



SUPERLOG[®] Design Assertion Subset

April 4, 2002

Revision 1.8

Copyright © 2001-2 Co-Design Automation Inc.

This document has been submitted to Accellera under the agreed terms of the “Co-Design Accellera SUPERLOG DAS Donation Agreement” of 27th February 2002. Usage is only permitted under the terms of that agreement.

Do not copy, fax, reproduce, or distribute without written permission.

Change History	3
Introduction	3
Syntax	4
Immediate Assertions	4
Strobed Assertions	6
Sequential Assertions	7
More Expression Sequences	10
Aborting Assertions Externally	10
Controlling Assertions	10
System Functions	11

Change History

Version 1.8 of this document has been updated from version 1.5 to reflect the results of the Accellera SystemVerilog Assertions committee meetings held on

Friday, 3/8/02 (phone)

Wednesday, 3/13/02 (phone and in-person at Verplex in Milpitas, CA)

Thursday, 3/21/02 (phone)

Thursday, 3/28/02 (phone)

Thursday, 4/4/02 (phone)

Introduction

An assertion is a statement that a property must be true. There are two kinds of assertions: concurrent assertions which state that the property must be always be true, e.g. throughout a simulation, and procedural assertions which are incorporated in procedural code and apply only for a limited time or under limited conditions.

There are various applications of assertions. They can be included in the design, to document the assumptions made by the designer and to facilitate "white box" testing. They can be outside the design, either in a testbench to check the response of the design to the stimulus, or to control a tool such as a stimulus generator or a model checker.

Concurrent assertions can be coded as modules in a library, but this limits the complexity of the property that can be expressed easily. It is more difficult to code procedural assertions as a library of tasks in Verilog, because events cannot be arguments, each assertion may need static data, and tasks block.

Procedural Assertions

SUPERLOG Design Assertions Subset provides four kinds of procedural assertions, which allow the user to test boolean expressions or sequences of boolean expressions, and perform some action based on whether the expression or sequence is true or false. *Immediate* assertions test the value of a boolean expression at the time the statement is executed, and may be used in always and initial blocks, tasks and functions. *Strobed* assertions schedule the evaluation of the expression to be delayed until the end of the current timeslice, to allow for glitches to settle. Strobed assertions may be used in initial and always blocks and tasks, but not in functions since functions must return immediately.

To test sequences of expressions, it is necessary to specify a *sampling clock* event on which to test each element of the sequence. Therefore, *Clocked Immediate* and *Clocked Strobed* assertions are added to allow progressive evaluation of sequences of expressions. Since these clocked assertions, by definition, take time, they cannot be used in functions. Clocked immediate assertions evaluate each expression in the sequence when the clock event triggers, and clocked strobed assertions evaluate each expression at the end of the timeslice at which the event triggers.

Syntax

```
<proc_assertion> ::= [<identifier> ':' ] <immediate_assert>
                  | [<identifier> ':' ] <strobed_assert>
                  | [<identifier> ':' ] <clocked_immediate_assert>
                  | [<identifier> ':' ] <clocked_strobed_assert>

<immediate_assert> ::= 'assert' '(' <expression> ')'
                    <statement_or_null> //pass
                    ['else' <statement_or_null>] //fail

<strobed_assert> ::= 'assert_strobe' '(' <expression> ')'
                  <restricted_statement_or_null> //pass
                  ['else' <restricted_statement_or_null>] //fail

<clocked_immediate_assert> ::= 'assert' [<reset>]
                             '(' <formula_expr> ')' <step_control>
                             <statement_or_null> //pass
                             ['else' <statement_or_null>] //fail

<clocked_strobed_assert> ::= 'assert_strobe' [<reset>]
                             '(' <formula_expr> ')' <step_control>
                             <restricted_statement_or_null>
                             ['else' <restricted_statement_or_null>] //fail

<formula_expr> ::= <expr_sequence>
                 | <expr_sequence> 'triggers' <formula_expr>

<expr_sequence> ::= <expression>
                  | '[' <constant_expression> ']' // skip n steps
                  | <range> // skip m to n steps
                  | <expr_sequence> ';' <expr_sequence> // sequence
                  | <expr_sequence> '*' '[' <constant_expression> ']'
                    //repetition
                  | <expr_sequence> '*' <range> // bounded repetition
                  | '(' <expr_sequence> ') '

<range> ::= '[' <constant_expression> ':' <constant_expression> ']'

<step_control> ::= '@@' <name>
                 | '@@' '(' <event_expressions> ') '

<reset> ::= 'accept' '(' <expression> ')' // sync
            | 'accept' <event_control> // async
            | 'reject' '(' <expression> ')' // sync
            | 'reject' <event_control> // async
```

The immediate assert statement is a test of an expression performed when the statement is executed in the

procedural code. The expression is treated as a condition like in an if statement:

```
[<identifier> ':'] assert (<expression>) [<pass_statement>] [else <fail_statement>]
```

The pass statement is executed if the assertion succeeds, i.e. the expression evaluates to 'true'. As with the 'if' statement, if the expression evaluates to 'X', 'Z' or '0', then the assertion fails. The pass statement may, for example, record the number of successes for a coverage log, but may be omitted altogether. If the pass statement is omitted, then no action is taken if the assert expression is true. The fail statement is executed if the assertion fails (i.e. the expression does not evaluate to a known, non-zero value) and can be omitted. The optional assertion label (identifier and colon) creates a notional named block around the assertion statement (or any other SystemVerilog statement) and can be displayed using the %m format code.

```
assert_foo : assert (foo) $display("%m passed"); else $display("%m failed");
```

Since the assertion is a statement that something must be true, the failure of an assertion shall have a severity associated with it. By default, the severity of an assertion failure is "error." Other severity levels may be specified by including one of the following severity system tasks in the fail statement:

- \$fatal is a Run-time Fatal, which terminates the simulation with an error code. The first argument passed to \$fatal shall be consistent with the argument to \$finish.
- \$error is a Run-time Error.
- \$warning is a Run-time Warning, which can be suppressed in a tool-specific manner.
- \$info indicates that the assertion failure carries no specific severity.

```
<assert_msg> ::= <fatal_func> | <msg_func>;  
  
<fatal_func> ::= $fatal ('<finish_num>', ' // finish arg  
                        [<format_string>]', '[<expr>...]' );  
;  
  
<msg_func> ::= <msg_name> (' [<format_string>]', '[<expr>...]' );  
  
<msg_name> ::= $error | $warning | $info
```

The syntax for these system tasks is shown in the above syntax box. All of these severity system tasks shall print a tool-specific message indicating the severity of the failure, and specific information about the specific failure, which must include the following information:

- The file name and line number of the assertion statement,
- The hierarchical name of the assertion, if it is labeled, or the scope of the assertion if it is not labeled.

Each system task may also include additional user-specified information in the format of \$display. For simulation tools, these tasks shall also include the simulation run-time at which the severity system task is called. If more than one of these system tasks is included in the else clause, then each shall be executed as specified.

If an assertion fails and no else clause is specified, the tool shall, by default, call \$error() unless a tool-specific command-line option is enabled to suppress the failure.

If the severity system task is executed at a time other than when the assertion fails, the actual failure time of the assertion may be recorded and displayed programmatically, as in:

```
time t;
```

```
always @(posedge clk)  
if(state == REQ)  
  assert(req1 || req2)  
  else begin  
    t = $time;  
    #5 $error("assert failed at time %0t",t);  
  end
```

end

If the assertion fails at time 10, the error message will be printed at time 15, but the user-defined string printed will be "assert failed at time 10".

The display of messages of warning and info types may be controlled by a tool-specific command-line option.

Since the fail statement, like the pass statement, is any legal SystemVerilog procedural statement, it may also be used to signal a failure to another part of the testbench:

```
assert (myfunc(a,b)) count1 = count + 1; else ->event1;  
assert (y == 0); else flag = 1;
```

The assert statement serves as guidance to non-simulation tools that the condition should be true. The second statement above is equivalent to:

```
if ( y!=0) begin flag = 1; end
```

Strobed Assertions

If an immediate assertion is in code triggered by a timing control that happens at the same time as a blocking assignment to the data being tested, there is a risk of the wrong value being sampled. For example:

```
always @(posedge clock) a = a + 1; // blocking assignment  
always @(posedge clock) begin  
....  
assert (a < b);  
end
```

This can be solved by using a strobed assertion, which waits in the background until the end of the time slot, like the \$strobe system task.

```
always @(posedge clock) begin  
....  
cas:assert_strobe (a < b);  
end
```

Strobed assertions can have pass or fail statements like immediate assertions. However, they are restricted to another assertion statement, a system task call, a statement preceded by a delay control or an event control, or sequential block containing them. This is because the statement happens after the assertion is evaluated, at the end of the time slot, and hence must not create more events at that time slot or change values. Statements which cause additional events to occur at the current time shall be an error.

The example below illustrates the effect of blocking and non-blocking assignments on immediate and strobed assertions. The immediate assertions are like \$display statements and the strobed assertions are like \$strobe statements:

```
module test;  
reg [3:0] a=0; c=0, d=0;  
reg clk = 0;  
wire b;  
  
initial begin  
#10 clk = 1;  
forever #5 clk = !clk; // posedge clk at 10,20,30,40...  
end  
  
assign b = a+1;
```

```

always @(posedge clk) begin
  a1: assert(c<3); // fails at time 40
  c = c+1;
  a2: assert(c<3); // fails at time 30
  a <= a+1;
  a3: assert(a<3); // fails at time 40
  a4: assert(b<3); // fails at time 40
  a5: assert_strobe(a<3); // fails at time 30
  a6: assert_strobe(b<3); // fails at time 30
end

always @(a) begin // models transient behavior on comb. nets
  d = a+2; // spikes to 2 at 0, 3 at 10, 4 at 20
  assert(d<3); // fails at time 10
  d = d-1; // settles to 1 at 0, 2 at 10, 3 at 20
  assert(d<3); // fails at time 20
end

always @(d) assert_strobe (d<3); // fails at time 20

endmodule

```

Sequential Assertions

In addition to assertions about single expressions, it is often useful to assert sequences of expressions over time. One way of doing this is to use nested immediate assertions:

```

always @(posedge clk or negedge rst)
  if(state == REQ)
    a7: assert(req1) // no semicolon
        @(posedge clk) assert (gnt) @(posedge clk) assert(!req1);

```

The above example verifies the sequence that, if state is equal to REQ, the req1 signal must be true immediately, then on the next posedge clk, gnt must be true and on the following posedge clk, req must be false. Note that the assertion statement itself is nonblocking, so the sequence in assertion a7 is equivalent to

```

always @(posedge clk or negedge rst)
  if(state == REQ)
    a8: assert(req1)
        process // Please see the SystemVerilog Spec, section 10,
                // for more details about the process statement
            @(posedge clk) assert (gnt) @(posedge clk) assert(!req1);

```

To simplify this complex nested assertion, a *sequential regular expression* is used in the assert statement. Sequential regular expressions require a *step control* event expression to specify the timing between evaluations of each element in the regular expression. Using a sequential regular expression, the assertion a8 could be rewritten as

```

always @(posedge clk or negedge rst)
  if(state == REQ)
    a9: assert(req1;gnt;!req1) @@(posedge clk); // note the @@ token to distinguish the step control
        // from the pass statement

```

A sequential regular expression is a semicolon-delimited list of expressions. The first expression in the list is evaluated immediately when the assert statement is executed. The other subsequent expressions are evaluated one at a time on successive occurrences of the step control event expression. In assertion a9 above, req1 is evaluated

immediately when the assert statement is executed, just as for an immediate assertion, then gnt is evaluated on the next posedge clk event, and so on.

The '@@' token is introduced to distinguish the step control from an ordinary event control at the start of the pass statement. Consider the following:

```
always @(posedge clock or negedge rst)
  if(state == REQ)
    assert (req1)
      @(posedge clk) // This is an event control in the pass statement
        $display("Hello at time %t", $time);
```

In this example, the "@(posedge clock)" in the pass statement causes the display action to occur on the next posedge clock after the assertion succeeds. Therefore, a new token is required to distinguish the assertion sequence step control from the pass statement.

Note that, since the first expression is evaluated immediately, assertion a9 above is equivalent to

```
always @(posedge clk or negedge rst)
  if(state == REQ)
    assert(req1)
      assert(1;gnt;!req1) @@(posedge clk);
```

The sequence notation "(1;<expression_or_sequence>)" is a convenient shorthand indicating that the <expression_or_sequence> is to be evaluated on the next occurrence of the step control event. This is because the expression '1' is evaluated immediately and is always true.

Sequential assertions using the **assert** keyword are called *clocked immediate* assertions, since the expressions are evaluated as with immediate assertions. Similarly, *clocked strobed* assertions may be written using the **assert_strobe** keyword, in which each expression in the sequence is evaluated at the end of the timeslice in which the assertion is executed or in which the step control event occurs. The pass and fail statements of clocked strobed assertions have the same restrictions as strobed assertions.

Specifying an explicit step control for a sequence makes it possible to use clocked assertions in combinational always blocks:

```
always @(foo,bar)
  assert_strobe (a;b;c) @@(posedge clk); // look for a when foo or bar change,
  // then look for b on next posedge clk
```

Since it is common for combinational always blocks to be executed multiple times in a single timestep as the signals in the event trigger expression settle, it is common to use strobed assertions be used in combinational always blocks. Immediate assertions are commonly used in clocked always blocks.

Note that to avoid races, the variables read in clocked immediate assertions should be written by non-blocking assignments. Expressions in clocked strobed assertions are always sampled at the end of the timestep, so no race conditions should occur.

An assertion could be executed twice in the same timestep via a zero-delay loop or a combinational always block, for example. If a clocked immediate assertion is executed more than once at the same timestep, the first expression in the sequence will be re-evaluated. If a clocked strobed assertion is executed more than once at the same timestep, the first expression in the sequence will be evaluated once at the end of the timestep.

An assertion shall only spawn a single process to evaluate the next expression in the sequence at the next step control event. If the step control event occurs multiple times at the same timestep, then in a clocked immediate assertion the current expression in the sequence shall be re-evaluated. In a clocked strobed assertion, the current expression will still be evaluated only once at the end of the timestep. The next expression in the sequence shall not be evaluated until the step control occurs in a later timestep.

As mentioned above, the execution of a sequential assertion spawns a process that monitors each event in the sequence when the step control event occurs. If the sequential assert statement is executed again before the sequence spawned by the original execution has expired, then a new process shall be spawned that looks for the sequence starting at the current timestep. It is therefore possible to have multiple processes in-flight, each monitoring the same sequence but offset in time. It is possible for these multiple processes to be satisfied by the same sequential behavior, even though the processes are offset in time. In such a case, both processes will terminate at the same timestep, in which both sequences are satisfied. Consider:

```

module top;
  reg clk = 0;
  reg a,b,c;

  initial begin
    #10 clk = 1;
    forever begin
      clk = 0;
      clk = 1; // 2 posedges clk at 10,20,30,40...
      #5 clk = 0;
      #5 clk = 1;
    end
  end

  always @(posedge clk)
    assert(a;b;c) @@(posedge clk); // 'a' is evaluated only once at 10, 'b' once at 20, 'c' once at 30

```

Note that the step control expression may be any valid event expression in SystemVerilog. The following assertions all use valid step control expressions:

```

bit clk;
event ev1;

always @(posedge clk or negedge reset) begin
  assert (a;b;c) @@(negedge clk); // sequence sampled on negedge clk
  assert (a;b;c) @@(clk); // sequence sampled on any edge of clk
  assert (a;b;c) @@(ev1); // sequence sampled when event ev1 fires
  a10: assert(a;b;c) @@(posedge clk iff !rst); // sequence sampled on posedge clk if rst == 0
end

```

Note the use of the 'iff' operator in assertion a10 above. In effect, this allows a "gated clock" to control the assertion without the user having to declare the gated clock explicitly (please see section 8.10 of the SystemVerilog3.0 Draft 5). Because this could have significant impact on the ability of Formal Verification tools to evaluate the assertion successfully, it is recommended that this construct be used only for simulation.

This flexibility also allows nested assertions to use different clocks:

```

always @(posedge clk) begin
  assert (a;b) @@(posedge clk) // on posedge clk
    assert (1;c;d) @@(negedge clk); // look for c and d on negedge clk
  assert (e;f) @@(posedge clk2)
    assert (1;g;h) @@(ev1);
end

```

More Expression Sequences

A number of steps can be skipped either by writing expressions which are always true:

```
assert (a;1;1;c) @@(posedge clk); // two steps between a and c
```

or by using the notation [n] to count the number of steps:

```
assert (a;[2];c) @@(posedge clk); // two steps between a and c  
assert (a;[1];[1];c) @@(posedge clk); // two steps between a and c
```

Note that in [n], the n must be a non-negative literal or a constant expression. [0] has no effect. The number of steps to be skipped may also be expressed using [min:max], where the minimum number of steps must be greater than or equal to zero. Both min and max must be a literal or constant expression.

```
assert (a;[0:10];b) @@(posedge clk); // b occurs between the next and 11th clock edges, inclusive
```

If an expression must be repeated a defined number of times, this can be expressed with a trailing *[n]. If it can be repeated a minimum or maximum number of times, this can be expressed with a trailing *[min : max]. These repetition counts must also be literals or constant expressions:

```
assert (a; b)*[5]; // a;b;a;b;a;b;a;b;a;b
```

```
assert (a*[0:3];b;c); // (a;b) or (a;b;a;b)Note that (a*[0:3];b;c) is equivalent to (b;c) or (a;b;c) or (a;a;b;c) or (a;a;a;b;c). This means that a sequence a;ab;a;b;c; will pass. The expression sequence is not equivalent to ((a && !b)* [0:3];b;c), which would fail the same sequence.
```

The rules for specifying repeat counts are summarized as:

- Each form of repeat count specifies a minimum and maximum number of occurrences:
 - `expr*[n:m]` n is the minimum, m is the maximum
 - `expr*[n]` same as `expr*[n:n]`
 - `[n]` same as `1*[n:n]`
- The sum of the minimum repeat counts for all terms in a sequence must be greater than 0.
- The sequence as a whole cannot be empty.
- The last term in a sequence may not have a min:max range of repetition. If it does, it shall be an error.

Aborting Assertions Externally

A named assertion can be disabled like any other named SystemVerilog block. If this is done before the expression sequence has finished, it means that neither the pass statement nor the fail statement is executed.

```
disable cas;
```

Note that if the disable is applied at the same simulation time step as the last clock step of a sequence, there is a race in the case of an immediate assertion, but a strobed assertion is always disabled.

If the pass or fail statement is executing when the disable is executed, the statement shall be disabled, just as if the statement were in another named block that gets disabled.

If a sequential assertion has been executed multiple times before the sequence has expired, then ALL instances of the assertion shall be disabled when the assertion is disabled.

Controlling Assertions

System tasks are provided to limit assertion checking to part of the design and part of the simulation time.

The \$assertoff system task stops the checking of all specified assertions. When these assertions are encountered before a subsequent \$asserton, the assert statement shall be ignored. Neither the pass statement nor the fail statement shall be executed. An assertion that is already executing, including execution of the pass or fail statement, is not affected by \$assertoff.

The \$assertkill system task disables all specified assertions and prevents them from executing until a subsequent \$asserton. As with disable, the checking of the sequence is aborted, and neither the pass nor fail statement is executed.

The \$asserton system task re-enables the execution of all specified assertions.

The assertion control system tasks may be used with or without arguments. When invoked with no arguments, the system task refers to all assertions throughout the model.

Assertions are on by default until turned off. When an assertion control task is specified with arguments, the first argument indicates how many *levels* of the hierarchy below each specified module instance to turn on or off. Subsequent arguments specify which scopes of the model in which to control assertions. These arguments can specify entire modules or individual named assertions within a module. Setting the first argument to 0 causes all assertions in the specified module and in all module instances below the specified module to be affected. The argument 0 applies only to subsequent arguments which specify module instances, and not to individual assertions.

System Functions

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is "one-hot." The following system functions are included to facilitate such common assertion functionality:

- \$onehot(<expression>) // one and only one bit of <expression> is high
- \$onehot0(<expression>) // at most one bit of <expression> is high
- \$inset (<expression>, ,<expression>...) // The first <expression> is equal to at least one of the subsequent <expression> arguments.
- \$insetz(<expression>,<expression>...) // <expression> is equal to at least other <expression> argument. Comparison is performed using 'casez' semantics, so 'z' or '?' bits are treated as don't-cares.
- \$isunknown(<expression>) // equivalent to " \wedge <expression> == 'bx'"

All of the above system functions have a return type of bit.