Architecting “Checker IP” for AMBA protocols

Ajeetha Kumari, Verification Consultant, VerifWorks Pvt. Ltd.
Srinivasan Venkataramanan, Verification Technologist, VerifWorks Pvt. Ltd
Agenda

• CIP – Introduction
• What to Verify in Assertion
• AHB CIP Example
• AXI3 Architecture
• CIP Guidelines
• Using Go2UVM Framework
• Using SVUnit
• Summary
• References
INTRODUCTION CIP

• CIP – Checker IP
  – Set of assertions for a given protocol
• AHB requirement - *htrans* signal from an AHB master.
• The signals *cip_pass* and *cip_fail* indicates state of assertions capturing this requirement.
  – IDLE $\rightarrow$ NONSEQ == legal
To enumerate the fail scenarios, other possible transitions of \textit{htrans} need to be explored.

In these figures the signal \textit{cip\_fail} goes HIGH indicating a protocol violation.
WHAT TO VERIFY IN ASSERTIONS?

• Assertion aspects
  – **Intent**: reflects the system engineer or designer's understanding or vision of what is desired, as defined in written, assumed, unsaid, or implied requirements, many of which may be loosely (or tightly) specified in timing diagrams and in engineers' heads.
  – **Accuracy**: deals with the proper expression of the requirements with emphasis on coding rules, style, and coverage of intended cases.
  – **Efficiency**: deals with coding that puts too much unnecessary overhead on the simulator because of unneeded threads.
  – **Purpose**: addresses the uses or application of the verification environment.
AXI3 CIP Architecture

Component with AXI master

- **AW**
  - Write channel address and control
  - AWVALID
  - AWREADY

- **W**
  - Write channel data
  - WVALID
  - WREADY

- **B**
  - Write channel response
  - BVALID
  - BREADY

- **AR**
  - Read channel address and control
  - ARVALID
  - ARREADY

- **R**
  - Read channel data
  - RVALID
  - RREADY

Component with AXI slave

- **AW**
- **W**
- **B**
- **AR**
- **R**

Write channels

Read channels

AXI3 Master

AXI3 Slave

- AWID
- AWADDR
- AWSIZE
- AWBURST
**Property as checker**

- `a_p_vw_awburst: assert property p_vw_awburst`

**Property as constraint**

```
ifdef M_CIP_SIM
else
  `vm_error [vj_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
endif //M_CIP_SIM
```

```
map_vw_awaddr_boundary: assume property (p_vw_awaddr_boundary);
map_vw_awaddr_wrap_align: assume property (p_vw_awaddr_wrap_align);
map_vw_awburst: assume property (p_vw_awburst);
map_vw_awcache: assume property (p_vw_awcache);
map_vw_awlen_wrap: assume property (p_vw_awlen_wrap);
```

**No of properties**

- Master Checkers
- Master Constraints
- Slave Checkers
Guidelines for CIP

• Use of `checker...endchecker` construct
• Use `module` as a container
• Use `bind` construct
• Use of `interface` as a container
• Use text macro in action blocks
• Use appropriate delays in value change functions
Assertion styles

• RTL designers – inlined checks
• Verification engineers – external using *bind*
• A set of small assertions is usually better than single large assertions
• Instantiations
  – Concurrent
  – Immediate
  – Procedural concurrent
• How to insert a group of assertions inline in RTL?
Using `checker...endchecker`

- SV 2009 added new container for assertions
  - `checker`
- Offers flexibility in terms of instantiation
  - All 3 forms – immediate, concurrent & procedural concurrent
- Useful for smaller checkers (OVL like)
- Support “free/rand variables”
  - Great for pure formal verification tools
- For CIP not recommended because of language restrictions
  - No parameters
Using *module* as container

- SystemVerilog *module* is well known construct
- Easy to use/code
- Modules can instantiate other modules
  - Direct
  - Indirect via *bind*
- Assertions can be coded inside modules
Problem with module as CIP container

• Can NOT bind a *module* to an *interface*
• Interface is widely used construct
Interface as assertion container

- Interface is a popular construct in SV
- Widely used for communication between dut and tb
Text macro in action blocks

• Text macros can be used to insert a block of code which is useful in simulation to make them FV friendly and vice versa
• VW_CIP_SIM is used
Goal: develop unit tests in UVM framework

Problem: UVM is too big for this task at hand

Solution: open-source Go2UVM package.
• **Self-checking of unit tests through SVUnit’s UVM Report Mock:**
  - 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
At clock tick 6, we would expect an 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)
```
Using repor_mock API

repeat (1) @ (this.vif.cb);
  `uvm_info (log_id, "End of: p_idle_or_nseq PASS trace IDLE --> NONSEQ", UVM_MEDIUM)
this.vif.cb.htrans <= ahb_transfer_kind_e'(IDLE);
repeat (5) @ (this.vif.cb);
this.vif.cb.htrans <= ahb_transfer_kind_e'(SEQ);
repeat (2) @ (this.vif.cb);
  uvm_report_mock::expect_error("FAIL");
  `uvm_info (log_id, "End of: p_idle_or_nseq FAIL trace IDLE --> SEQ", UVM_MEDIUM)
this.vif.cb.htrans <= ahb_transfer_kind_e'(IDLE);
repeat (5) @ (this.vif.cb);
  // uvm_report_mock::expect_error();
this.vif.cb.htrans <= ahb_transfer_kind_e'(BUSY);
repeat (2) @ (this.vif.cb);
  // TBD find a better way to handle this
  go2uvm_test_fail_count += (!uvm_report_mock::verify_complete());
  `uvm_info (log_id, "End of: p_idle_or_nseq FAIL trace IDLE --> BUSY", UVM_MEDIUM)
endtask : main

endclass : vwb_ahb_lite_cip_test
Summary

• Simple checkers can be developed quickly and used across design entities,
• Comprehensive CIP (Checker IP) takes
  • a good architecture
  • 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.
  • How we used a self-checking unit test framework to verify each assertion in a CIP.
References

1. SystemVerilog LRM -  


3. ARM releases assertion models -  
   https://www.arm.com/about/newsroom/12266.php

4. Experiencing Checkers for a Cache Controller Design  

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


Q & A

Thanks