 start ##1 !start[*0:$] Example2: signal1**[=2] ##1** signal2  Figure 11
Goto non-consecutive repetition [->n], [->n:m]	Example1: signal1**[->2]** The difference between the two non-consecutive repetition is that the pattern matching is finished after the last active pulse. signal1**[->2]** is a shortcut for the following sequence: !start**[*0:$] ##1** start **##1** !start**[*0:$] ##1** start **##1** Observe that signal1 is matched before returning to the LOW state Example2: signal1**[=2] ##1** signal2  Figure 12

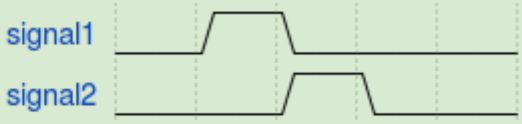
and	Syntax: seq1 **and** seq2 Example: (signal1 **##[1:8]** signal2) **and** (signal3 **##[0:$]** signal4)  Figure 13 The evaluation starts at the same clock time (if each sequence has it's own clock, then the AND starts at the first clocking event of each sequence), but it is not necessary to finish at the same time. The and sequence fails to match when any of the sequences fail to match.
or	Syntax: seq1 **or** seq2 Example: (signal1 **##1** signal2) **or** (signal3 **##1** signal4)  Figure 14 The result of or-ing two sequence is a match when at least one of

	the two sequences is a match.

intersect	It is similar to the and operator, except that the the two sequences must end at the same time. Syntax: seq1 **intersect** seq2 Example: (signal1 **##[1:8]** signal2) **intersect** (signal3 **##[0:$]** signal4)  Figure 15

within	Syntax: seq1 **within** seq2 Example: signal3 **##1** signal4**[*2] within** signal1 **##[4:6]** signal2  Figure 16 A match must satisfy the following conditions: • The starting point of seq1 must be before or at the same time as the starting point of seq2. • The ending point of seq1 must be after or at the same time as the ending point of seq2.

throughout	The throughout operator specifies that a signal must hold throughout a sequence. Syntax: signal **throughout** seq Example R/~W **throughout** (starting_bit **##[3:7]** ending_bit)  Figure 17
Method	**Description**

first_match	Used to specify that only the first sequence match is considered from a range of possible matches, the rest being discarded. Syntax: *first_match*(seq); Example: **sequence** my_seq signal1 **##[1:$]** signal2; **endsequence** **property** @(**posedge** clk) *first_match*(my_seq) **|=>** signal3 **endproperty** my_seq generates an infinite number of threads and an infinite number of matches. To prevent unexpected errors the first_match method is used to ensure that only the first of multiple matches is considered. Implicit first_match: • When a sequence is treated as a property (no implication operator in the property) Example: **assert property** @ (**posedge** clk) signal1 **##[1:2]** signal2  Figure 18. Possible match1

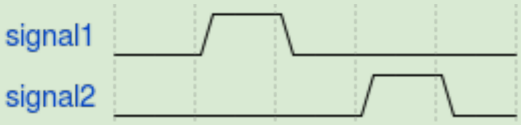


Figure 19. Possible match2

For the assertion of a sequence to be successful it is sufficient to have one matched thread.

- When a sequence is used as a consequent
  The first match of a consequent causes the property to hold and to successfully complete the assertion.
  Example:
  assert property @ (**posedge** clk) signal1 **|->**signal2 **##[1:$]** signal3

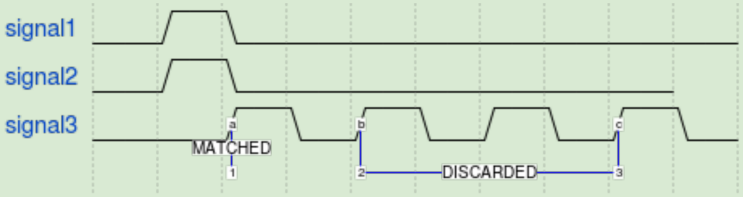


Figure 20

Required first match:

- When a sequence is used as an antecedent.
  Each thread generated by an antecedent must match to cause the property to hold. If a thread fails to match, the assertion fails.
  Example:
  assert property @ (**posedge** clk) *first_match*(signal1 **##[1:2]** signal2) **|->**signal3

  Successful assertion with first_match
  Option1:

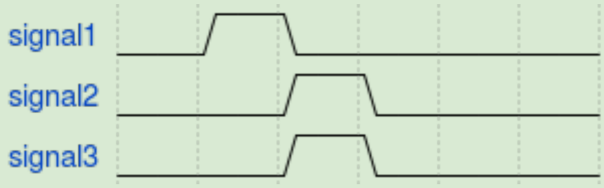


Figure 21

Option2:

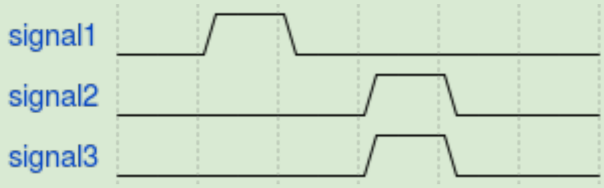


Figure 22

Successful assertion without first_match:

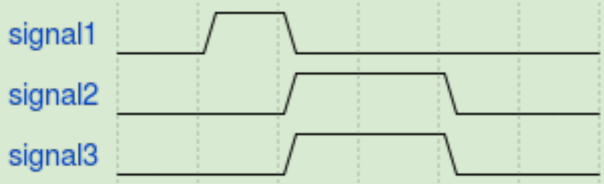


Figure 23

Failed assertion without first_match:

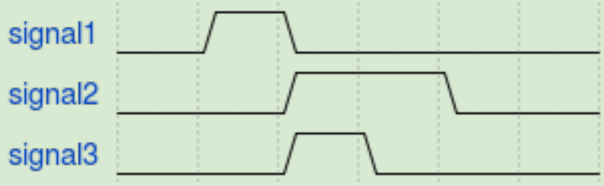
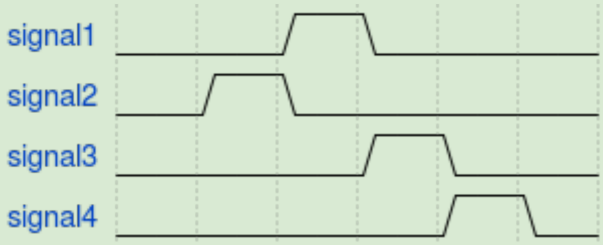


Figure 24

triggered	Used to test if the end point of a sequence was reached. An end point is a boolean expression that represents the evaluation of a thread at its last clock cycle. Syntax: My_seq.*triggered* Example: **sequence** my_seq; signal2 **##2** signal3; **endsequence** **sequence** my_new_seq; signal1 **##1** my_seq.*triggered* **##1** signal4 **endsequence**  Figure 25

Variables

Declaration of a sequence	**sequence** my_sequence[(Formal arguments)] <Assertion Variable Declaration> **endsequence**

Legal types	Legal local variable types **bit** **byte** **shortint** **integer** **logic** **struct** **int** **time** **reg** **enum** **longint** untyped Typed formal arguments that are are *not* local variables • **sequence** • **event**
Illegal types	Illegal local variable types **shortreal** **string** Associative arrays **real** **class** Dynamic arrays **realtime** **chandle**
Initialization	Rules: • Formal Arguments are initialized before AVD variables • Local variables are initialized with a nonlocal variable, using the value from the time slot in which the evaluation attempt begins. • Non-initialized local variables are unassigned and have no default values. Example: **sequence** my_sequence(//FA **local logic** fa_signal = signal1;); // AVD **logic** avd_ signal = 1; **endsequence** Fa_signal will be initialized first with the preponed value of signal1, then avd_signal will be initialized. Initialization of Formal Arguments:

	• Only local arguments of input direction can be initialized. • **inout**, **output** and event cannot be declared as **local.**

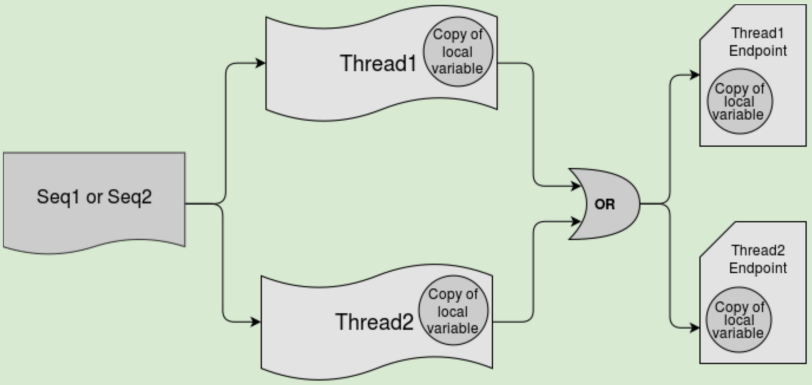
Assignments	Local variables can be assigned within the sequence matched item list, each variable being separated from each other by using **comma** in the parentheses.
	
	The variables are assigned in order of appearance.
	
	Non-local variables cannot be directly assigned in a sequence, it is necessary to assigned them through function calls.
	
	Example:
	sequence my_sequence(//FA
	local logic fa_signal = signal1;
	);
	// AVD
	logic avd_ signal = 1;
	(signal1, fa_signal = 1) ##1 (signal2, avd_signal = !signal2)
	##1 ($fall(signal3), set_flag())
	endsequence
	
	// flag and set_flag function are declared in the module
	function void set_flag()
	flag <= 1;
	endfunction
	
	Assignments can be used in repetitions:
	(signal1, counter = 0) **##1** (signal2, counter += 1)**[*10] ##1** counter == 10

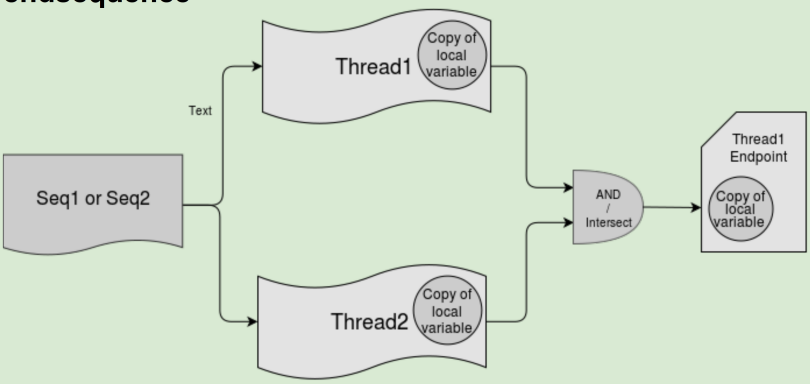
User-defined repetitions	Local variables cannot be used in temporal ranges, but they can be used as counters for user-defined repetitions or delays. Example: Illegal use of local variables *$rose*(signal1) **##1** signal2**[0:MAX] ##1** signal3 Legal use of local variables **local int** count; (*$rose*(signal1), count = 0) **##1** (signal2, count += 1)**[0:$] ##1**

	(signal3 && count < MAX)

Passing and binding Local Variables to instance of a subsequence	Method 1: Using an untyped formal argument **sequence** binded_seq(signal) signal3 **##1** (signal1, s = signal2) **endsequence** **sequence** my_seq **local logic** s; signal2 **##1** binded_seq(s) **##1** signal4 == s **endsequence** my_seq is equivalent to: signal2 **##1** signal3 **##1** (signal1, s = signal2) **##1** signal4 == s Method 2: Using a typed formal argument **sequence** binded_seq(local output logic signal) signal3 **##1** (signal1, s = signal2) **endsequence** **sequence** my_seq **local logic** s; signal2 **##1** binded_seq(s) **##1** signal4 == s **endsequence** Method 3: Using triggered **sequence** binded_seq(local output logic signal) signal3 **##3** (signal1, s = signal2) **endsequence** **sequence** my_seq **local logic** s; signal2 **##1** binded_seq(s).*triggered* **##1** signal4 == s **endsequence** In this case the end point of binded_seq(s) must occur 1 clock cycle after signal2 was asserted. The starting point of

	binded_seq(s) is before signal2 is asserted.

Using **and, or, intersect** with local variables	Using local variables on parallel "**or**" Threads.
	The **or** operand generates two concurrent threads, each thread having separate copies of the local variables. If a local variable is set on one thread, the other thread wouldn't be able to access it.
	 
	
	Using local variables on parallel "**and"/intersect"** Threads.
	The **and/intersect** operands generate two concurrent threads, each having separate copies of the local variables. At the end of the evaluations, the two threads are merged into one. This will create problems when a local variable is assigned in both threads, and later used after the threads are merged.
	
	Example:
	//illegal sequence
	sequence my_sequence;
	int signal_cp;
	((signal1 **##1** (signal2, signal_cp = signal3)) **and** (signal3 **##1** (signal4, signal_cp = signal4))) **##1** (signal1 == signal_cp)
	endsequence
	In this case we have an illegal assignment of signal_cp in both threads.
	
	// legal sequence

	sequence my_sequence; **int** signal_cp1; **int** signal_cp2; ((signal1 **##1** (signal2, signal_cp1 = signal3)) **and** (signal3 **##1** (signal4, signal_cp2 = signal4))) **##1** (signal1 == signal_cp1) **endsequence** 

System functions

Function	Description
$onehot(signal)	Returns true if only one bit of the signal HIGH
$onehot0(signal)	Returns true if only one bit of the signal is LOW
$isunknown(signal)	Returns true if at most one bit of the signal is X or Z
$rose(signal)	Returns true if the signal has changed value to 1 in the current evaluation cycle
$fell(signal)	Returns true if the signal has changed value to 0 in the current evaluation cycle

$stable(signal)	Returns true if the signal has the same value as it had in the previous evaluation cycle
$past(signal, number_of_cc)	Returns the value of the signal at a previous evaluation cycle specified through the number_of_cc argument

SVA - .json files