SystemVerilog Assertions

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

Prof Michael Quinn
From Slides by Dr. Niels Burkhardt

Introduction

- SystemVerilog adds assertions to Verilog
- Primarily used to validate the behavior of a hardware design
- Also possible to use assertions to collect coverage
- Properties describe the relationship of signals in time

- Important: Assertions do not create stimulus for a DUV during simulation.

Assertions Types

- Immediate Assertions
  - Are executed like a statement in procedural code
  - Primarily used in simulation
- Concurrent Assertions
  - Based on clock semantics
  - Executed independently from HDL code
  - Used in simulation and formal

Immediate Assertions

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

- Are evaluated when the assertion statement is executed in procedural code
- Immediate assertions are non-temporal
- Acts like procedural “if” statements
- Evaluates to false if the assertions expression returns X,Z, or 0
- Syntax:

```
[label :] assert ( expression ) action_block
action_block ::= statement_or_null | [statement] else statement
```

- action_block specifies what actions are taken upon success or failure of the assertion
- Example:

```
always @(posedge clk)
        if(state == REQ)
                assert(req1 || req2)
                else $error(“No request is pending in REQ state”);
```

Concurrent Assertions

- Describe a behavior that spans over time
- Need a reference clock for evaluation
  - Are only evaluated at this clock
- Evaluation is done in an extra time slot of the simulator after all variables are assigned
- Can be specified in
  - a module
  - an interface
  - a program block
- Syntax:

  [label:] **assert** **property**(property) [[pass_stat] **else** fail_stat];

Properties

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Properties are the building blocks for concurrent assertions
- Properties define the designs behavior in time
- A Property can be used as
  - Check (**assert property**())
  - Assumption (**assume property**())
  - Coverage (**cover property**())
- Can be declared in
  - a module
  - an interface
  - a program block
- Example

```
assert  property(  @(posedge  clk)  disable  iff(!res_n)
          (s1  &&  s2)  |=>  s3);
```

Property Syntax

property name[(list_of_formals)];
    property_spec;
endproperty [: name]

property_spec ::= [clocking_event][**disable** **iff**(expression)]
property_expression

property_expression ::=
    sequence_expr
    | (property_expression)
    | property_operator
    | **if** (expression) property_expression [**else**
property_expression]
    | property_instance
    | clocking_event property_expr

Property Operators

- Not
  - **not** property_expression
- Or
  - property_expression **or** property_expression
- And
  - property_expression **and** property_expression
- Implication
  - Overlapping implication
    - property_expression **|->** property_expression
    - If there is a match for the antecedent property_expr, then the end point of the match is the start point of the evaluation of the consequent property_expr.
  - Non-Overlapping implication
    - property_expression **|=>** property_expression
    - The start point of the evaluation of the consequent property_expr is the clock tick after the end point of the match.

Property Example

```
wire  s1,  s2,  s3;

property  p1;
          @(posedge  clk)  disable  iff(!res_n)
                   (  s1  &&  s2  )  |=>  s3;
endproperty  :  p1

assert  property  (p1);
```

A short form:

```
assert  property(@(posedge  clk)  disable  iff(!res_n)
(  s1  &&  s2  )  |=>  s3);
```

Sequences

- More complex properties can be constructed out of 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.
- An example sequence:

```
a   ##1   b   ##1   c
```

- The first clock tick "a" must be true, the second one "b", and at the last clock tick "c". The whole sequence fails, if one of these conditions fail.

Sequence Operators

- **Delay**: a ##n b, a ##[n:m] b
  - The delay operator specifies the number of clock ticks from the current clock tick until the next behavior occures. Beside a constant value, it is possible to specify an range of clock ticks. This is done with the range operator ([:]). An open range is specified by a $ character.
  - Example: a ##[3:$] b

- **Consecutive repetition**: a[*n], a[*n:m]
  - The consecutive repetition specifies finitely many iterative matches of the operand sequence, with a delay of one clock tick from the end of one match to the beginning of the next match. The overall repetition sequence matches at the end of the last iterative match of the operand.
  - Example: a[*3] equals a ##1 a ##1 a

Sequence Operators

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

- **Goto repetition**: a[->n], a[->n:m]
  - The goto 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.
  - Example: a[->1] equals (!a)[*0:$] ##1 a
- **Non-consecutive repetition**: a[=n], a[=n:m]
  - The non-consecutive repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at or after the last iterative match of the operand, but before any later match of the operand.
  - Example: a[=1] equals (!a)[*0:$] ##1 a ##1 (!a)[*0:$]

Sequence Operators

- **And** Operator: a and b
  - The "and" operator matches, if both sequences match, and the start time of both is the same. The end time can be different.
- **Or** operator: a or b
  - The "or" operator matches, if one of the sequences match.
- **Intersect** operator (AND with length restriction): a intersect b
  - The "intersect" operator matches, if both sequences match and the start and end time is the same.
- **First_match** operator: first_match(a)
  - The "first_match" operator matches only the first of possibly multiple matches for an evaluation attempt of its operand sequence.
- **Throughout** operator: a throughout b
  - The "throughout" operator matches, if a is true during the whole match of b. It is an abbreviation for: (expr)[*0:$] intersect b
- **Within** operator: a within b
  - The "within" operator matches, if b matches and a is true at some point of this interval.

Local Variables

- SystemVerilog supports the declaration of local variables inside properties and sequences.
- They are used to pass data for one stage in a sequential expression to a later stage.
- The following data types are supported as local variables:
  - bit, byte, int, integer, logic, reg, time, packed struct, class, arrays of supported types
- An example for local variables:

```
property adder;
    bit [63:0] x;
    @(posedge clk) disable iff (!res_n)
        (valid, x = data_in) |=> ##5 (data_out == (x+1));
endproperty : adder
```

System Functions

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

- Assertions are commonly used to evaluate specific characteristics of a design.
- System functions are available to simplify the evaluation.
- The following functions are available:
  - **$onehot**(<expression>)
    - returns true if only one bit of the expression is high
  - **$onehot0**(<expression>)
    - returns true if at most one bit of the expression is high
  - **$isunkown**(<expression>)
    - returns true if any bit of the expression is X or Z. This is equivalent to ^<expression> = = = ’bx.
  - **$countones**(<expression>)
    - returns the number of 1s in the expression.

Assertion Writing Guidelines

Minimize the number of attempts:

- Properties with an enabling condition, that is true a lot of times, slows down the simulator a lot.

```
// "slow assertion"
property bad;
@(posedge clk) disable iff(!res_n)
  valid && enable |-> a_very_long_sequence;
endproperty : bad
```

Use instead:

```
property good;
  @(posedge clk) disable iff(!res_n)
    $rose(valid) && enable |-> a_very_long_sequence;
endproperty : good
```

Assertion Writing Guides

Minimize false starts:

- Try to start sequences or property enabling conditions with a condition that is rarely true.

```
sequence   bad;
  a   ##1   b   ##2   c;
endsequence   :   bad
```

- If b is rarely true and a is true very often, a better solution is the following one:

```
sequence   good;
  ($past(a)   &&   b)   ##2   c;
endsequence   :   good
```

Example

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

```
wire   data_valid,any_sox,any_eox;

sequence   length_of_packet;
      ##[1:32]   ##1   any_eox;
endsequence   :  length_of_packet

property   legal_data_valid;
    @(posedge   clk)   disable   iff(!reset_n)
          (data_valid   &&   $rose(any_sox))   |->
              data_valid   throughout   length_of_packet;
endproperty   :  legal_data_valid

chk_data_valid   :  assert   property(legal_data_valid);
```

Bind Statement

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

- SVA can be defined either directly in the HDL code, or in an own Verilog module, which is then binded to a design instance.
- This allows an easy reusability of SVAs.
- The bind statement can be used in
  - Verilog code, e.g. in the testbench
  - a bind file, which is loaded by the simulator
- Syntax:

  bind module_type : hdl_instance module_with_assertion instance_name port_list

Bind Example

```
module assertion_module(
    input logic clk,
    input logic res_n,
    input logic req,
    input logic gnt
);
    req_gnt : assert property(
        @(posedge clk) disable iff(!res_n) req |=> gnt);
endmodule

module tb_top;
    wire clk_top, res_n_top, req_top, gnt_top;
    dut dut_I( // of type dut
        .clk(clk_top), .res_n(res_n_top),
        .req(req_top), .gnt(gnt_top)
    );
    bind dut : tb_top.dut_I assertion_module assertions_I (
        .clk(clk), .res_n(res_n), .req(req), .gnt(gnt)
    );
endmodule
```