SYSTEM VERILOG ASSERTIONS

```
OUT : assert property(ABC)
      <Success_Message>
       else
      <Fail_Message>
```

JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER

Assertions

An assertion specifies a behavior of the system. Assertions are primarily used to validate the behavior of a design. In addition, assertions can also be used to provide functional coverage, and to flag that input stimulus, which is used for validation. Assertions can be checked dynamically by simulation, or statically by property checking or formal verification tools.

SystemVerilog supports rich constructs to implement assertions in terms of sequences and property specifications. This section will cover commonly asked questions related to SystemVerilog Assertions and methodology and help you understand these better.

1. **What is an assertion and what are the benefits of using assertions in Verification?**
   An assertion is a description of a property of the design as per specification and is used to validate the behavior of the design. If the property that is being checked for in a simulation does not behave as per specification, then the assertion fails. Similarly if a property or rule is forbidden from happening in the design and occurs during simulation, then also the assertion fails.
   Following are some of the benefits of using Assertions in Verification:
   1. Assertions improve error detection in terms of catching simulation errors as soon a design specification is violated
   2. Assertions provide better observability into design and hence help in easier debug of test failures.
   3. Assertions can be used for both dynamic simulations as well as in formal verification of design
   4. Assertions can also be used to provide functional coverage on input stimulus and to validate that a design property is infact simulated.

2. **What are different types of assertions?**
   There are two types of assertions defined by SystemVerilog language –immediate assertions and concurrent assertions.

3. **What are the differences between Immediate and Concurrent assertions?**
   Immediate assertions use expressions and are executed like a statement in a procedural block. They are not temporal in nature and are evaluated immediately when executed.
   Immediate assertions are used only in dynamic simulations. Following is an example of a simple immediate assertion that checks “if a and b are always equal”:

```
always_comb begin
a_eq_b: assert (a==b) else $error (“A not equal b“);
end
```

Concurrent assertions are temporal in nature and the test expression is evaluated at clock edges based on the sampled values of the variables involved. They are executed concurrently with other design blocks. They can be placed inside a module or an interface. Concurrent assertions can be used with both dynamic simulations as well static (formal) verification. Following is a simple example of a concurrent assertion that checks "if c is high on a clock cycle, then on next cycle, value of a and b is equal":

```
ap_a_eq_b : assert property((@posedge clk) c |=> (a == b));
```

4. **What is the difference between simple immediate assertion and deferred immediate assertions?**

   Deferred assertions are a special type of immediate assertions. Simple immediate assertions are evaluated immediately without waiting for variables in its combinatorial expression to settle down. Hence, simple immediate assertions are very prone to glitches as the combinatorial expression settles down. This can cause the assertions to fire multiple times and some of them could be false.

   To avoid this, deferred assertions are defined which gets evaluated only at the end of time stamp when the variables in the combinatorial expression settles down. This means they are evaluated in the reactive region on the timestamp.

5. **What are the different ways to write assertions for a design unit?**

   1. Assertions can be written directly inside a design module. This is mostly followed if the assertions are written by design engineers for some of the internal signals or interfaces in the design.
   2. Assertions can also be written in a separate interface or module or program and then that can be bound to a specific module or instance from which signals are referenced in assertion. This is done using the bind construct in SystemVerilog.

   This method is generally followed if the assertions are written by the Verification engineers.

6. **What is a sequence as used in writing SystemVerilog Assertions?**

   A sequence is a basic building block for writing properties or assertions. A sequence can be thought of a simple boolean expression that gets evaluated on a single clock edge or it can be a sequence of events that gets evaluated across multiple cycles. A property may involve checking of one or more sequential behaviors beginning at various times. A property can hence be constructed using multiple sequences combined logically or sequentially. The basic syntax of a sequence is as follows:

   ```
   sequence name_of_sequence;
   ```

<boolean expression >
endsequence
For Example: The following sequence samples values of a and b on every positive edge of clk and evaluates to true if both a and b are equal.

```
sequence s_a_eq_b;
@posedge(clk) (a ==b);
endsequence
```

7. **Is there a difference between $rose(tst_signal) and @posedge(tst_signal)?**
   Yes, there is a difference. @posedge(*tst_signal*) waits until a rising edge event is seen on the *tst_signal*. However, $rose() is a system task that checks if the sampled value of the signal changed to 1 between previous sample and the current sample (Previous sample could be a 0/x/z). Accordingly, $rose(tst_signal) only returns true if there are at least two sampled values.
   For example: In the following sequence, only if the signal "a" changes from a value of 0/x/z to 1 between two positive edge of clock, then $rose(a) will evaluate true

```
sequence S1;
@(posedge clk) $rose(a)
Endsequence
```

8. **Explain when the following sequence matches?**
   req ##2 gnt ##1 !req
   When *gnt* signal goes high two cycles after *req* signal is high, and one cycle after that *req* signal is deasserted to zero, this sequence will evaluate to true.

9. **What is a sequence repetition operator? What are the three different type of repetition operators used in sequences?**
   If a sequential expression needs to be evaluated for more than one iteration, then instead of writing a long sequence, repetition operator can be used to construct a longer sequence.
   SVA supports three types of repetition operators:
   1. **Consecutive Repetition** ([*const_or_range_expression]): If a sequence repeats for a finite number of iterations with a delay of one clock tick from end of one iteration, then a consecutive repetition operator can be used. Following is an example of how to use consecutive repetition operator.

```
a ##1 b [*5]
```

   In above example, if "a" goes high and then if "b" remains high for 5 consecutive cycles, we can use the repetition operator [*] to specify number of iterations.

2. **Go-to repetition** ([->const_or_range_expression]): Go-to repetition specifies finitely many iterative matches of the operand Boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at the last iterative match of the operand.

a ##1 b [->2:10] ##1 c

In the above example, the sequence matches over an interval of consecutive clock ticks provided **a** is true on the first clock tick, **c** is true on the last clock tick, **b** is true on the penultimate clock tick, and, including the penultimate, there are at least 2 and at most 10 not necessarily consecutive clock ticks strictly in between the first and last on which **b** is true.

3. **Non-consecutive repetition** [=const_or_range_expression]): The Nonconsecutive repetition is like the Go-to repetition except that a match does not have to end at the last iterative match of the operand Boolean expression

a ##1 b [=2:10] ##1 c

Above sequence shows the same example using non-consecutive repetition. The difference between this repetition and the Go-to repetition is that in this case: after we see a minimum of 2 and maximum of 10 occurrences of non-consecutive **b,** there can be several cycles where **b** is not true and then **c** can be true. Whereas, in a sequence that uses Go-to repetition, after the maximum number of **b** occurrences are seen, next cycle needs to have **c** as true.

10. Write an assertion check to make sure that a signal is high for a minimum of 2 cycles and a maximum of 6 cycles.

Following property uses a sequence such that if a signal “a” rises, then from same cycle, we check it remains high for a minimum of 2 and maximum of 6 cycles and in the next cycle “a” goes low.

```
property a_min_2_max_6: @(posedge clk)
$rose(a) |-> a[*2:6] ##1 (a==0)
endproperty
assert property (a_min_2_max_6);
```

11. What is an implication operator?

An implication operator specifies that the checking of a property is performed conditionally on the match of a sequential antecedent. This construct is used to precondition monitoring of a property expression and is allowed only at the property level. Following is the syntax of two types of implication operators supported in property expressions:

1. **Overlapped Implication Operator** (|->)

assert property prop_name (sequence_expr |-> property_expr)

2. **Non-Overlapped Implication Operator** (|=>)

assert property prop_name (sequence_expr |=> property_expr)

In above examples, the left hand side of the implication operator is called antecedent and the right hand side of the operator is called consequent. The antecedent is the precondition that needs to happen before evaluating the consequent.

12. What is the difference between an overlapping and nonoverlapping implication operator?

1. **Overlapped Implication Operator** (|->):

   For overlapped implication, if there is a match for the antecedent sequence_expr, then the endpoint of the match is the start point of the evaluation of the consequent property expression. For Example: In following example, as soon as a match happens on the sequence (a==1), in the same cycle if "b" is true and the following cycle "c" is true then this property passes.

assert property abc_overlap (@posedge clk (a==1) |-> b ##1 c)

2. **Non-Overlapped Implication Operator** (|=>):

   For non overlapped implication, the start point of the evaluation of the consequent property_expr is the clock tick after the end point of the match of antecedent. For Example: In following example, when (a==1) matches on any clock cycle, then in next cycle if "b" is true and a cycle later if "c" is true, then following property will pass.

assert property abc_overlap (@posedge clk (a==1) |=> b ##1 c)

13. Are following assertions equivalent?

1) @(posedge clk) req |=> ##2 $rose(ack);
2) @(posedge clk) req |-> ##3 $rose(ack);

Yes: |-> is an overlapping operator that starts evaluating the consequent in same cycle when antecedent is true while |=> is non overlapping operator that starts consequent evaluation a cycle after antecedent is true. So, adding an explicit cycle delay after overlapping operator will make it equivalent to non-overlapping operator.

14. What does the system task $past() do?

$past is a system task that is capable of getting values of signals from previous clock cycles.

15. Write an assertion checker to make sure that an output signal never goes X?

The system function $isunknown(signal) returns a value of 1 if the signal has an unknown value (x). Hence this can be used to write an assertion as below.

```
assert property (@(posedge clk) ! $isunknown(mysignal));
```

16. Write an assertion to make sure that the state variable in a state machine is always one hot value.

The $isonehot() system function returns if a bit vector is one hot. Hence, this can be used to write an assertion as follows:

```
assert property (@(posedge clk) $isonehot(state));
```

17. Write an assertion to make sure that a 5-bit grant signal only has one bit set at any time? (only one req granted at a time)

The system function $countones() will return the number of ones present in a signal. Hence, this can be used to write an assertion to check for number of bits set in any signal as follows:

```
assert property (@(posedge clk) $countones(grant[5:0])==1);
```

18. Write an assertion which checks that once a valid request is asserted by the master, the arbiter provides a grant within 2 to 5 clock cycles

```
property p_req_grant;
@(posedge clk) $rose (req) |-> ##[2:5] $rose (gnt);
endproperty
```

19. How can you disable an assertion during active reset time?

A property can use a "**disable iff"** construct to explicitly disable an assertion. Following is an example that disables an assertion check when reset is active high.

```
assert property (@(posedge clk) disable iff (reset) a |=> b);
```

20. How can all assertions be turned off during simulation?

Assertions can be turned off during a simulation using the **$assertoff()** system task. If no arguments are specified, all the assertions are disabled.

If this system task is called in the middle of simulation, then any active assertions at that given point of time are allowed to complete before disabling.

For selectively disabling assertions, the task supports two arguments as follows:

```
$assertoff[(levels[, list])]
```

The first argument specifies how many levels of hierarchy this applies and the second argument is a list of properties that need to be turned off in these levels of hierarchy.

21. What are the different ways in which a clock can be specified to a property used for assertion?

There are different ways in which a clock can be specified to a property as explained below:

1. A sequence instance that is used in property has an explicit clock specified. In this case property uses that clock.

```
sequence seq1;
@(posedge clk) a ##1 b;
endsequence
property prop1;
not seq1;
endproperty
assert property (prop1);
```

2. Specify the clock explicitly in the property.

```
property prop1;
@(posedge clk) not (a ##1 b);
endproperty
assert property (prop1);
```

3. Infer the clock from the procedural block in which the property is used as shown below.

```
always @(posedge clk) assert property (not (a ##1 b));
```

4. If the property is defined in a clocking block, the clock of the clocking block can be inferred in the property. The property can be used to assert outside by hierarchical reference as shown below:

```
clocking master_clk @(posedge clk);
property prop1;
not (a ##1 b);
endproperty
endclocking
assert property (master_clk.prop1);
```

5. If none of the above is used, then the clock will be resolved to the default clocking event. For Example: if a clocking block (shown above) has defined a default clocking event (as shown below) then the property infers the same clock.

```
default clocking master_clk ; // master clock as defined above
property p4;
not (a ##1 b);
```

```
endproperty
assert property (p4);
```

22. For a synchronous FIFO of depth=32, write an assertion for following scenarios. Assume a clock signal (clk), write and read enable signals, full flag and a word counter signal.

[1]. If the word count is >31, FIFO full flag is set.

[2]. If the word count is 31 and a new write operation happens without a simultaneous read, then the FIFO full flag gets set.

```
assert property (@(posedge clk) disable iff (!rst_n) (wordcnt>31 |-> fifo_full));
assert property (@(posedge clk) disable iff (!rst_n) (wordcnt==31 && write_en &&
!read_en |=> fifo_full));
```

Note that for the second case, a non-overlapping implication operator is used as the full flag will go high only in next cycle after write_enable is seen.