A Practical Guide to Deploying Assertions in RTL

Texas Instruments

Charu Aggarwal, Genadi Osowiecki, and Shobha Subramanian

Session # 2.8

Presented at

cadence designer network

cdn LIVE™

Silicon Valley 2007

Abstract:

With the recently growing interest in assertion-based formal analysis tools, we are reminded again that RTL designers have a large role to play in creating assertions for their designs. After a brief review of the typical taxonomy of assertions, this session walks RTL designers through the reasoning behind why it is in their own best interests to participate in assertion specification and highlights situations in which the RTL designer is indeed the best person on the team to create some assertions for use in both formal analysis and simulation. Finally, this session provides several examples of "real world" design fragments, and the type of assertions that an RTL designer might consider deploying.

This session is based on the authors' research and experience deploying assertions within RTL and Verification design teams for multiple projects at Texas Instruments. The emphasis is on practical approaches rather than theoretical arguments about the value of assertions in idealized project environments.

The intended audience of this session includes RTL designers who are in the process of deploying assertions (perhaps under duress due to external pressure) as well as verification engineers, who will benefit from understanding where best to draw the line between their responsibilities and those of the RTL designers.

Outline:

Introduction

The purpose of this paper is to provide an RTL Designer a very basic introduction to the exciting world of Properties and Assertions. After some brief introduction to the taxonomy of properties, we discuss the benefits of deploying properties early in the design cycle. Our focus is to explain the benefit of properties in terms of reducing the work effort of the designer as well as the DV team, and demonstrate how proper and careful deployment of properties will accelerate the design and verification cycle. Lastly, we provide some examples taken from “real world” designs to help “whet the appetite” and provide a starting point.

Brief Overview and Benefit of Properties

Properties are facts about the design and its environment that specify the behavior. There are 3 distinct kinds of properties based on their usage.

Assertions

“Assertions” are properties describing the behavior of the design. Formal tools are used to prove these properties. Conceptually they are equivalent to “checkers”.

Assumptions

“Assumptions” are properties describing the behavior of the environment around the block. It specifies the assumptions that have been made about the environment and its interaction with the block. Formal tools use this to constrain the verification state space for the given design.

Covers

Covers are behavior properties of the design or the environment, that are used to validate that such behavior can indeed occur. Covers have several uses including: testing the liveness of the design during initial development stage, “mark” important features in a design, or check for over-constrained environment.

Features

Features in a given specification can be categorized into one of the following behaviors. [2]

- Condition: Requirement on one or more signals regarding their value and/or relationship.
  For example, chip select is one hot.

- Stability: Requirement that a signal must stay stable over a given period of time.
  For example, request must stay asserted till grant is asserted.

- Response: Requirement about the response to a trigger condition.
  For example, grant must be asserted within 10 cycles of request being asserted.

- Stimulus: Requirement about the response occurring only after a trigger condition is seen.
  For example, Grant should only assert for a pending request. In other words, if grant is asserted, then request should have been asserted in the previous cycle.

- Sequence: Requirement regarding sequencing of conditions.
  For example, Request should be followed by grant being asserted. Grant must then be de-asserted a cycle later.

Benefit of Using Properties

Using properties during the development of a design has several key benefits.

Reusability

Properties can be reused in several different ways

- Some assertions for one block can be reused as assumptions for the block downstream and vice versa.

- Properties could also be reused across projects using the same IP.

- Properties can be reused at the higher level. For example block level properties can be reused in the chip level simulation environment.

- Properties can be then used to build a whole library. They can then be just plugged in and reused. They can be used for compliance checks or for complete formal verification of the feature. A good example would be OCPIP assertion library that is available for verification of the OCP Protocol. There are complete set of properties also available for various design components like the FIFO and arbiter that designers can use.

Portability

Properties can be ported across several different platforms

- Properties can be used in formal analysis and Dynamic simulation.
- Properties can be ported into gate level simulations. [3]
- Properties can also be ported into emulation environment.

Low Overhead

Development of properties does not have a big overhead. There is no need for a lot of infrastructure, especially if formal analysis is used. The designer just needs to obtain or write the properties. The properties may contain the minimum set of constraints and the assertions for the features the designer wishes to verify.

Other Benefits

Reduction in debug time

Properties are localized and hence help reduce debug time. They fire in case of a violation, and help with debug. This can indicate an actual bug in the design or an incorrect usage of the block. This is especially useful as we move up the hierarchy.

For example, Say we have an assumption that request must stay asserted till grant is asserted. However, at chip level, the request is being de-asserted before grant is asserted. It will take a lot of time and effort to find this bug. On the other hand, when block level properties are used as assertions at the SOC level, the property will cause a simulation error, or a formal analysis violation and identify the bug.

Performance analysis

Performance analysis can be performed by using simple response behavior property. This is useful when using formal analysis tools.

For example, say we have a requirement that grant must eventually be asserted.
A tighter restriction can be added to see if the block meets the requirement that grant is always issued within 5 cycles after request.

Create waveforms without test bench

Designers can write some simple cover properties for simple transactions. Formal tools will generate a waveform for each cover property that can be met. This is a very easy way to create waveforms without having to create a test bench and driving stimulus.

Why RTL Designers Need to Participate in Developing Assertions?

Why formal?

Formal requires a very limited setup cost relative to simulation testbenches. Since Formal Verification can typically start earlier than simulation, resolution of bugs at this stage by the RTL designer provides a more stable model for the start of simulation by the DV Engineer. Assertions can help verify corner case bugs that are extremely hard to hit in simulation [1].

Note that even without the deployment of a formal tool, properties in the RTL can be used in simulation to expose simulation coverage holes.

Thus, using formal at unit level by RTL designers significantly reduces system simulation time and expedites time to closure.

Impeding Factors to using Properties by RTL designers

The major impeding factor in the use of properties by RTL designers is that they are reluctant to use them. This is because of the combination of three factors. First, many designers are comfortable with simulation and favor the “easy to set-up and use” simulation methodology to do their initial debug. Secondly, RTL designers don’t feel the need to be thorough in their RTL check and only perform basic sanity checks, since, most believe that verification is the job of the DV Engineer. Lastly, the demonstration of value in using formal techniques requires an upfront investment which designers seldom do due to time pressure. In this presentation, we aim to contribute to the consideration of Formal Verification deployment by RTL designers by highlighting the importance and benefits achievable for the entire design team.

Why RTL Designers Need to Participate in Developing Assertions?

Properties can be classified broadly into two major types at block level – those within the block and those which interface with outside world.

The properties that are localized to the block are called white box properties. These deal with the intricacies of the block like FSM interactions, protocol violations, corner case behavior specifications and properties on never occurring conditions. White box properties are typically a function of inputs, outputs, internal signals and design specific behavior. The RTL designer, being the closest to the block level design, is the best person to address the white box properties.

White box properties have a relatively significant impact on partitioning the design. For instance, instead of interspersing the control logic with the data code, the designer may decide to have a common control block to ease writing and debugging of properties. Partitioning to create smaller sub-blocks make the design easier to understand, analyze and interface with other sub-blocks, decreasing the chances for coding errors. Also, adding parameterization in RTL to queues, memories, counters and transaction variables like address/data bus width, burst-length, etc. helps in reduction of formal complexity and block reusability. All this emphasizes the involvement needed by the design engineer. This way, the block bring up and sanity check of the block is done way ahead of time by the RTL designer herself without getting the DV Engineer involved. In a nutshell, since white box properties focus on less complex problems, deal with smaller cones of influence, making debug easier and ROI (Return on Investment) higher, these should be done first and by the RTL designer.

The properties that interface with the outside world are called black box properties. Both the RTL designers and DV Engineers need to be involved in writing these properties.

How Assertions Reduce the Amount of Work that RTL Designers Need to Do?

As mentioned previously it is imperative for the RTL designer to participate in the assertion planning and implementation phase of the project, having helped develop many useful assertions such as white-box and control properties.

Most RTL Designers develop their own private HDL-based test bench prior to "handing off" the design to the DV team for verification. It is only natural, after all, to try and do the most basic verification of one's own work before "burdening" others with it.

Most HDL-based test benches perform tasks such as:

- Check that the design comes out of reset correctly
- Check that major state machines can transition through most of the "important" states
- Perform basic bus and memory accesses
- Fill and empty FIFOs
- Handle simple and short transactions.

These test benches are usually not self-checking, are of little use to other persons or teams except for the original developer, often "break down" with changes to the design, and are typically developed during "spare time" such as evenings, weekends, and team meetings.

Note, however, the similarities between the scopes of these HDL-based test benches and the assertion scopes described in the previous section. Given the fact that the effort to develop "lower level" properties such as White Box, and Control properties, it is only natural to use these – in conjunction with a FA tool to verify many, if not all, the properties that are traditionally targeted by a private HDL test-bench.

To summarize:

- RTL is of better quality at handoff time to DV engineer
- Sub-blocks have been verified to a good extent by formal analysis, so the amount of back-and-forth debug time between the DV Engineer and the RTL designer significantly reduced
- May totally eliminate designer's need for a preliminary testbench
- Net result -> Overall reduction of time that the RTL designer spends on the design.

How RTL Assertions Help the DV Effort and Reduce Overall Time to Market?

As stated previously, the verification of properties using formal tools does not necessitate the creation of a test bench. The authors observed time and again RTL designers spending considerable effort creating their own "sand-box" test benches where they can do some basic verification of their blocks prior to handing off the block to the Verification Team. If the creation of such "sand-box" test benches is eliminated then the RTL can be handed to the Verification team sooner.

While it is difficult to quantify "quality" – and at present time the authors have not seen any statistics to either support or contradict the following - the "common notion" is that with properties the quality of the RTL hand-off is better – therefore – reducing the number of design iterations and accelerating the design cycle.

Using directed-random techniques it is possible to waste considerable time trying to hit a hard-to-reach corner case. If the corner case, however, can be formulated as an assertion – then a good Formal Analysis tool will allow the team to quickly assess if the corner case is even reachable in the design, and if so, have on-hand a sequence of events to guide the development of a test-case to hit the corner case.

Assumptions (constraints) on block inputs, once developed, can be fully validated at the next level of integration (i.e. SOC level) using Formal Analysis tools. On the other hand, if the next level of integration is verified using dynamic simulation, then block input assumptions play a vital role at quickly alerting the Verification Team to violations of input constraints, helping the Verification Team to quickly focus on trouble areas, and greatly accelerating the debug cycle.

Interaction between the RTL Designer and the DV Engineer

- The current interaction between the RTL Designer and DV Engineer roughly follows the following steps:
  - RTL designers develops a "sanity check" testbench (or wait on the DV engineers for the TB)

    - RTL designers think of the cases they wish to test.
    - Checking is performed by viewing the waveforms.
    - TB usually not capable of handling corner cases.
    - TB generally <u>not re-used</u> by DV team.
- In Contrast the ABV methodology is as follows:
    - RTL designers develop “constraints” (assumptions) on the input
    - RTL designers develop the “check” representing expected behavior of the feature being tested (assertions) – most importantly, are White Box assertions for expected behavior of circuits which are inside the design, and, therefore, are normally outside the work scope of the DV Engineer.
    - RTL designer develops coverage properties of scenarios which are deemed important.
    - Run Formal Analysis tool.
    - The tool does exhaustive checking.
    - Waveforms for any case can be generated without having to create a check.
- Writing properties calls for closer interaction between the RTL designer and the DV Engineer.
- RTL Designer needs to maintain a properties document to discuss the white box properties with the DV Engineer. This will help the DV Engineer decide on how to use the white box properties in simulation (if at all) and how to leverage the existing properties to develop the black box functional and protocol properties.
- Any feature or functionality that the RTL designer expresses concerns about – for example – if the feature was not tested fully by the RTL designer or the RTL designer feels the need for additional verification - should be documented in the properties document or RTL for the DV engineer to “pick up”. The DV Engineer then focuses on the black box interface and protocol properties at the block level in addition to the areas pointed to by the RTL designer.
- As previously stated, this approach allows for early start of verification by the RTL and maximum reuse by the DV Engineer of the efforts put forward by the RTL designer. Furthermore, the sanity check done by the RTL designer minimizes the back and forth debug time that is typically done saving both the designer’s and DV Engineer’s time.

- **ABV Reusability :**

  - Designers can “capture” important features into cover items. These items can then be reused by DV team for coverage.
  - DV teams can reuse properties written by the RTL designer during simulation to aid in debug.
  - DV teams may reuse properties written by the RTL designer during Formal Analysis as well.
  - The following table summarizes the leveraging of properties from lower level to higher level of integration. Note that the use model varies slightly depending on the choice of Dynamic or Formal verification at the higher level of integration.

	DV with Simulation	**DV with Formal**
Assumptions **At top-most block level**	Use as assertions	Reuse as assumption with caution - use an “ifdef” to conditionally “hide” from DV team
Assumptions **At sub-block level**	Use as assertions	Use as assertions – can potentially identify over-constrained assumptions
Assertions	Use as assertions and Cover Items	Can be used as assertions if the input constraints used by the RTL team and the DV team are different
Covers	Use as Coverage items	Use as cover (to identify over-constrained situations)

Steps for Deploying Assertions in RTL

- Design Partition - Most design methodologies strongly encourage the partitioning of control and data path sections into separate blocks. Formal Analysis is no different.

- Parameterization - Following good design practices, it is strongly recommended that the design be parameterized. Some common examples are: depths of queues, width of busses, number of request inputs to an arbiter, etc.

- Lint and Automatic Formal Analysis Checks (Deadcode, FSM deadlock etc)

- Add simple cover statements to check for design liveness

- Start with Control Blocks first.
    - Identify Blocks where existing Verification IP's can be deployed (i.e. standard protocols)
    - Identify features that you wish to check
    - Add properties for features to be checked
    - Identify important/corner cases (You don't have to check these, just identify)
    - Add cover statements for the corner cases. (If not covered by the DV, they will show up as coverage holes in assertion coverage)

- Move to datapath blocks and repeat above steps.
    - Note that one can reuse properties from control block in the datapath as assumptions.

Real World Examples of Using Assertions

State Machine Properties:

Following are two simple examples of assertions to verify the correct behavior of an FSM. Note that while these examples might appear on the surface as trivial, these examples can be easily extended to complicated system states, where the "FSM State"could be – in fact – a combination of many disjoined signals.

In Example 1 S is an FSM state. The property states that after S traverses the sequence of S0 and START, following by S1, then the next state should be S2.

```
 psl property assert_s0_error : always (
       {(S == S0) &&  START; (S == S1)} |=>
                (S == S2) abort !RST_N
 ) @ (posedge CLK);
```

Example 1

In Example 2, the property states that if the FSM has transitioned from S4, with a signal called M3 being asserted, and then into S7, then the next state should be S5.

```
psl property assert_state4_m3_error: always (
   {(S == S4) &&  M3; (S == S7)} |=>
      (S == S5) abort !RST_N
) @ (posedge des_clk);
```

Example 2

State Machine Covers:

Here’s an example of a cover for a somewhat typical scenario. The cover involves an FSM, S, and a signal called DONE. The area of interest is to see if DONE can transition from zero to one when the FSM is in either S4 or S13

```
psl cover_done_1 : cover {
   DONE == 1'b0;
   DONE == 1'b1 && ((S == S4) || (S == S13))
};
```

Example 3

Here is another example, similar to the above. The scenario of interest is the signal NEXT transitioning from zero to one the FSM is in either S3 or S12.

```
psl cover_nxt_1 : cover {
   NXT == 1'b0;
   NXT == 1'b1 && ((S == S3) || (S == S12))
};
```

Example 4

The below assertion states that if state = STATE0 and START and D_MODE are asserted then the FSM proceeds to STATE1, STATE2, STATE3, STATE4 and STATE5 states in subsequent clock cycles.

```
                       psl property assert_state0_to_5_error = always (
                             {(state == STATE0) && START && D_MODE} |=>
                             {(state == STATE1);
                              (state == STATE2);
                              (state == STATE3);
                              (state == STATE4);
                              (state == STATE5)} abort !RST_N
                         ) @ (posedge CLK);
```

Example 5

The below assertion states that if state = STATE0 and START is asserted and D_MODE is deasserted then the FSM proceeds to STATE1, STATE2, STATE3, STATE4 and STATE6, STATE7, STATE8, STATE9, STATE10, STATE11, STATE12, STATE13, STATE14 states in subsequent clock cycles.

```
psl property assert_state0_to_14_error = always (
    {(state == STATE0) && START && !D_MODE} |=>
                       {(state == STATE1);
                        (state == STATE2);
                        (state == STATE3);
                        (state == STATE4);
                        (state == STATE6);
                        (state == STATE7);
                        (state == STATE8);
                        (state == STATE9);
                        (state == STATE10);
                        (state == STATE11);
                        (state == STATE12);
                        (state == STATE13);
                        (state == STATE14)} abort !RST_N
                         ) @ (posedge CLK);
```

Example 6

State Machine Never:

The below never statement ensures that DMODE assertion and state == STATE6 can never happen at the same time.

```
psl property assert_never_state6_on_des_error : assert never (
    (DMODE && (state == STATE6)) @ (posedge CLK);
```

Example 7

The below never statement ensures that DMODE deassertion and state == STATE5 can never happen at the same time.

```
psl property assert_never_state5_on_3des_error : assert never (
     (!DMODE && (state == STATE5)) @ (posedge CLK);
```

Example 8

Data Flow and Correct Transformation:

In Example 9, a data transformation block is supposed to process input data in 5 clock cycles. In the first cycle START is asserted, and during the processing cycle MODE is supposed to be asserted and ENC de-asserted.

The transformation logic is driven by an FSM, whose state S is supposed to count up from 0 to 4.

Example 9 is much more suited to Formal Analysis, and is not designed to exhaustively verify the logic, only to prove, that given a specific input (DI of 0x0123 and SELECT of 0xDFF1), the logic will produce the appropriate output (0x85E8) – therefore proving some measure of "liveness".

```
psl property assert_top_data_1toolzeros_error = always ({
  (START && (DI == 16'h0123) && (SELECT == 16'hDFF1) && (S == 4'h0) && MODE && !ENC);
           ((DI == 16'h0123) && (SELECT == 16'hDFF1) && (S == 4'h1) && MODE && !ENC);
           ((DI == 16'h0123) && (SELECT == 16'hDFF1) && (S == 4'h2) && MODE && !ENC);
           ((DI == 16'h0123) && (SELECT == 16'hDFF1) && (S == 4'h3) && MODE && !ENC);
           ((DI == 16'h0123) && (SELECT == 16'hDFF1) && (S == 4'h4) && MODE && !ENC)
    } |=> (DO == 16'h85E8)
    abort !RST_N) @ (posedge CLK);
```

Example 9

Simple Input Assertions

Following are some examples of assertions one would typically find on module inputs and outputs.

Example 10 specifies that the signal DONE is a pulse of one clock cycle in duration:

```
psl assert_done_pulse : assert always ({done} |->
                         next {!done} abort !RST_N) @ (posedge CLK);
```

Example 10

or

```
psl assert_done_pulse : assert always ({done} |=>
                        {!done} abort !RST_N) @ (posedge CLK);
```

Example 11

Example 12 is of a somewhat complicated protocol. Here, the protocol stipulates that if X and Y are zero and one respectively, then for the next 10 clock cycles BUSY will be asserted, DONE will be de-asserted, unless we get a new START. On the 11th clock cycle DONE will be asserted, and BUSY de-asserted.

```
psl assert_done_busy : assert always ({START && (X == 0) && (Y == 1)} |=> {
                       BUSY && !START && !DONE [*10];
                      !BUSY && !START &&  DONE} abort !RST_N) @ (posedge CLK);
```

Example 12

Example 13 stipulates that the design should never go BUSY without a START in the previous cycle, and START must go low before BUSY goes high. Note that both properties are necessary to fully specify this property.

```
psl assert_valid_busy : assert always ({START; !START } |->
                         {rose(BUSY)}) abort !RST_N) @ (posedge CLK);
psl assert_valid_start : assert always ({!BUSY; BUSY} |->
                         {fell(START)}) abort !RST_N) @ (posedge CLK);
```

Example 13

The rising edge of DONE should always be accompanied by falling edge of BUSY, and vice versa

```
psl assert_valid_done: assert always ({!DONE; rose(DONE)} |->
                          {fell(BUSY )}) abort !RST_N) @ (posedge CLK);
psl assert_valid_done: assert always ({BUSY; (fell(BUSY)} |->
                          {rose(DONE )}) abort !RST_N) @ (posedge CLK);
```

Example 14

If FREEZE is not high, FREEZE_ACK will not be high

```
psl freezeack_not_high: assume always {FREEZE ='0'} |-> {FREEZE_Ack='0'};
```

Example 15

If INITZ is not low, MOD_READY will not be high

```
psl input_initz_not_high: assume always {INITZ='1'} |-> {MOD_READY='0'};
```

Example 16

FREEZE will remain active until it is either acknowledged with a FREEZE_ACK or WAKEUP.

```
psl output_freeze_always_high_wo_ack: assert always (
   {PO_FTFREEZE AND NOT(PI_FTFREEZE_Ack) AND NOT(PI_Wakeup_event)}
       |=> {PO_FTFREEZE}) abort (RESET_PULSE);
```

Example 17

FREEZE cannot be high unless SLEEP is high the previous cycle.

```
psl output_freeze_never_high_wo_sleep: assert never {NOT SLEEP; FREEZE };
```

Example 18

nCPUReset cannot be issued without a hardware or software reset

```
psl assert always (NOT nCPUReset -> (NOT SYSRESET OR SW_RST))
```

Example 19

Examples for Performance/FIFO/Arbiter :

All hi priority requests should be served within 10 cycles.

```
       psl assert_hi_grant_in_10 : assert never ({(HI_REQUEST && !HI_GRANT)[*10]});
```

Example 20

All low priority requests should be served within 20 cycles.

```
       psl assert_lo_grant_in_20 : assert never ({(LO_REQUEST && !LO_GRANT)[*20]});
```

Example 21

Only 1 grant is asserted at any given time.

```
       psl assert_single_grant : assert never (HI_GRANT && LO_GRANT);
```

Example 22

Lo and Hi priority FIFO can never fill up at the same time.

```
       psl assert_hi_lo_fifo_full : assert never (HI_FIFO_FULL && LO_FIFO_FULL);
```

Example 23

Control Logic Examples:
Arbitrator examples:

AXI Protocol Stability Assertions:

Pin Name	Width	Source	Description
AWID	4	Master	Identification tag for the write address group of signals
AWREADY	1	Slave	Indicates that the slave is ready to accept an address and associated control signals
AWVALID	1	Master	Indicates that the valid write address and control information are available
ARESETN	1	Reset Source	Global reset signal – active LOW

AWID must be stable until AWREADY goes high:

```
// psl master_stable_awid : assert always (( (AWVALID && !AWREADY) ->
//                                              next (AWID == prev(AWID))
//                                           ) abort (! ARESETN) )
```

Example 24

OCP Protocol Covers:

Pin Name	**Width**	**Source**	**Description**
Mcmd	3	Master	Transfer Command
SResp	2	Slave	Transfer Response
SCmdAccept	1	Slave	Slave accepts transfer
No_rsp_pending	1	Auxiliary Code Register	Value 1 indicates that there are no pending response requests, 0 otherwise

Cover the sequence where end of request phase to begin of response phase latency is MAX_LATENCY clocks

```
// psl cover_reqToRespLatencyMax: cover { No_rsp_pending         &&
//                                         (Mcmd_i !=  MCMD_IDLE ) &&
//                                         (SResp_i == SRESP_NULL) &&
//                                          SCmd_Accept_i           ;
//                                         (Mcmd_i == MCMD_IDLE)   &&
//                                         (SResp_i == SRESP_NULL) [*MAX_LATENCY-1];
//                                         (Mcmd_i == MCMD_IDLE)   &&
//                                         (SResp_i != SRESP_NULL)
//                                        };
```

Example 25

Dos and Don'ts

Listed below are a few do's and don'ts that we discovered as a part of our learning experience implementing formal properties in a few of our projects:

Do Initialize

- **What** : Initialize all uninitialized FF's

- **Why** : This reduces the state space for the tool to explore. This is important for DV Engineers when using a reference model (ie System Verilog), as these are typically not initialized.

- **Tip** : In IFV just load the design and do "init –show –uninit" in order to view un-initialized registers.

<u>Do</u> Label

- **What** : Label all properties and follow a consistent naming convention. Include block name and the property type in the name.

- **Why** : Allows easy selection/deselection of properties when tcl files used

- Example

  – For Assumptions, prefix all properties with "xyz_rtl_assume_"

    -
      ```
      // psl arb_rtl_assume_start_is_pulse : assume always (
           {start} |=> {!start});
      ```

  – For Assertions, prefix all properties with "xyz_rtl_assert_"

    -
      ```
      // psl arb_rtl_assert_grant_is_pulse : assert always ({grant} |=>
      {!grant});
      ```

  – For Covers, prefix all properties with "xyz_rtl_cover_"

    -
      ```
      // psl arb_rtl_cover_grant_is_set : cover {grant; !grant};
      ```

- Usage in tcl file
  - o assertion –add   arb_rtl_assert_*
  - o assertion –delete arb_rtl_asssume_*

<u>Do</u> Encapsulate

- **What** : Encapsulate all properties in ifdef's

- **Why** : Allows easy inclusion/exclusion of properties in compile scripts (especially in dynamic simulation env). Don't know how the code will be used and if the tools downstream support the constructs used. Hence should always provide a method for excluding the properties from the RTL.

- Example

- For Assumptions, encapsulate in “#ifdef XYZ_RTL_ASSUME_ENABLE”
- For Assertions, encapsulate in “#ifdef XYZ_RTL_ASSERT_ENABLE”
- For Covers, encapsulate in “#ifdef XYZ_RTL_COVER_ENABLE”

<u>Do</u> Run RTL checks

- **What** : Run the design through Formal tools even if you <u>don’t</u> have properties.
- **Why** : Formal tools have some inbuilt property checkers that can help find some design issues especially the ones that can cause the RTL to behave differently than gates. They require no properties, just the design and the tcl command file.
- **Some of the checks provided by Cadence Incisive Formal Verifier are**
    - Deadcode Check
    - FSM Check (state transition, deadlock)
    - X Check
    - Bus Check (floating bus, multiple drivers, contention)
    - Multiple drivers Check
    - Parallel Case/Full Case Check
    - Array Index out of bounds Check

<u>Do</u> Divide and Rule

- **What** : Break bus properties into separate properties for each bit
- **Why** : To reduce design complexity and avoid getting an “Explored” in IFV. (Note this works for outputs only).
- Example
    - The assertion below is comparing the output of the DUV (64-bit bus dout) to the Expected Output of a Reference Model (64-bit bus exp_dout).

```
blk_a_dout  : assert always (dout == exp_dout)
```

can be rewritten as:

```
blk_a_dout_0 : assert always (dout[0] == exp_dout[0])
blk_a_dout_1 : assert always (dout[1] == exp_dout[1])
...
blk_a_dout_63 : assert always (dout[63] == exp_dout[63])
```

Do Review each and every assumption in your environment

- **What** : **Do** review and check **each and every assumption** you have in your Formal Analysis environment. Add assumptions one at a time . Avoid using wildcards. This is not a problem in dynamic simulation as all constraints are treated as assertions and do not affect the input space.

- **Why** : In formal Analysis, they "add" constraints in the environment and affect the stimulus. They may end up over constraining the environment. This is especially true if you are using a sub-block from a 3rd party. The module may have embedded assumptions. (bad coding practice, should always encapsulate properties inside ifdefs). This is especially dangerous when doing DV as the DV engineer may not know where assumptions exist deep inside the design.

- **Solution** :
  - Always encapsulate all properties inside separate ifdefs
  - Follow a consistent naming convention.
  - Do not include the assumptions by default
  - The first thing you do after you load the design is "delete all constraints"
    - In IFV, you can add the following command to the TCL file
      constraint -delete -all
  - Convert the sub-block level "assumptions" into assertions. This will aid in identifying differences in assumptions made at various levels.
    - In IFV, you can add the following command to the TCL file
      assertion –add <top_blk>.<subblk>.<assumption_name>
  - Then add the top level assumptions you wish to add explicitly, one at a time. This way you have total control on what gets added. Using wildcards may result in addition of exrta properties.
    - In IFV, you can add the following command to the TCL file
      assertion –add <top_blk>.<assumption_name>
  - Add cover properties for input conditions you want verified.

Don’t Re-write RTL

- **What** : Don’t write properties that “duplicate” the RTL implementation.

- **Why** : You are not testing anything.

- **Solution** : Use different logic to test the same feature.

- Example :

  If the specification mandates that
  - In state 4, if ‘mode’ is asserted, go to state 5 else go to state 6
  - To get in state 4, ‘req’ needs to be high in states 0, 1, 2 and 3.

  there is not much achieved by coding assertions as shown below since this is pretty much what the rtl will look like.

```
// psl assert_state_5 :  assert always  ({(state == state4) && mode } |=>
//                                        (state == state5)
//                                       abort !RST_N)  @  (posedge CLK);


// psl assert_state_6 :  assert always  ({(state == state4) && !mode } |=>
//                                        (state == state6)
//                                       abort !RST_N)  @  (posedge CLK);
```

Instead we could write a state sequence as shown below:

```
//psl assert_state_5_mod : assert always ({(state == state0) && req && mode} |=>
//                                        {(state == state1) && req;
//                                         (state == state2) && req;
//                                         (state == state3) && req;
//                                          state == state4;
//                                          state == state5
//                                        } abort !RST_N ) @ (posedge CLK);

//psl assert_state_6_mod : assert always ({(state == state0) && req && !mode} |=>
//                                        {(state == state1) && req;
//                                         (state == state2) && req;
//                                         (state == state3) && req;
//                                          state == state4;
//                                          state == state6
//                                        } abort !RST_N ) @ (posedge CLK);
```

And we could write covers as below to check that we did cover an instance of the required sequence and the never statement is yet another check that is a different from the way the rtl has been coded:

```
// psl cover_state_5_mod : cover{  (state == state0) && req && mode;
//                                 (state == state1) && req;
//                                 (state == state2) && req;
//                                 (state == state3) && req;
//                                 (state == state4);
//                                 (state == state5)};
// psl never_state_5_mod : assert never {(state == state4) && mode ;
//                                         (state== state6)};


// psl cover_state_6_mod : cover {(state == state0) && req && !mode;
//                                 (state == state1) && req;
//                                 (state == state2) && req;
//                                 (state == state3) && req;
//                                 (state == state4);
//                                 (state == state6)};


// psl never_state_6_mod : assert never {(state == state4) && !mode ;
//                                        (state== state5)};
```

This way, we are testing the state machine behavior comprehensively using formal properties.

Conclusions

We have presented an effective guide for an RTL designer to implement properties in any design. We have underlined the importance of getting the RTL designer involved and highlighted how this can potentially save time for the entire design team. Even though we have not been able to quantify the time savings, we firmly believe, based on our experiences on a couple designs, getting the RTL designers involved in writing properties minimizes overall DV time and ensures quality design handoff to the DV Engineer. The examples we have presented are taken from real projects and should motivate RTL designers to start implementing properties.

Acknowledgments

The authors would like to express their most sincere appreciation to the following folks:

Our teachers and mentors at Cadence – specifically Amir Attarha, Robert Juliano, Jose Barandiaran, and Duy Nguyen.

Our colleagues at Texas Instruments – specifically the DV and RTL teams on the Mustang , Leo, and Panther projects.

Our supervisors, Jim Coates and Levon Petrosian who “looked the other way” when we took time from our busy schedules to work on this paper.

Lastly, thanks to Jeroen Vliegen who gave Gery his first “taste” of Formal Analysis.

References

[1] Yaron Wolfsthal, Rebecca M. Gott, Formal Verification – Is it Real Enough?, DAC 2005.

[2] Incisive Formal Verifier Advanced Training Notes
[3] http://www.verilab.com/files/svagate_paper_dvcon2006.pdf

About the Authors:

Charu Aggarwal is a Functional Verification engineer at Texas Instruments Inc with the Communication Infrastructure and Voice Business Unit. She has over 6 years of experience in Coverage/Constraint based verification methodology using Specman, C/C++ and System Verilog. She has also developed System level Transaction level Models for IP using SystemC. More recently she has been involved with Assertion based verification for Formal Analysis and Simulation.

Genadi Osowiecki is a Senior DV Engineer with Texas Instruments' Wireless Terminal Business Unit in Dallas. Gery has been responsible for design and deployment of Coverage Based Verification Technologies and Methodologies within T.I. and is one of the pioneers in this field. Recently Gery has been involved as "champion" in the deployment of Assertion Based Verification at T.I. Gery has well over twenty-five years of experience in designing and verifying computer systems, microprocessors, DSPs, and peripherals.

Shobha Subramanian is a Design Engineer with Texas Instruments' Communication Infrastructure and Voice Group at Dallas. She has over ten years of experience in back-end design, front-end design, DV and system level modeling of co-processors and controllers. Recent experience includes System Verilog, System C based transaction level modeling (TLM) of IP and assertion based RTL implementations.