Architecting “Checker IP” for AMBA protocols

Srinivasan Venkataramanan[1] Ajeetha Kumari[2]

[1]Organization: VerifWorks Pvt. Ltd.
2nd Floor, VTD Square, 183/3, Sarjapur Road, Dommasandra, Above HDFC Bank Dommasandra branch, Anekal Taluk, Bangalore, India 562125
Job Title: Verification Technologist, Email: srini@cvcblr.com Mobile no: +91-9620209225

[2]Organization: VerifWorks Pvt. Ltd.
Job Title: Verification Consultant Email: akumari@cvcblr.com Mobile no: +91-9620209224

Abstract- Checkers are critical part of design verification process. With advent of constrained-random techniques, lot of stimulus generation capabilities are supported by languages, methodologies and supporting tools. Ensuring that a design behaves as per the specification is achieved using various forms of checkers. The term “checker” at times means different things to different people. With SystemVerilog 2009 standard making “checker” as a keyword in the language itself, we would like to use that for temporal checks using assertions. In this paper, we share our experience of architecting Checker IP (CIP) for typical protocols.

I. INTRODUCTION

The world of electronics system design has grown leaps and bounds in terms of complexity, at the same time reducing time to market for new products, derivatives etc. Such a trend has been enabled by IP’s (Intellectual Properties) – both at the design front and the verification front. When it comes to verification, these IP’s are commonly referred to as Verification Intellectual Properties (VIP’s) – a loosely used term today in the industry to imply simulation based, stimulus-driver-and-scoreboard components. In a typical SystemVerilog UVM-based context these are also often referred to as Universal Verification Components (UVCs).

There are usually two significant parts to a VIP - one portion consists of sequences, tests, sequence libraries, configurations etc. The other portion of a standard VIP performs checking (also known as scoreboards, checkers) and tracks coverage (functional & assertion).

Among these two portions, the first one mentioned above is very much necessary in simulation based verification. However, adjacent technologies such as Formal Verification (FV, Model Checking) only require the checkers/properties and a set of coverage points. Armed with what-to-check, FV tools tend to explore the entire state space, constrained only by the user-provided constraints models, and tries to prove or falsify the properties on the given DUT. In a way, a lot of heavy lifting of generating stimulus is handled by formal engines under the hood, though strictly speaking FV tools perform a mathematical exploration rather than a traditional stimulus-based approach.

To differentiate from traditional VIP, we use the term Checker IP - CIP. CIP strictly speaking, is a sub-set of standard VIP. In this paper, we present our experience of building a CIP – Checker IP that focusses on the “properties” of the design in the form of checkers, covers and constraints. Such CIPs are reusable in simulation as well as formal verification, provided enough care is taken during the architecting of these IPs.

II. CHECKER IP (CIP)

Checker IP is an IP that captures properties of a given protocol in the form of SystemVerilog asserts, covers and assumes [1]. Motivation behind creating a CIP is to focus on the expected outcome of the DUT in terms of expected behaviors, unexpected protocol violations, desired scenarios etc. In a CIP, we separate the "does-it-work", "are-we-done" queries from the means of doing it (either simulation or formal).

For standard protocol, such as ARM's AMBA family [2], it is common to define a set of compliance checks ([3]) from the specification itself. Given the number of systems being built around standard buses, it is imperative for the industry to be able to leverage on a standard CIP that checks for a list of well-defined compliance rules. However, a set of properties and asserts around them is not a reusable piece of CIP – they are simply collection of properties. There are several techniques and guidelines in making such a list of properties into an IP (CIP).

III. ARCHITECTING CHECKER IPS

The first and foremost requirement of any IP is that it is easily configurable to different configurations of the protocol/system. For instance, consider an AMBA AXI3 protocol with Masters and Slaves. A simple block diagram ([2]) of a master-slave over AXI bus is shown in Figure-1.

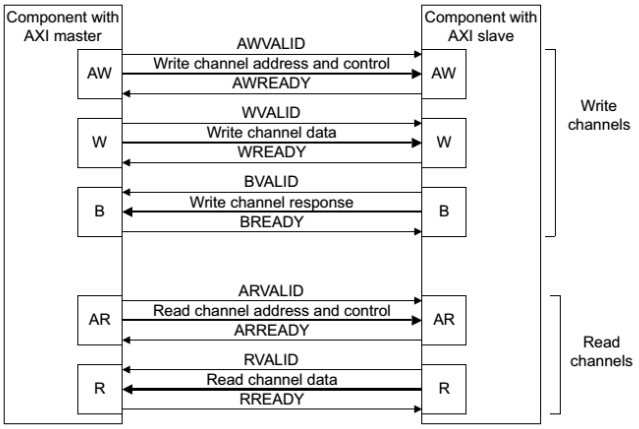

Figure-1 simple block diagram of a master-slave over AXI bus [2]

In the above picture, we only show the Master and Slave. In a complete AXI subs-system, there may be additional components such as interconnects, arbiters, decoders etc. Since the protocol is well documented, it is natural to extract a compliance check-list for AXI3 (ARM has a version released in [3]).
While capturing the checklist to an executable checker, mere translation of English specifications to a set of SystemVerilog Assertions could be a good start. However, such a simple laundry list of properties will be very difficult to use as a CIP in a simulation setup, for instance. This is so because there may be different ports in a Master vs. Slave interface. Then there are optional features that add additional ports to the interface (such as protection, locks, grants etc.) that may be present only in Master interface and not in Slave interface and vice versa.

In general, a given property can be an assert on one interface while the same is considered as an assumption in the other interface (e.g. Master should assert ***awvalid***, hence an *assert* or a check for Master interface; while the same ***awvalid*** is an input to the slave interface, thereby turning out to be an *assume* there). From a language perspective, SystemVerilog has:

- Assert (For checks)
- Assume (Constraints)
- Cover (For desired behaviors)

A typical CIP should use all the 3 above at every interface, for example a Master interface should contain properties classified into asserts, assumes and covers.

Consider AXI3 protocol, the specification (Section 4.4, Table 4-3) mentions legal values for ARBURST and AWBURST as shown in Figure-2 below:

Table 4-3 Burst type encoding

ARBURST[1:0] AWBURST[1:0]	Burst type	Description	Access
b00	FIXED	Fixed-address burst	FIFO-type
b01	INCR	Incrementing-address burst	Normal sequential memory
b10	WRAP	Incrementing-address burst that wraps to a lower address at the wrap boundary	Cache line
b11	Reserved	-	-

Figure-2 Extract from ARM AXI3 Specification

One could translate this specification to an executable property as shown in Figure-3 below:

```
574  //=========================================================================
575  // AXI Spec Section 4.4, Table 4-3 shows AWBURST == 2'b11 as a Reserved value
576  // Value of VW_AWBURST_RSRVD (2'b11) on AWBURST is not allowed when AWVALID is high
577  property p_vw_awburst;
578    disable iff(!ARESETn)
579    !($isunknown({AWVALID,AWBURST})) &
580    AWVALID
581    |-> (AWBURST != VW_AWBURST_RSRVD);
582  endproperty : p_vw_awburst
583
```

Figure-3 Sample CIP code for AWBURST requirement

A typical AXI system utilizes this signal in its Write channel as shown below:

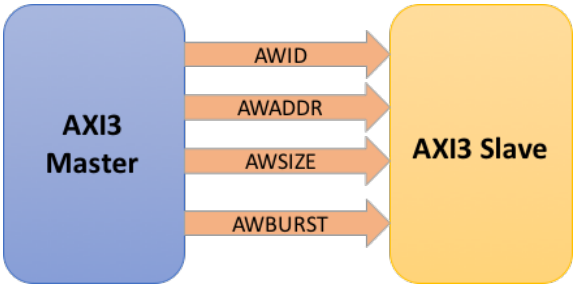

Figure-4 AXI Write channel (sub-set)

AWBURST is an output from the Master and an input to the slave. A property written as shown in Figure-3 above is applicable to both Master and Slave. However, from a Master perspective this property shall be a ***checker*** and from Slave perspective it is supposed to be an ***assumption*** or a ***constraint***. A CIP for AXI3 hence shall be architected in a way that the common properties are declared once.

Property as checker

A master_checker file shall use this property as an *assert* as shown in Figure-5 below:

```
1599
1600    a_p_vw_awburst: assert property (p_vw_awburst)
1601
1602    `ifdef VW_CIP_SIM
1603      else
1604        `uvm_error (vw_id,"p_vw_awburst: When AWVALID is high, a value of 2'b11 on AWBURST is not permitted. Spec: table 4-
    3 on page 4-5")
1605    `endif //VW_CIP_SIM
1606    ;
1607
```

Figure-5 AXI3 Master Checker

In the above assertion, action block is used to flag failures through popular UVM reporting infrastructure. Integration with UVM is highly recommended and useful when the testbench is in UVM. Such error reporting makes log files easier to read, parse/mine for failures. And with UVM one gets the added benefits of controllability of these messages and their associated actions.

Property as constraint

A property describing a behavior of a set of signals shall be configured as a constraint when the corresponding signals are inputs to the module under test. In case of AXI3, since AWBURST is an input to an AXI3 slave model, this property shall be used an assumption or a constraint. Sample code for AXI3 slave CIP is shown below in Figure-6.

```
1321
1322  m_p_vw_awaddr_boundary: assume property (p_vw_awaddr_boundary) ;
1323
1324  m_p_vw_awaddr_wrap_align: assume property (p_vw_awaddr_wrap_align) ;
1325
1326  m_p_vw_awburst: assume property (p_vw_awburst) ;
1327
1328  m_p_vw_awcache: assume property (p_vw_awcache) ;
1329
1330  m_p_vw_awlen_wrap: assume property (p_vw_awlen_wrap) ;
1331
```

Figure-6 AXI3 Slave Constraint

To provide a feel for the complexity, table below summarizes the approximate number of checkers and constraints for an AXI3 CIP.

Category	Number of entities (asserts/assumes)
AXI3 Master Checkers	102
AXI3 Master Constraints	20
AXI3 Slave Checkers	43
AXI3 Slave Constraints	67

III Guidelines for creating CIPs

SystemVerilog provides powerful and rich set of constructs to model temporal behaviors of designs in an unambiguous and concise manner [1]. However, not all constructs are formal verification compatible. While there is no standard subset of SVA for formal as per LRM, with our long experience with various EDA tools, we have arrived at a common minimal sub-set. We share some of our learnings on this domain in this paper.

Using ***checker*** construct

The ideal container for a CIP should be the relatively new ***checker..endchecker*** construct as described in our DVCon 2010 paper [4]. However, our experience shows that this construct is not well supported across tools and also there are certain restrictions on parameters etc. in the language itself ([1]). However, ***checker*** construct provides more flexibility in terms of its instantiation (procedural, inline, concurrent etc.). Hence for smaller assertion based entities such as the OVL-types [5], the authors believe ***checker*** is very useful. However, attempting to use them for a complex CIP will be more pain than worth.

Using ***module*** as container

One of the most common containers in Verilog and SystemVerilog is ***module***. It is quite common to use a ***module*** as container or encapsulation unit for assertions in SystemVerilog. To keep the assertions separate from the design, often engineers use ***bind*** construct to attach a checker module to a design module.

Quoting from a SystemVerilog Assertions book [6], section 4.7, consider a design module named "dut" and a checker entity named "vf", Figure-7 below shows how typically a ***bind*** construct is used to tie these 2 modules.

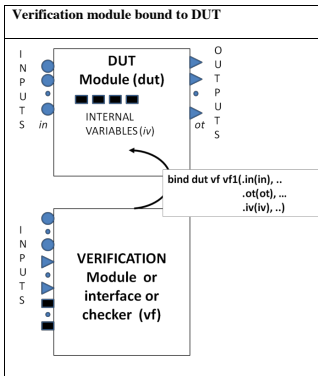

Figure-7 Using SystemVerilog ***bind***

SystemVerilog ***bind*** is elaborated by the simulator and internally it instantiated the bounded entity ***inside*** the module it is being bounded to. To demonstrate this concept, consider the equivalent of Figure-7 above, after a tool does the elaboration; it instantiates an instance of ***vf*** module inside the ***dut*** module. This is shown in Figure-8 below:

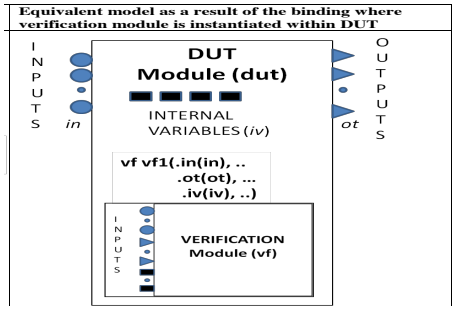

Figure-8 Internal mechanism of SystemVerilog ***bind***

*Using **interface*** as container

Given the prominent use of SystemVerilog and UVM-based testbenches in the industry, many teams use SystemVerilog ***interface*** feature as the primary means of communication across testbench and DUT. While an ***interface*** as in SystemVerilog LRM [1] can have many features as that of a ***module***, it cannot instantiate a ***module.*** This is explained below with the help of Figure-9 and Figure-10.

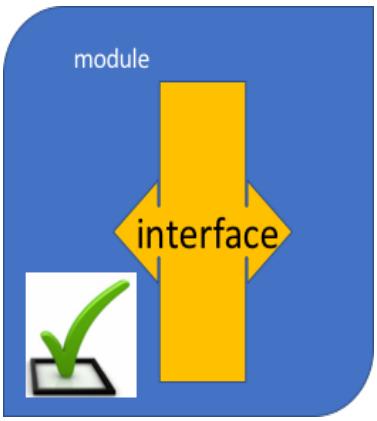

Figure-9 module can instantiate an interface

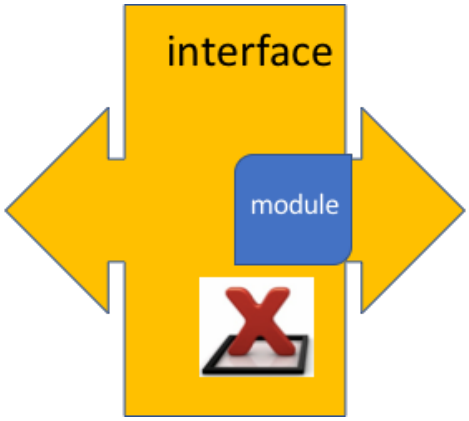

Figure-10 Interface cannot instantiate a module

Why this is important? Consider a typical AXI sub-system being verified with UVM based VIPs. The AXI pins are modelled using a SystemVerilog interface that gets instantiated as many times as necessary. The SystemVerilog interface serves as the primary means of communication between UVM class based system to a SystemVerilog module based design. Now, let's consider a scenario wherein, to strengthen the quality of verification a CIP is being bound to this AXI interface. If the CIP container were to be a ***module*** then we get a compile error from the tool indicating the problem highlighted in Figure-10 above.

A solution to this problem is to use ***interface*** as container for CIPs. As SystemVerilog allows an ***interface*** to instantiate another ***interface***, a good CIP modeled using ***interface*** can be bound to a DUT inside a ***module*** or a VIP based simulation setup using ***interface***. This is described in Figure-11 below:

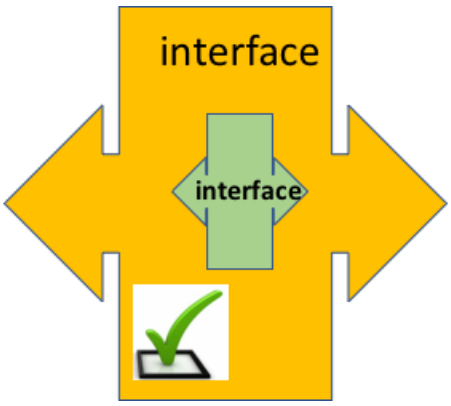

Figure-11 Interface can instantiate another interface

Guard simulation specific constructs with a text macro

SystemVerilog is a rich language with many features. Not all features are supported in all technologies that use this language. In the context of CIPs, it is desirable to keep the CIP usable in both simulation and formal verification flow. there are also some advanced non-determinism related features used in a pure formal IPs that are not simulation compatible. As a coding guideline, it is recommended to use appropriate text macros to mask parts of code from simulator and FV tools as needed.

One common example of this is the action block using UVM messaging system with assertions. Given that UVM is heavily class based and not synthesizable code, FV tools usually do not like that code. Hence a text macro (VW_CIP_SIM) is used to guard the action block code from FV tools and use it only in simulation as shown in Figure-12 below.

```
1599
1600    a_p_vw_awburst: assert property (p_vw_awburst)
1601
1602    `ifdef VW_CIP_SIM
1603       else
1604         `uvm_error (vw_id,"p_vw_awburst: When AWVALID is high, a value of 2'b11 on AWBURST is not permitted. Spec: table 4-
    3 on page 4-5")
1605    `endif //VW_CIP_SIM
1606    ;
1607
```

Figure-12 Using text macro in action blocks

Using appropriate delays with value change functions

SystemVerilog provides rich set of functions to model temporal behaviors. Some of the value access functions such as $past, $sampled etc. refer to values of signals from the previous clock cycles. Similarly, there are several value change functions such as $rose, $fell etc. that rely on the previous value of the signal being monitored with respect to a clock. While these are good, handy features in the language, care should be taken while coding them to avoid false failures in simulation and/or formal verification. Consider the following assertion (Taken from an online technical forum [7]):

```
2
3  a_p_poor_code : assert property( $rose(ack) |-> $past(req,DLY))
4   else
5      `uvm_error ("VW_CIP", "Expected delay of %0d violated", DLY);
6
```

Figure-13 Need to be careful while using $past in assertions

The issue with above code is: “What does the $past return in clocks prior to DLY”? In simulation, the answer is X. That could cost some debug cycles. However, in FV tools, this can lead to false failures as the tool is free to assume any value to the ***req*** signal in this example in clocks prior to DLY (DLY is a parameter, say 5).

A recommended solution to this problem is to delay the checking of such properties by appropriate number of clock cycles. A better code that fixes the above problem is shown in Figure-14.

```
6
7  a_p_better_code : assert property( ##DLY $rose(ack) |-> $past(req,DLY))
8   else
9      `uvm_error ("VW_CIP", "Expected delay of %0d violated", DLY);
10
```

Figure-14 Add ##delay to the start of the property to avoid uninitialized values

Unit test each assertion

Assertions are "checkers" of your design, or the core of "verifier" – but who will verify *that* verifier? Given that a comprehensive CIP is complex code, it requires thorough verification itself. We have developed a series of unit tests inside Go2UVM framework [8] to tackle this problem. In a nutshell, this involves creating pass and fail trace for each assertion with a simple UVM test. These unit tests should be smart enough to be self-checking. We have used a UVM report mocker from open-source SVUnit framework [9] to self-check each unit test around assertions.

Consider AHB protocol requirement on *htrans* signal as shown in Figure-15 below:

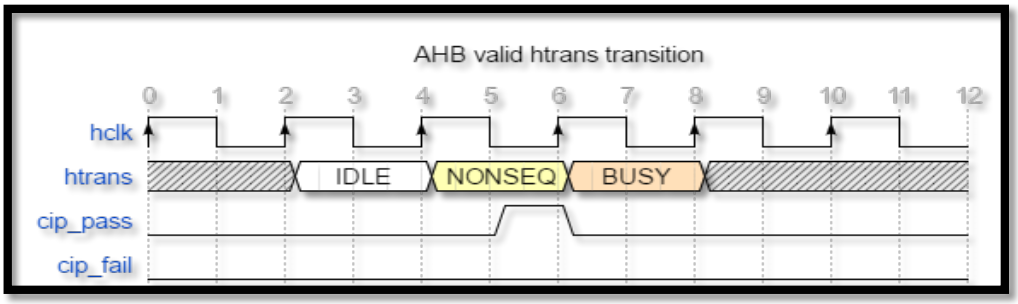

Figure-15 AHB *htrans* requirement

A representative property for this requirement in SVA is shown below in Figure-16.

```
82
83 // After an IDLE transfer,next clock transfer type be either IDLE or NSEQ
84 property after_idle_next_clk_goto_idle_or_nonseq;
85   @(posedge hclk) disable iff(!hresetn)
86   (htrans == 'b00) && hgrant |=> ((htrans == 'b10) || (htrans == 'b00));
87 endproperty:after_idle_next_clk_goto_idle_or_nonseq
88
```

Figure-16 AHB *htrans* check

Now to ensure that the above assertion indeed "verifies" the arbiter, one needs to test this code. To be compete in the testing one needs to consider pass and fail scenarios. Some of the fail scenarios are captured below in Figure-17 and Figure-18.

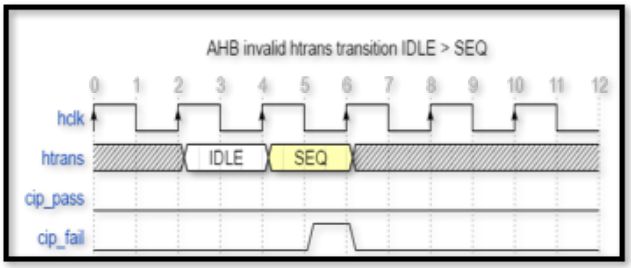

Figure-17 AHB *htrans* invalid transition - 1

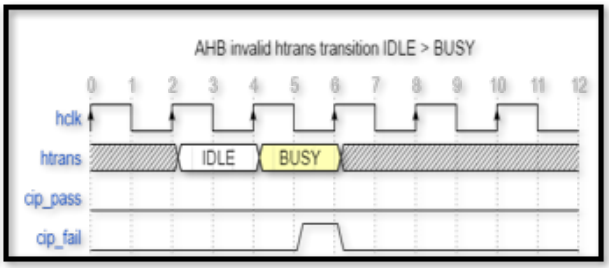

Figure-18 AHB *htrans* invalid transition - 2

Once we identify the required scenarios, question is can we code them in UVM framework? A full-fledged UVM bench for this testing would be an overkill.

We have used a simple test layer on top of UVM, named Go2UVM [8] test as shown in Figure-19 below:

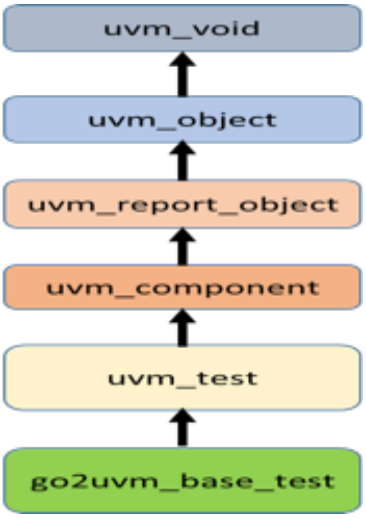

Figure-19 Go2UVM test base class

With Go2UVM package, one need not worry about objections, various components etc. We can write quality traces with very few lines of code. A pass trace for the above assertion in Go2UVM framework is shown below in Figure-20.

```
41
42   /*
43   Go2UVM Unit test for the following property
44  // After an IDLE transfer,next clock transfer type be either IDLE or NSEQ
45   property p_idle_or_nseq;
46     (htrans == ahb_transfer_kind_e'(IDLE)) && hgrant |=>
47     ((htrans == ahb_transfer_kind_e'(NONSEQ)) || (htrans == ahb_transfer_kind_e'(IDL
E)));
48   endproperty : p_idle_or_nseq
49   */
50
51   task main ();
52     `uvm_info (log_id, "Start of main", UVM_MEDIUM)
53     `uvm_info (log_id, "p_idle_or_nseq PASS trace IDLE --> NONSEQ", UVM_MEDIUM)
54     this.vif.cb.hgrant <= 1'b1;
55     this.vif.cb.htrans <= ahb_transfer_kind_e'(IDLE);
56     repeat (5) @ (this.vif.cb);
57     this.vif.cb.htrans <= ahb_transfer_kind_e'(NONSEQ);
58     repeat (1) @ (this.vif.cb);
59     `uvm_info (log_id, "End of: p_idle_or_nseq PASS trace IDLE --> NONSEQ", UVM_MEDI
UM)
```

Figure-20 AHB *hrtans* Go2UVM test (PASS trace)

The FAIL traces – or the negative scenarios during which the assertion is supposed to fire are even more important to verify. Coding them in Go2UVM is fairly straight forward. But the challenge there lies in automating these unit tests and classifying them as PASS or FAIL (of the test, an intended negative test is supposed to fail with UVM_ERROR for instance). In unit tests, we need to declare PASS/FAIL automatically based on the user's intent. This challenge is more than typical DUT PASS/FAIL declaration (that could be based on presence of UVM_ERROR in log file).

Unit tests inject "error scenarios" by definition Manual classification of expected UVM_ERRORs is not feasible. We used a ***uvm_report_mock*** feature originally developed by Neil Johnson as part of his SVUnit [9]. This base class is now part of latest Go2UVM package as well. Consider the following trace in Figure-21 for the *htrans* assertion:

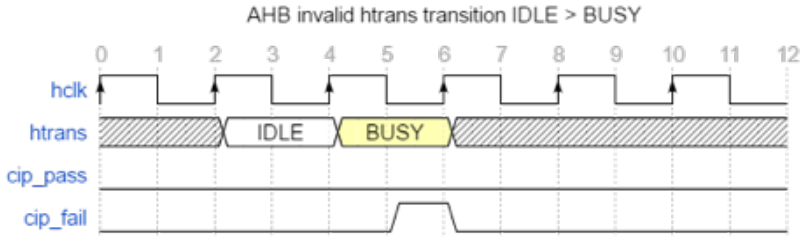

Figure-21 AHB *hrtans* invalid transition (negative test)

At clock tick 6, we would expect the assertion to fire. If we run the trace as-is, it reports an UVM_ERROR like shown below:

```
UVM_ERROR../vw_cip_src/vw_ahb_lite_cip.sv(134) @ 175.00 ns: reporter [SVA] Invalid
htrans transition - from IDLE only NSEQ is allowed. Assertion 'a_p_idle_or_nseq' FAILED
at time: 175ns (18 clk), scope:vw_ahb_lite_cip_go2uvm.vw_ahb_lite_cip_0,  start-time:
165ns (17 clk)
```

Figure-22 Sample UVM ERROR due to FAIL trace (negative test)

A very useful package called ***uvm_mock_pkg*** is available as part of open-source ***SVUnit***. At a high level this package provides three key features:

- A callback to capture all UVM errors coming from the simulation
- An API to specify expected errors by the end user
- An API to compare the expected and actual errors at the end and flag any mismatches

uvm_report_mock::expect_error() API internally populates queue of "expected errors". For instance, in our AHB CIP unit test we call the ***expect_error*** for negative trace as shown in Figure-23 below:

```
60      this.vif.cb.htrans <= ahb_transfer_kind_e'(IDLE);
61      repeat (5) @ (this.vif.cb);
62      this.vif.cb.htrans <= ahb_transfer_kind_e'(SEQ);
63      repeat (2) @ (this.vif.cb);
64      uvm_report_mock::expect_error("SVA");
65      `uvm_info (log_id, "End of: p_idle_or_nseq FAIL trace IDLE --> SEQ", UVM_MEDIUM)
66      this.vif.cb.htrans <= ahb_transfer_kind_e'(IDLE);
67      repeat (5) @ (this.vif.cb);
68      // uvm_report_mock::expect_error();
69      this.vif.cb.htrans <= ahb_transfer_kind_e'(BUSY);
70      repeat (2) @ (this.vif.cb);
71      // TBD find a better way to handle this
72      go2uvm_test_fail_count += (!uvm_report_mock::verify_complete());
73      `uvm_info (log_id, "End of: p_idle_or_nseq FAIL trace IDLE --> BUSY", UVM_MEDIUM
)
74    endtask : main
```

Figure-23 Self-checking Go2UVM unit test with ***expect_error*** call

With this handy report mocker our unit tests are self-checking. We strongly believe a good CIP should be accompanied by a quality set of unit tests that can be regressed anytime there is a bug fix to the CIP.

IV Summary

Design Verification with SystemVerilog assertions has been popular in the industry for well over a decade. While simple checkers can be developed quickly and used across design entities, a comprehensive CIP (Checker IP) takes a good architecture and set of coding guidelines to keep them reusable. In this paper, we have shared our experience of converting a plain set of properties to a reusable CIP. We also shared how we used a self-checking unit test framework to verify each assertion in a CIP.

REFERENCES

[1] SystemVerilog LRM - http://standards.ieee.org/getieee/1800/download/1800-2012.pdf
[2] ARM AXI specification – https://www.arm.com/products/system-ip/amba-specifications
[3] ARM releases assertion models - https://www.arm.com/about/newsroom/12266.php
[4] Experiencing Checkers for a Cache Controller Design
http://systemverilog.us/DvCon2010/DvCon10_Checkers_paper.pdf

[5] Accellera Open Verification Library (OVL) http://accellera.org/activities/working-groups/ovl
[6] SystemVerilog Assertions handbook, www.systemverilog.us, www.verifnews.org/publications/book
[7] "What are $past compared to on first clock event?" http://bit.ly/2hkb7nV
[8] Go2UVM open-source test layer, www.go2uvm.org.
[9] SVUnit - http://www.agilesoc.com/open-source-projects/svunit/