OVL Usage Guidelines

Table of Contents

1. Introduction

The OVL library is a collection of standard assertions, which can be instantiated in a design to increase observability of errors during verification, and provide targets during formal verification. They also provide good documentation of design intent.

This file contains guidelines for writing and verifying OVL assertions, to promote consistency between users and EDA tools. It has been written primarily for the Verilog version of OVL, but is applicable to other language implementations.

1.1. Reference Documentation

There are 3 forms of useful reference documentation for OVL:

- Language Reference Manual
- OVL Quick Reference
- OVL Timing Diagrams

1.2. Online Resources

There are separate websites for the OVL technical committee and the OVL users group.

- www.accellera.org/activities/ovl
- www.verilog.org/ovl

2. Writing OVL Assertions

This section describes some guidelines for writing OVL assertions.

2.1. Locate at end of Module

OVL assertions, and any auxiliary logic (see section 2.4), should be located at the bottom of the Verilog module that instances them. They should all be enclosed by a single `` `ifdef `` control so that they have to be explicitly enabled (avoids being synthesised, regardless of how the OVL modules are defined). This structure is illustrated in Figure 1.

```
module <name>
  // Verilog for design
  <...>

`ifdef OVL_ASSERT_ON
  `include "std_ovl_defines.h"
  <all OVL instances & auxiliary logic>
`endif
```

Figure 1: Location of OVL assertions

Assertions should also be located in the most local module that's possible, in order to avoid out-of-module references. However, out-of-module references will be needed for OVL assertions that check logic across several modules (these must be located in a module high enough in the hierarchy to be able to refer down to all these).

See section 2.6 for the location of interface assertions.

2.2. OVL Ports and Parameters

This section contains some coding guidelines for instantiating an OVL assertion.

2.2.1. Named Ports

A good coding guideline for instantiating any Verilog module is to use named ports rather than positional ports. This is equally applicable to OVL assertions, so you should use e.g. `".start_event (req), .test_expr(ack)"` rather than just `"req, ack"`. Positional ports are more concise, but are error prone.

2.2.2. Clock and Reset

Avoid using functional signals on the `.clk` and `.reset_n` ports, which should only be driven by clock and reset signals. The only exception is for assertions that contain internal state requiring a functional reset e.g. `assert_fifo_index` may need to be reset by a `fifo_flush` design signal.

2.2.3. Parameters

There are several coding guidelines for OVL parameters, including:

- Instantiate all parameters, at least up to (and including) the `msg` parameter
- Set `severity_level` to indicate the severity of the failure, with `OVL_FATAL used to stop simulations
- Set `property_type` to `` `OVL_ASSUME `` for constraints on inputs e.g. no back-to-back requests
- Set the `msg` parameter to a meaningful message, not just `"FAILED"`
- Group related assertions by using a common prefix in the `msg` parameter

2.3. Explicit X Checking

In addition to all OVL assertions checking for X (unless disabled by `OVL_XCHECK_OFF`), there are two explicit X checking assertions:

- assert_never_unknown
- assert_never_unknown_async

2.4. Auxiliary Logic

Auxiliary Verilog can sometimes be required to simplify the OVL assertions. This could be Verilog tasks/functions, or additional storage e.g. a sampled request input with a one-cycle delay could be called `req_d1`.

Auxiliary logic should be located with the OVL instances, disabled by the same `` `ifdef `` to avoid it being compiled during synthesis (see section 2.1).

Any storage in auxiliary logic should be reset to avoid an additional source of X. All DFFs should be reset, as this logic is not synthesised into the design (so does not need to be minimized with non-reset DFFs).

2.5. Input Assumptions

It's important to distinguish OVL assertions that are checking input constraints, e.g. possible Cache size encodings, for formal verification (to ensure that only legal input sequences are applied). To do this you can set the `property_type` parameter to `` `OVL_ASSUME ``.

2.6. Interface Checking Modules

For interfaces between blocks, a structured approach is necessary (to avoid missing assumptions, or having conflicting assumptions and assertions). As bugs are often

found on the interfaces of blocks, it's well worth putting effort into interface assertions. A good approach is to create separate interface modules that only contain interface related OVL assertions, with one or more of their own `property_type` parameter to indicate direction of assertions e.g. set to 1 to assume, or to 0 to prove. Having a common definition of an interface protocol is a good way to avoid misunderstandings between designers of two blocks.

An interface module can be instanced either:

- Once: In the top level module
- Twice: In both the driver module and the receiver module

The advantage of instancing one module twice is that the receiver block contains a description of how it's inputs can be driven – which is good for IP reuse (of that block in another system). If you make property_type a parameter of the interface module, you can control the direction of the checks when you instance it. To avoid duplicated checks in simulation, you can use `ifdef OVL_ASSUME_ON to avoid rechecking in the receiving block. This is illustrated in Figure 2 below, where the common "a2b" interface module is configured for checking (in block_a) and assumptions (in block_b).

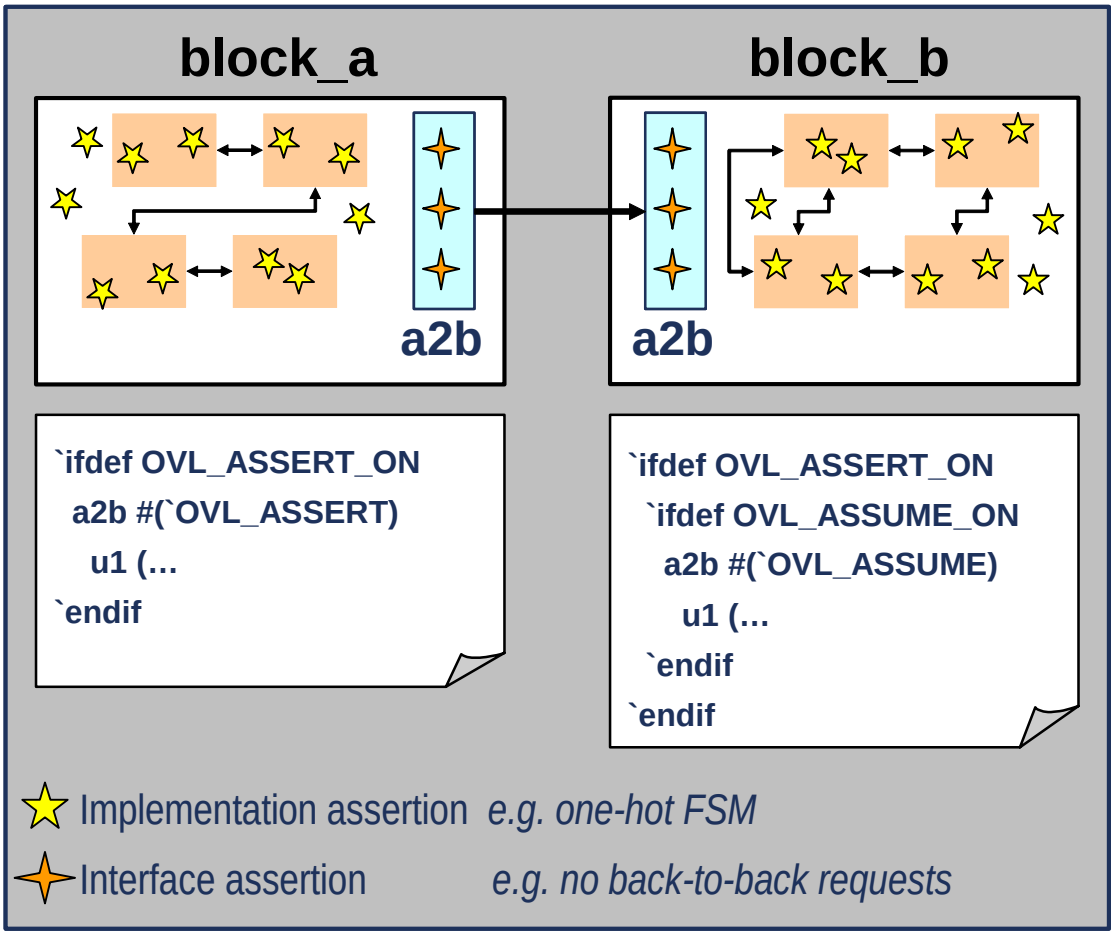

Figure 2: Interface Checking Modules

3. Verification of OVL Assertions

3.1. Compilation

Figure 3 illustrates a *verilog command file* in order to compile the OVL assertions for running in simulations. The commented-out defines for advanced initialisation messages are discussed in later sections.

```
+libext+.v+.vlib

// Switch on OVL
// =============
+define+OVL_ASSERT_ON

// Limit Messages (for each OVL instance)
// ==============
+define+OVL_MAX_REPORT_ERROR=2

// Initialisation Messages (uncomment 2nd line for count-only)
// ========================
+define+OVL_INIT_MSG
//+define+OVL_INIT_COUNT=tbench.ovl_count

// X-Checking (uncomment out to disable)
// ==========
//+define+OVL_XCHECK_OFF

// OVL Library
// ===========
-y /<...>/accellera/ovl/verilog

// Include OVL Task (can be customized)
// ================
+incdir+/<...>/accellera/ovl/verilog

// Compile Design
// ==============
<...>
```

Figure 3: Verilog Command File

3.2. Messages

3.2.1. Initialisation Messages

With just `OVL_INIT_MSG` defined (as per Figure 3), you will get initialisation messages at the start of simulation – an example of which is illustrated by Figure 4.

```
# OVL_NOTE : ASSERT_NEXT initialized @
tbench.DUT.arb_rule1.ovl_init_msg Severity: 1, Message:
ARB_Rule1 - Should only have one grant.
```

Figure 4: Initialisation Message

A design can contain hundreds or thousands of assertions, each of which will give an initialisation message. Extra controls exist to display the total number of OVL assertions initialized and enable/disable the individual messages. For instance, you can get the total number of OVL reported (as illustrated in Figure 5).

```
# ===========
# OVL_METRICS : 487 OVL assertions initialized
# ===========
```

Figure 5: Initialization Metrics

To get the `OVL_METRICS` line illustrated in Figure 5 you need to `` `include `` the `std_ovl_count.h` file in your simulation test-bench module. You also need to define both `OVL_INIT_MSG` and `OVL_INIT_COUNT` (see commented out lines in Figure 3).

3.2.2. Activation Messages

An assertion will fire with the appropriate message, e.g. an error will report:

```
#OVL_ERROR : ASSERT_NEXT : ARB_Rule1 - Should only have
one grant : severity 1 : time 400 ns :
tbench.DUT.arb_rule1.ovl_error
```

Figure 6: OVL Error Message

3.3. Checking for X/Z

Prior to version 1.7, only four OVL assertions had X/Z checking in addition to their specific functional checks. Since version 1.7, all OVL assertions can now also fail due to X/Z in an intelligent way (e.g. assert_implication will only not fail when consequent_expr is 1'bX but antecedent_expr is 1'b0).

You can disable all X checking by defining the macro: `OVL_XCHECK_OFF`. This can be useful if you wish to debug functional failures before any X issues.

3.4. Functional Coverage Points

The OVL library contains built-in functional coverage points, which are off by default but can be turned on by defining `OVL_COVER_ON`. It is also possible to control the amount of coverage reported by each OVL instance, by setting the `coverage_level` parameter.

4. FAQs

This section attempts to answer some frequently asked questions

1. How many assertions should I write?

- Unknown!
- Depends on application and requirements spec.

2. When should I add an assertion?
   - If there's additional information, e.g. about the environment or design intent.
   - DON'T simply repeat the RTL verbatim!

3. Which assertions should I use?
   - Virtually any!
   - Keep it simple (`assert_always` & `assert_never` are the most common)
   - Take care with non-default parameter values
   - Avoid `assert_proposition` (better to use clocked `assert_always`)

4. Where can I use assertions?
   - One hot state machines
   - Counters
   - Bus protocols
   - FIFOs (always a good source of errors)

5. Is OVL a language?
   - No – it's a *library* of predefined assertions.
   - OVL is currently implemented in Verilog, SVA, and PSL.

6. Which EDA tools support OVL?
   - Almost all tools should at least support an HDL version of OVL

5. OVL Assertion Subtleties

This section highlights some of the OVL subtleties to watch for when using OVL assertions.

- **General: Be careful about clock and reset inputs to assertions.** All assertion (other than assert_proposition) require a valid clock; a common user error is to not connect a clock available at the same hierarchy level as the assert. Also, many OVLs do not behave properly if they do not have a real reset signal that goes to 0 at some point while the clock is running. So an assertion with reset tied to 1 or to a bogus signal may never get triggered.

- **General: Be careful about level-sensitive vs edge-sensitive.** Most assertions are checked on positive edges of the clock, but are level sensitive with regard to all other signals involved. If you want an assertion to trigger only on the rising edge of a condition, you will often need to keep a flopped version of your signal, and check conditions on `(sig & !sig_ff)`.

- **assert_proposition**: **Avoid using this assertion.** Since it's not clocked, its internal X-check has to be triggered by an always @(test_expr). Which means that if there is a glitch, an intermediate value may be checked in the middle of a cycle, and a bogus error reported.
  - So, if you're in a design with a clock available, it's probably best to try to use assert_always (same concept as assert_proposition, but with a clock input) instead.

- **assert_always_on_edge, and other assertions with edge-triggered aspects**: This assertion checks that the condition is met immediately **after** the given edge, rather than before. In other words, the condition that triggers the error is: (!sampling_event_prev) && (sampling_event) && (!test_expr).

- **assert_frame**: The documentation of this one is confusing on one point: an assert_frame is violated unless test_expr has a **rising edge** during the prescribed number of clocks. So, if test_expr might already be 1 at the start of the time frame, the assertion will actually be incorrect. Also, if the rising edge happens before the min_cks value by coincidence, the assertion fails-- a second rising edge within the specified time frame does not rescue it.

- **assert_next, assert_time, and other assertions with an option that might check overlaps**: Sometimes these assertions internally turn into two assertions in FPV tools: one that checks the main condition, and an _overlap version that checks for overlaps if not allowed.

- **assert_next and overlaps**: assert_next is a bit non-standard in that if the overlap_flag is 0, and the overlapping condition occurs, the assertion will fire. Since it is level-sensitive, this also means the assertion will automatically fire if the starting condition is held more than a cycle.

- **assert_handshake** is rather complex, and generates a group of sub-assertions that are proven. So in your FPV runs you may see separate answers for _multiple_req_handshake, _min_ack_cycle_handshake, etc.

- **assert_no_{over,under}flow**: The assertion waits until the max/min is hit, then starts checking that the value does not increase/decrease. So, if your test expression is potentially changing by a value >1 so it may not hit the max/min exactly, this assertion will not work. (You might be better off with a simple 'assert_never ... (...val > max)').

- **assert_width**: The default value for min_cks and max_cks is **1**, not 0 (unchecked) as one might expect. So be careful if not specifying all parameters.