SystemVerilog Assertions Tutorial

Introduction

Assertions are primarily used to validate the behaviour of a design. ("Is it working correctly?") They may also be used to provide functional coverage information for a design ("How good is the test?"). Assertions can be checked dynamically by simulation, or statically by a separate property checker tool – i.e. a formal verification tool that proves whether or not a design meets its specification. Such tools may require certain assumptions about the design's behaviour to be specified.
In SystemVerilog there are two kinds of assertions: immediate (`assert`) and concurrent (`assert property`). Coverage statements (`cover property`) are concurrent and have the same syntax as concurrent assertions, as do `assume property` statements. Another similar statement – `expect` – is used in testbenches; it is a procedural statement that checks that some specified activity occurs. The three types of concurrent assertion statement and the `expect` statement make use of sequences and properties that describe the design's temporal behaviour – i.e. behaviour over time, as defined by one or more clocks.

Immediate Assertions

Immediate assertions are procedural statements and are mainly used in simulation. An assertion is basically a statement that something must be true, similar to the `if` statement. The difference is that an `if` statement does not assert that an expression is true, it simply checks that it is true, e.g.:

```
if (A == B) ...  // Simply checks if A equals B
assert (A == B); // Asserts that A equals B; if not, an error is generated
```

If the conditional expression of the immediate `assert` evaluates to X, Z or 0, then the assertion fails and the simulator writes an error message. An immediate assertion may include a pass statement and/or a fail statement. In our example the *pass statement* is omitted, so no action is taken when the `assert` expression is true. If the pass statement exists:

```
assert (A == B) $display ("OK. A equals B");
```

it is executed immediately after the evaluation of the `assert` expression. The statement associated with an `else` is called a *fail statement* and is executed if the assertion fails:

```
assert (A == B) $display ("OK. A equals B");
  else $error("It's gone wrong");
```

Note that you can omit the pass statement and still have a fail statement:

```
assert (A == B) else $error("It's gone wrong");
```

The failure of an assertion has a severity associated with it. There are three severity system tasks that can be included in the fail statement to specify a severity level: `$fatal`, `$error` (the default severity) and `$warning`. In addition, the system task `$info` indicates that the assertion failure carries no specific severity.
Here are some examples:

```
ReadCheck: assert (data === correct_data)
             else $error("memory read error");
Igt10: assert (I > 10)
         else $warning("I is less than or equal to 10");
```

The pass and fail statements can be any legal SystemVerilog procedural statement. They can be used, for example, to write out a message, set an error flag, increment a count of errors, or signal a failure to another part of the testbench.

```
AeqB: assert (a === b)
  else begin error_count++; $error("A should equal B"); end
```

Concurrent Assertions

The behaviour of a design may be specified using statements similar to these:
"The Read and Write signals should never be asserted together."
"A Request should be followed by an Acknowledge occurring no more than two clocks after the Request is asserted."
Concurrent assertions are used to check behaviour such as this. These are statements that assert that *specified properties* must be true. For example,

```
assert property (!(Read && Write));
```

asserts that the expression `Read && Write` is never true at any point during simulation.
Properties are built using sequences. For example,

```
assert property (@(posedge Clock) Req |-> ##[1:2] Ack);
```

where `Req` is a simple sequence (it's just a boolean expression) and `##[1:2] Ack` is a more complex sequence expression, meaning that `Ack` is true on the next clock, or on the one following (or both). `|->` is the implication operator, so this assertion checks that whenever `Req` is asserted, `Ack` must be asserted on the next clock, or the following clock.
Concurrent assertions like these are checked throughout simulation. They usually appear outside any initial or always blocks in modules, interfaces and programs. (Concurrent assertions may also be used as statements in initial or always blocks. A concurrent assertion in an initial block is only tested on the first clock tick.)
The first assertion example above does not contain a clock. Therefore it is checked at every point in the simulation. The second assertion is only checked when a rising clock edge has occurred; the values of `Req` and `Ack` are sampled on the rising edge of Clock.

Implication

The *implication* construct (`|->`) allows a user to monitor sequences based on satisfying some criteria, e.g. attach a precondition to a sequence and evaluate the sequence only if the condition is successful. The left-hand side operand of the implication is called the *antecedent sequence expression*, while the right-hand side is called the *consequent sequence expression*.
If there is no match of the *antecedent sequence expression*, implication succeeds vacuously by returning true. If there is a match, for each successful match of the *antecedent sequence expression*, the *consequent sequence expression* is separately evaluated, beginning at the end point of the match.
There are two forms of *implication*: overlapped using operator `|->`, and non-overlapped using operator `|=>`.
For overlapped implication, if there is a match for the *antecedent sequence expression*, then the first element of the *consequent sequence expression* is evaluated on the same clock tick.

```
s1 |-> s2;
```

In the example above, if the sequence `s1` matches, then sequence `s2` must also match. If sequence `s1` does not match, then the result is true.
For non-overlapped implication, the first element of the *consequent sequence expression* is evaluated on the next clock tick.

```
s1 |=> s2;
```

The expression above is basically equivalent to:

```
`define true 1
s1 ##1 `true |-> s2;
```

where `` `true `` is a boolean expression, used for visual clarity, that always evaluates to true.

Properties and Sequences

In these examples we have been using, the properties being asserted are specified in the `assert property` statements themselves.
Properties may also be declared separately, for example:

```
property not_read_and_write;
  not (Read && Write);
endproperty assert property (not_read_and_write);
```

Complex properties are often built using sequences. Sequences, too, may be declared separately:

```
sequence request
  Req;
endsequence

sequence acknowledge
  ##[1:2] Ack;
endsequence

property handshake;
  @(posedge Clock) request |-> acknowledge;
endproperty

assert property (handshake);
```

Assertion Clocking

Concurrent assertions (`assert property` and `cover property` statements) use a generalised model of a clock and are only evaluated when a clock tick occurs. (In fact the values of the variables in the property are sampled right at the end of the previous time step.) Everything in between clock ticks is ignored. This model of execution corresponds to the way a RTL description of a design is interpreted after synthesis.
A clock tick is an atomic moment in time and a clock ticks only once at any simulation time. The clock can actually be a single signal, a gated clock (e.g. `(clk && GatingSig)`) or other more complex expression. When monitoring asynchronous signals, a simulation time step corresponds to a clock tick.
The clock for a property can be specified in several ways:
o Explicitly specified in a sequence:

```
sequence s;
  @(posedge clk) a ##1 b;
endsequence
property p;
  a |-> s;
endproperty
assert property (p);
```

o Explicitly specified in the property:

```
property p;
  @(posedge clk) a ##1 b;
endproperty
assert property (p);
```

o Explicitly specified in the concurrent assertion:

```
assert property (@(posedge clk) a ##1 b);
```

o Inferred from a procedural block:

```
property p;
  a ##1 b;
endproperty
always @(posedge clk) assert property (p);
```

o From a `clocking` block (see the Clocking Blocks tutorial):

```
clocking cb @(posedge clk);
  property p;
    a ##1 b;
  endproperty
endclocking
assert property (cb.p);
```

o From a default clock (see the Clocking Blocks tutorial):

```
default clocking cb;
```

Handling Asynchronous Resets

In the following example, the `disable iff` clause allows an asynchronous reset to be specified.

```
property p1;
  @(posedge clk) disable iff (Reset) not b ##1 c;
endproperty

assert property (p1);
```

The `not` negates the result of the sequence following it. So, this assertion means that if `Reset` becomes true at any time during the evaluation of the sequence, then the attempt for `p1` is a success. Otherwise, the sequence `b ##1 c` must never evaluate to true.

Sequences

A `sequence` is a list of boolean expressions in a linear order of increasing time. The `sequence` is true over time if the boolean expressions are true at the specific clock ticks. The expressions used in sequences are interpreted in the same way as the condition of a procedural `if` statement.
Here are some simple examples of sequences. The `##` operator delays execution by the specified number of clocking events, or clock cycles.

```
a ##1 b               // a must be true on the current clock tick
                      // and b on the next clock tick
a ##N b               // Check b on the Nth clock tick after a
a ##[1:4] b           // a must be true on the current clock tick and b
                      // on some clock tick between the first and fourth
                      // after the current clock tick
```

The `*` operator is used to specify a consecutive repetition of the left-hand side operand.

```
a ##1 b [*3] ##1 c    // Equiv. to a ##1 b ##1 b ##1 b ##1 c
(a ##2 b) [*2]        // Equiv. to (a ##2 b ##1 a ##2 b)
(a ##2 b)[*1:3]       // Equiv. to (a ##2 b)
                      // or (a ##2 b ##1 a ##2 b)
                      // or (a ##2 b ##1 a ##2 b ##1 a ##2 b)
```

The `$` operator can be used to extend a time window to a finite, but unbounded range.

```
a ##1 b [*1:$] ##1 c  // E.g. a b b b b c
```

The `[->` or *goto repetition* operator specifies a non-consecutive sequence.

```
a ##1 b[->1:3] ##1 c  // E.g. a !b b b !b !b b c
```

This means a is followed by any number of clocks where c is false, and b is true between 1 and three times, the last time being the clock before c is true.
The [= or *non-consecutive repetition* operator is similar to *goto repetition*, but the expression (`b` in this example) need not be true in the clock cycle before `c` is true.

```
a ##1 b [=1:3] ##1 c // E.g. a !b b b !b !b b !b !b c
```

Combining Sequences

There are several operators that can be used with sequences:
The binary operator `and` is used when both operand expressions are expected to succeed, but the end times of the operand expressions can be different. The end time of the end operation is the end time of the sequence that terminates last. A sequence succeeds (i.e. is true over time) if the boolean expressions containing it are true at the specific clock ticks.

```
s1 and s2        // Succeeds if s1 and s2 succeed. The end time is the
                 // end time of the sequence that terminates last
```

If `s1` and `s2` are sampled booleans and not sequences, the expression above succeeds if both `s1` and `s2` are evaluated to be true.
The binary operator `intersect` is used when both operand expressions are expected to succeed, and the end times of the operand expressions must be the same.

```
s1 intersect s2 // Succeeds if s1 and s2 succeed and if end time of s1 is
                // the same with the end time of s2
```

The operator `or` is used when at least one of the two operand sequences is expected to match. The sequence matches whenever at least one of the operands is evaluated to true.

```
s1 or s2         // Succeeds  whenever at least one of two operands s1
                 // and s2 is evaluated to true
```

The `first_match` operator matches only the first match of possibly multiple matches for an evaluation attempt of a sequence expression. This allows all subsequent matches to be discarded from consideration. In this example:

```
sequence fms;
  first_match(s1 ##[1:2] s2);
endsequence
```

whichever of the `(s1 ##1 s2)` and `(s1 ##2 s2)` matches first becomes the result of sequence `fms`.
The `throughout` construct is an abbreviation for writing:

```
(Expression) [*0:$] intersect SequenceExpr
```

i.e. `Expression throughout` *`SequenceExpr`* means that *`Expression`* must evaluate true at every clock tick during the evaluation of *`SequenceExpr`*.
The `within` construct is an abbreviation for writing:

```
(1[*0:$] ##1 SeqExpr1 ##1 1[*0:$]) intersect SeqExpr2
```

i.e. *`SequenceExpr1`* `within` *`SequenceExpr2`* means that *`SeqExpr1`* must occur at least once entirely within *`SeqExpr2`* (both start and end points of *`SeqExpr1`* must be between the start and the end point of *`SeqExpr2`*).

Variables in Sequences and Properties

Variables can be used in sequences and properties. A common use for this occurs in pipelines:

```
`define true 1

property p_pipe;
  logic v;
  @(posedge clk) (`true,v=DataIn) ##5 (DataOut === v);
endproperty
```

In this example, the variable `v` is assigned the value of `DataIn` unconditionally on each clock. Five clocks later, `DataOut` is expected to equal the assigned value. Each invocation of the property (here there is one invocation on every clock) has its own copy of `v`. Notice the syntax: the assignment to `v` is separated from a sequence expression by a comma, and the sequence expression and variable assignment are enclosed in parentheses.

Coverage Statements

In order to monitor sequences and other behavioural aspects of a design for *functional coverage*, `cover property` statements can be used. The syntax of these is the same as that of `assert property`. The simulator keeps a count of the number of times the property in the cover property statement holds or fails. This can be used to determine whether or not certain aspects of the designs functionality have been exercised.

```
module Amod2(input bit clk);
  bit X, Y;
  sequence s1;
    @(posedge clk) X ##1 Y;
  endsequence
  CovLavel: cover property (s1);
  ...
endmodule
```

SystemVerilog also includes `covergroup` statements for specifying functional coverage. These are introduced in the Constrained-Random Verification Tutorial.

Assertion System Functions

SystemVerilog provides a number of system functions, which can be used in assertions.
`$rose`, `$fell` and `$stable` indicate whether or not the value of an expression has changed between two adjacent clock ticks. For example,

```
assert property
  (@(posedge clk) $rose(in) |=> detect);
```

asserts that if in changes from 0 to 1 between one rising clock and the next, detect must be 1 on the following clock.
This assertion,

```
assert property
  (@(posedge clk) enable == 0 |=> $stable(data));
```

states that `data` shouldn’t change whilst `enable` is 0.
The system function `$past` returns the value of an expression in a previous clock cycle. For example,

```
assert property
  (@(posedge clk) disable iff (reset)
                  enable |=> q == $past(q+1));
```

states that `q` increments, provided `reset` is low and `enable` is high.
Note that the argument to `$past` may be an expression, as shown above.
The system functions `$onehot` and `$onehot0` are used for checking one-hot encoded signals. `$onehot(expr)` returns true if exactly one bit of `expr` is high; `$onehot0(expr)` returns true if at most one bit of `expr` is high.

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

There are other system functions.

Binding

We have seen that assertions can be included directly in the source code of the modules in which they apply. They can even be embedded in procedural code. Alternatively, verification code can be written in a separate program, for example, and that program can then be bound to a specific module or module instance.
For example, suppose there is a module for which assertions are to be written:

```
module M (...);
  // The design is modelled here
endmodule
```

The properties, sequences and assertions for the module can be written in a separate program:

```
program M_assertions(...);
  // sequences, properties, assertions for M go here
endprogram
```

This program can be bound to the module `M` like this:

```
bind M M_assertions M_assertions_inst (...);
```

The syntax and meaning of `M_assertions` is the same as if the program were instanced in the module itself:

```
module M (...);
  // The design is modelled here
  M_assertions M_assertions_inst (...);
endmodule
```

Prev Next