Assertions –
A Practical Introduction for HDL Designers

Webinar
Agenda

• Introduction
• Why to Use Assertions
• Where to Use Assertions
• Basic Terms and Ideas
• Languages Supporting Assertions
• Practical Examples of Sequences, Properties, Assertions and Covers
• Assertions in the Simulator – Live Demonstration
• Questions and Answers Session
**Why Assertions**

- Assertions are already popular in ASIC design and will appear in typical FPGA designs soon.
- Although relatively new, they are governed by IEEE standards (PSL, SystemVerilog, VHDL).
- Assertions are relatively simple (once you learn the basics).
- Assertions are based on design specification.
- Assertions create an additional layer of safety in simulation (while you concentrate on better synthesis/implementation, they are reminding you about the original specs).
- Assertions create ‘live documentation’ (better documentation makes management of your design easier).
Where to Place Assertions

1. All tools let you place assertions in separate units and *bind* them to your regular RTL code.

   Verification Engineers like this, HDL Designers – not so much…

2. Good simulators let you place assertions directly in the RTL code, hidden in *special comments*:

   ```-- psl property p1 is... or // psl property p2=...```

3. SystemVerilog assertions can be placed *directly* in the Verilog code.

   Solutions (2) and (3) are perfect for HDL Designers …

4. Once VHDL 2008 is implemented in more tools, users will be able to place PSL assertions *directly* in the VHDL code.

   ① and ② are perfect if you think that assertions will harm your synthesis flow.
   ③ and ④ would require the use of synthesis pragmas around assertions.
Assertions and Synthesis Tools

• Most popular, regular versions of synthesis tools cannot handle assertions yet. (Exceptions to this rule include some high-end tools like Synplicity’s Identify Pro + Synplify Premiere combo.)

• Some methods of placing assertions in the design mentioned in the previous slide should make managing assertions in mixed simulation/synthesis flow pretty smooth.

• Libraries of ready-to-use property checkers written in VHDL or Verilog may be a solution here. Example: Open Verification Library (OVL) maintained by Accellera started introducing synthesizable versions of some checkers in most recent releases of the library.
KEY TERMS
Key Ideas

- **Assertion Based Verification** deals with significantly more ideas than just assertions:
  
  sequences → properties → assertions/covers

- The very basic idea of ABV is **property**: formalized description of certain behavior of your design
  
  e.g. “broken window triggers alarm”, “security responds to alarm in 30 seconds”.

- Properties can be used by verification tools in many ways:
  
  - **assert** `<property>` verifies that **bad things do not happen**.
    e.g. “assert [that] broken window always triggers alarm”.
  
  - **cover** `<property>` verifies that **good things happen**.
    e.g. “cover response to triggered alarm in 30 seconds”.
The World of Properties

- Typical digital design specification is full of design properties expressed in plain English.
- Designer rewrites properties in HDL with correct hardware implementation in mind.
- Properties, assertions and covers represent pure behaviors (desired or undesired) of the design:
  - They can be very efficient documentation of the design.
  - They work as a reference during design verification.
  - They are accepted by variety of functional and formal verification tools.
The Logic of Properties

• Formal properties use principles of **temporal logic**: Boolean logic with added time dimension.

• If we use **discrete time**, then properties represent **sequences of states** of the design.

**NOTE:** all popular PBD/ABV solutions operate on sampled values of objects in the design.

• To express relationships in time (existence, succession, etc.) properties use **temporal operators**: *next*, *finally*, *globally*, *until*, etc.

**HOMEWORK:** to learn more, google “linear temporal logic” or “LTL”…
Building Blocks of Properties

- **Boolean expressions** as we know them from HDLs are the part of properties, but rather a simple part.

- **Sequences** are generally recognized as the basic **temporal building block** of properties: they represent succession of design states seen at discrete time points. Typical sequence represents one simple execution path in the design.

- You can:
  - **fuse** sequences (one sequence ends at the same moment the other begins),
  - **concatenate** sequences (one sequence ends and the other starts at the next time point),
  - say that one sequence is an **implication** of another sequence,
  - **and** or **or** sequences,
  - check if one sequence is contained **within** another sequence,
  - check if sequence **repeated** given number of times (consecutively or not).
Using Properties

- Once design property was formalized, all tools can use it in one of two directives:
  - `assert` – raises alarm when property does not hold,
  - `cover` – confirms that property was successfully tested.

- Some tools (mainly formal verification, but also some simulators) allow more directives, e.g. to control design stimulus or restrict environmental conditions.
PROPERTY LANGUAGES
There are two languages with practical applications of expressing properties, assertions and covers:

- **PSL** (*Property Specification Language*):
  - standard IEEE Std 1850™-2005,
  - comes in VHDL, Verilog, SystemVerilog and SystemC *flavors*,
  - its *Simple Subset* is a part of the recent version of VHDL standard (IEEE Std 1076™-2008).

- **SVA** (*SystemVerilog Assertion subset*):
  - an assertion-related subset of the SystemVerilog language (initially Accellera project, then standard IEEE Std 1800™-2005),
  - based on *Superlog* and *OpenVera* donations,
  - its properties/assertions features also borrow from PSL.

✅ **ALL property languages are IEEE standards!**
Which Language?

- **VHDL** designers can use both SVA and PSL, but they will feel better with **PSL**:
  - PSL can be placed directly in VHDL code (live documentation) but SVA cannot,
  - PSL is the part of the latest VHDL standard.

- **Verilog** designers can use both PSL and SVA, but they will feel better with **SVA**:
  - SVA placed directly in Verilog code can do more than PSL placed directly in Verilog code,
  - SystemVerilog and Verilog are in the process of merging into one standard.

*Good News:* PSL and SVA properties look almost the same…
BUILDING PRACTICAL ASSERTIONS
Some ‘Sequence’ Property Facts

The simplest sequence is a Boolean expression, but more frequently you glue together several expressions to form complex sequence that ‘spreads in time’:

- To make elements stick together at the same clock use **fusion** (: ) in PSL or **zero cycle delay** (##0) in SVA.

- To introduce one cycle break between elements use **concatenation** (:) in PSL or **one cycle delay** (##1) in SVA.

- You can specify longer delays using range of values (##[m:n]) in SVA.

- PSL uses **consecutive repetition operator** (<sequence>[!*m to n]) to specify ‘range of values’ delay: if there is no sequence to repeat, it is assumed that True is the argument of repetition.
Repetition

- If the same condition should hold for more than one cycle, we can use **consecutive repetition operator** instead of repeating the condition using concatenation or one cycle delay.
- The operator looks identical in PSL and SVA:
  
  Sequence \[ \text{[} \ast \text{ Number_or_Range] \] }

- Simple form with number says that the sequence should be repeated (hold) given number of times:
  \[ A \text{[}\ast 7 \text{]} \]

- Range version says that the sequence should hold for any number of cycles within the natural range (to specify infinite upper bound, use \textit{inf} in PSL or $ in SVA):
  \[ B \text{[}\ast 1 \text{ to inf}] \quad B \text{[}\ast 1 : \$] \]
**Clocking Sequences and Properties**

All property languages use *discrete time*; it means that *clocking* (or *sampling*) method must be specified.

- If no clocking is specified, the fastest available *tool default* is applied (in simulator it may be the clock with period equal simulation resolution).

- If one clocking method is used in the majority of properties, user can specify *default clock*.

- To specify clocking event, use `<clock condition>` at the end of sequence in PSL or at the beginning in SVA.

- For clock condition, use clock detector preferred in the native HDL, e.g. `rising_edge` function in VHDL or `negedge` in Verilog.
How About Resets?

• Sometimes global signals (e.g. reset) can happen in the middle of property evaluation, causing assertion failure or other unwanted situation.

• Both PSL and SVA provide mechanism for abandoning property evaluation without adverse effects:
  - **PSL** lets you add `async_abort` or `sync_abort` operator and reset condition at the end of property:
    ```verilog
    always (({A};(B)}) async_abort C='1') @rising_edge(CLK);
    ```
  - **SVA** lets you add `disable iff (condition)` phrase at the beginning of the property:
    ```verilog
    @(posedge CLK) disable iff (C) A ##1 B;
    ```
Importance of Implication

• Property languages support *temporal implication* in its standard form:

  Antecedent  Implication Symbol  Consequent

• Property with implication has three possible behaviors during verification:
  - Vacuous pass (antecedent is false),
  - Non-vacuous pass (antecedent and consequent are true),
  - Fail (antecedent is true, consequent is false).

• Vacuous pass is crucial in efficient simulation of assertions with longer sequences:
  - if *trigger event* specified as antecedent does not happen, the remainder of the sequence is not evaluated and assertion passes without alarm.
**Flavors of Implication**

- **Overlapping implication** \( \text{(sequence } \mid \rightarrow \text{ property) and non-overlapping implication} \ (\text{sequence } \mid =\Rightarrow \text{ property)} \) operators are common to PSL and SVA.

- Consequent of overlapping implication starts at the time point when antecedent ends.

- Consequent of non-overlapping implication starts one cycle after the end of antecedent.

- PSL has regular logical implication \( \text{(condition } \rightarrow \text{ property)} \) showing ‘overlapping’ behavior. In some cases it can replace sequence implication, but beginners should stick to \( \mid \rightarrow \) …
Assertion with Implication

Let’s approach typical design property “ACK should be activated 1 to 3 cycles after REQ provided that RESET is not active” using PSL *(shaded box)* and SVA *(framed box)*.

Note the assertion directive that follows property definition.

```plaintext
property req_ack is
  always(
    ( {REQ='1' } |-> {ACK='0' }[*1 to 3]; (ACK='1' } )
  async_abort RESET='1' )@rising_edge(CLK);
assert (req_ack) report "No timely ACK after REQ!";

property req_ack;
  @(posedge CLK) disable iff (RESET)
    (REQ=1) |-> (ACK=0)[*1:3] ##1 (ACK=1);
endproperty
assert property(req_ack) else $error("No timely ACK after REQ!");
```
Benefits of Edge Detection

• The assertion from the previous slide has one disadvantage: if REQ is high for more than one clock cycle, the property starts new evaluation thread on each clocking event, which may lead to:
  ♦ Redundant multiple violation messages when assertion fails,
  ♦ Unnecessary slowdown of simulation.

• To avoid this situation, edge detection functions can be used:
  ♦ rose() / fell() in PSL/VHDL,
  ♦ $rose() / $fell() in PSL/Verilog and SVA.

• If edge detection is not convenient, more selective condition should be selected for implication antecedent.
Improved Assertion

Let’s rewrite the property from the previous example to make the assertion more efficient. This time we use edge detection functions – please remember that their results depend on sampled values of arguments!

```verilog
property req_ack is
    always( ( {rose(REQ)} |-> { (ACK='0'[*1 to 3]; rose(ACK)} )
        async_abort RESET='1' )@rising_edge(CLK);
    assert (req_ack) report "No timely ACK after REQ!";
endproperty
```

```verilog
property req_ack;
    @posedge CLK disable iff (RESET)
        $rose(REQ) |-> (ACK==0)[*1:3] ##1 $rose(ACK);
endproperty
assert property(req_ack) else $error("No timely ACK after REQ!");
```
Properties in Functional Coverage

The use of implication in coverage properties may trigger false results. Old, good, plain sequences will do fine in this case of state machine with state register M1...

```vhdl
sequence std_seq(const max) is
  (M1=init); (M1=idle)[*1 to max]; (M1=act1)[*1 to max];
  (M1=act2); (M1=act3)[*1 to max]; (M1=idle);
property slow_path is std_seq(4)@falling_edge(CLK);
cover (slow_path) report "Slow execution path covered!";

sequence std_seq(max);
  (M1==INIT)##1(M1==IDLE)[*1:max]##1(M1==ACT1)[*1:max]##1
  (M1==ACT2)##1(M1==ACT3)[*1:max]##1(M1==IDLE);
endsequence
property slow_path; @(negedge CLK) std_seq(4); endproperty
cover property(slow_path) $warning("Slow execution path covered!");
```

The use of parameters promotes reuse...
SIMULATION OF ASSERTIONS AND COVERS
Sample Workspace

- We are going to see the workspace with two designs:
  - VHDL design with PSL assertions embedded in the comments,
  - Verilog design with SVA assertions embedded directly in the design code.
- Both designs have identical functionality (counter modulo 31) and use identical naming of ports and signals.
- Properties, assertions and covers were written to highlight similarities between PSL and SVA (we could tweak them using features unique to each language, but we didn’t want to torture the audience…)

www.aldec.com
Properties Used

- **Property “reset_length”:** Asynchronous RESET should be active for at least 3 clock cycles. (We are asserting this property.)
- **Property “roll_back”:** While counting, terminal count (30) should be followed by zero count. (We are asserting this property.)
- **Property “reset_tst”:** Simulation should test non-initial reset, i.e. reset from non-zero count. (We are using this property in a cover statement.)
WRAP-UP
Summary

- **Properties, Assertions** and **Coverage** are unavoidable: even if your designs are not using them now, *they will* – sooner or later.
- Don’t be afraid of assertions and coverage.
- Don’t worry which language to choose.
- Check what your tool supports.
- Get yourself a good book or decent training.
- You should be ready to go in one day!
- Attaining expert level will take a little bit longer…
Resources

- Temporal Logic in software verification:
  http://www.cis.ksu.edu/santos/bandera/Talks/Bandera-Course-3-TL.ppt

- PLS/Sugar resources:
  http://www.pslsugar.org/technical_papers.html

- PSL 1.1 free download:
  http://www.eda.org/vfv/docs/PSL-v1.1.pdf

- SystemVerilog Assertions Introduction (Project Veripage):
  http://www.project-veripage.com/sva_1.php

- OVL Users Site (documents, downloads, etc.):
  http://www.eda.org/ovl/

- Documents ALDEC website:
  http://www.aldec.com/TechnicalDocuments/
  http://www.aldec.com/Events/
## ALDEC

<table>
<thead>
<tr>
<th></th>
<th>Website:</th>
<th>E-mail</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Website:</strong></td>
<td><a href="http://www.aldec.com">http://www.aldec.com</a></td>
<td><strong>E-mail</strong></td>
</tr>
<tr>
<td><strong>Telephone</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USA</td>
<td>(702) 990-4400</td>
<td><a href="mailto:sales@aldec.com">sales@aldec.com</a></td>
</tr>
<tr>
<td>Canada:</td>
<td>+1-613-867-8600</td>
<td><a href="mailto:sales@aldec.com">sales@aldec.com</a></td>
</tr>
<tr>
<td>Europe:</td>
<td>+33-6-80-32-60-56</td>
<td><a href="mailto:sales-eu@aldec.com">sales-eu@aldec.com</a></td>
</tr>
<tr>
<td>Japan:</td>
<td>+81-3-5312-1791</td>
<td><a href="mailto:info@aldec.co.jp">info@aldec.co.jp</a></td>
</tr>
<tr>
<td>China:</td>
<td>+86-21-6875-20-30</td>
<td><a href="mailto:info@aldec.com.cn">info@aldec.com.cn</a></td>
</tr>
</tbody>
</table>

- To attend additional ALDEC events, go to [www.aldec.com/events/](http://www.aldec.com/events/)