SVA - Summary

Contents

[Introduction 2](#_Toc160801420)

[Immediate assertions 2](#_Toc160801421)

[Casting 2](#_Toc160801422)

[Static casting 2](#_Toc160801423)

[Dynamic casting 2](#_Toc160801424)

[Randomization 3](#_Toc160801425)

[Concurrent assertions 3](#_Toc160801426)

[Assertion example 3](#_Toc160801427)

[Separate proprieties and assertions 4](#_Toc160801428)

[Combined proprieties and assertions 4](#_Toc160801429)

[Macros and assertions 4](#_Toc160801430)

[Good practices 4](#_Toc160801431)

# Introduction

There are 2 types of assertions:

* Immediate assertions execute once and are placed in line with the code. They are useful in very few situations.
* Concurrent assertions activate properties that typically sample design signals just before each new active clock edge to determine if the design is behaving as it was claimed that it should behave. They are either placed directly in the RTL code or are bound to an RTL file using the bindcommand. They are the most widely used type of assertion.

Assertion can be control using multiple dynamic methods:

* $assertoff: disable all assertions but allows currently active assertions to complete before being disabled.
* $asserton: turn all assertions back on
* $assertkill: kill and disable all assertions including currently active assertions.

# Immediate assertions

Immediate assertions do not offer broad assertion usage value, but they do offer a concise and convenient form to test the success of dynamic casting and constrained randomization of variables.

## Casting

### Static casting

Static-Speed casting is used when engineers are interested in higher performance simulations and are confident that all of the casting in their code was type and boundary-safe. Static casting is faster and more efficient than dynamic casting.

The SystemVerilog static cast uses a data type to cast an argument enclosed within parentheses to the new data type and then assigns the cast-modified value to a target of the same or compatible type.

int x = int’(4.5);

### Dynamic casting

Dynamic casting is used when engineers are concerned about type and boundary checks. It is slower and less efficient than static casting.

The SystemVerilog dynamic cast system call takes two arguments. The dynamic cast assigns the second argument to the first argument and in the process of making the assignment, converts the second argument into the type of the first argument.

The dynamic cast used as a system task has no return value and causes a run time error.

The dynamic cast used as a system function returns true or false. Using the “if-error-display-message” coding style, immediate assertion can be used to replace the function and display.

ERROR\_what\_happened: assert ($cast(x, 4.5)); $display (“x = %0f”, x);

## Randomization

Immediate assertions are used to check whether the randomization completed successful or not.

assert (packet.randomize());

# Concurrent assertions

The most valuable assertion style that can be used in design and verification environments is the concurrent assertion. Concurrent assertions are little monitors that sit down inside of a block of code to periodically sample and test signals and generate error messages if the assertion ever fails.

Concurrent assertions are typically sampled once per clock period at the end of the clock cycle, just before the next active clock edge. Concurrent assertions require the assertion of a property. A property is a design rule that should always be true.

The property definition requires a sampling signal that is:

* Explicitly listed in the property definition;
* Inherited from a default clocking block definition.

The property definition can also specify a condition under which the property is disabled. Each property can be declared individually and then separately asserted. The separate property declarations can be grouped into a library of properties to be asserted on multiple designs.

## Assertion example

`define assert\_clk( arg ) \

assert property (@(posedge clk) disable iff (!rst\_n) arg )

ERROR\_q\_did\_not\_follow\_d:

`assert\_clk(q==$past(d));

An assertion is made of:

* assert property - keywords to start the definition of an assertion.
* ( arg ); - placeholder for the assertion.
* @(posedge clk) - sample signal for the concurrent assertion.
* disable iff (!rst\_n) - definition of when the assertion should become inactive.
* q==$past(d) - actual assertion test.

## Separate proprieties and assertions

This technique includes the declaration of separately named properties, followed by the assertion of the named properties. This technique has merits especially for verification teams that intend to construct a large set of reusable properties that can then be used by others on the project team. It is not intended for simple design specific assertions.

property example\_property;

@(posedge clk) disable iff (!rst\_n)

(something -> something);

endproperty

ERROR\_EXAMPLE\_PROPERTY:

assert property (example\_property);

## Combined proprieties and assertions

This technique includes the declaration of asserted properties without separate declaration of named properties. This is a more common coding style, but still too verbose for the average simple design.

ERROR\_EXAMPLE\_PROPERTY:

assert property (@(posedge clk) disable iff (!rst\_n)

(something -> something));

## Macros and assertions

This technique is used by defining a couple of simple, yet powerful, macros that can reduce the coding effort required to add assertions to the typical design.

`define assert\_clk(arg) \

assert property (@(posedge clk) disable iff (!rst\_n) arg)

ERROR\_EXAMPLE\_PROPERTY:

`assert\_clk (something -> something);

Example of a more complex macro definition:

`define assert\_clk(arg, enable\_error=0, msg="") \

assert property (@(posedge clk) disable iff (!rst\_n) arg) \

else if(enable\_error) $error("%t: %m: %s", $time, msg)

ERROR\_Q\_DID\_NOT\_FOLLOW\_D: assert\_clk ((q==$past(d)), 1, "\*\*\*ERROR!!\*\*\*");

# Good practices

* Always use long and descriptive labels for concurrent assertions. The long labels help to debug the assertions in a waveform display.
* The macro and assertions technique requires the least effort and keeps the code clean.