Novel Verification Techniques for ARM A15 Multi-core Subsystem Using IEEE 1647
Agenda

• Introduction
• Goals and Requirements
• Challenges Overview
• Testbench Architecture
• Coherence Verification
• Getting It Done with Specman/e
• Summary
Introduction

Modern designs are becoming bigger and more complex, yielding revolutionary and evolutionary products in the user space. This trend manifests in longer DV cycles and tougher challenges. Specman/e based eRM provides an elegant solution for handling these issues efficiently.

Here we present our testbench methodology and the application of Specman/e in a ARM® Cortex® -A15 multi-core subsystems verification effort.
Goals and Requirements

- **One environment for all variations of ARM Cortex-A15 multi-core subsystem.**
  
  Unified verification environment for all planned ARM Cortex-A15 multi-core subsystem configurations: quad-core, dual-core, and single core.

- **Re-usable ARM software library/testcase/verification assets.**
  - Maintain a compatible system programming view.
  - Plug and play unit level tests for SoC DV.
  - Re-usable unit level verification assets at SOC level: checkers/assertions.

- **Testbench must be portable to HW emulator/FPGA.**
  - TB must be synthesizable.
  - Synthesizable TB must reflect full system topology.

- **Coverage Driven Verification (CDV).**
  - Support CDV: functional, code, and assertions coverage.
  - Minimal integration effort: TI’s internal eVCs + external/3rd party VIPs.
  - Standardized independent multi-users code development and coverage closures.
Challenges Overview

How do we address all these diverse and exclusive requirements with one common environment?

• TB that resemble full system programming view
  – Critical for software development platform for SoC.
  – Critical to enable SoC bug replication.
  – Implies topological similarity between unit-level TB and SoC.
  – Implies TB configuration complexity, which is an additional overhead for DV engineers to comprehend.

• Synthesizable TB but yet support CDV
  – Must have synthesizable TB to support emulation/prototyping for performance analysis.
  – CDV requires high level verification languages (HVL) such as Specman/e, Vera and SystemVerilog as used in VMM/UVM/OVM.
  – HVLs best suited for TB but they are not synthesizable.
TB/Env Architecture

TB Env

- System boot configuration
- Power reset clock controller
- One time fuse setup
- System config and memory bus
- Interrupt handler
- Memory BIST controller

fully synthesizable TB, topologically equivalent system level view

Layer 1: Synthesizable TB (Verilog)

DUV
TB/Env Architecture

Layer 1: Synthesizable TB (Verilog)

- Physical layer Programming Interface (PPI)
- Configuration utilities/system calls to the TB physical layer.
- Inter-layers communication protocol (between TB layers).
- Monitoring portal the physical layer/DUV.
- An abstraction layer (hide all protocol/sequencing details from CDV).

Layer 2: PPI (e/specman)
TB/Env Architecture

Layer 1: Synthesizable TB (Verilog)
- Bus Functionality Modules
- Stimulus Generation Sequences
- Score boards/Monitors/Checkers
- Functional Coverage Bins

Layer 2: PPI (e/specman)

Layer 3: CDV layer (specman/e)
TB/Env Architecture

- **Layer 1: Synthesizable TB**
  - Checker
  - Scoreboard
  - Coverage

- **BFM**
  - PPI Layer
  - Protocol BFMs
  - Shared Memory
  - Checkers/Monitors
  - Scoreboard

- **PPI Layer**

- **TB Environment**

- **Specman/e Component**
  - PPI Layer
  - Protocol BFMs
  - Shared Memory
  - Checkers/Monitors
  - Scoreboard
TB/Env Architecture

Simulation Run
- TB/Env configuration
  - Setup physical layer
  - Config eVC active/passive
  - Load program code
- DUV Initialization
  - Tieoff Drive
  - Power up sequence
  - Fuse
  - Reset calls
- Processor Boot
  - Shared memory response
- Test Activity
  - BFM Drive
  - Program execution
  - Monitor response
  - Scoreboarding/Checking
  - Coverage Collection
- Reporting
  - Test Completion checks
  - Program Status
  - Error reporting

Layer 1: Synthesizable TB
- Checker
- Scoreboard
- Coverage

TB Environment
- BFM
- PPI Layer
- DUV
- Monitors
- Shared Memory
Coherency Verification Requirements

• Generating all the transactions a protocol component may generate
  — 29 Different type of Read (14), Write (7) and Snoop (8) defines in ACE depending on the state of the cacheline.

• Generating all the freedoms of the protocol in transaction responses, for instance, ACE Read, Write and snoop responses.
  — The possible snoop response (~45) depend on the state of the snooped cacheline
Generating all the cache-line conflict cases where multiple coherent agents are operating on the same cacheline.
Coherency Verification Challenges in SoC

- **Number of Scenarios**: Meeting coherency verification requirements become manifold in SoC’s where multiple coherent masters are interacting.
  - Complexity & Interactions between coherent masters.
    - Possible ACE crossings \(\sim (7000 \times N)\)\(^1\) where \(N\) is number of Masters in the system.

- **Transaction Generation**: Generating ACE transactions, possible responses and conflict cases is non trivial when initiating master is A15
  - Controllability is not the same as compared to a Coherent BFM/VIP for unit level verification.

- **Data Integrity**
  - Concurrent Access:
    - Data integrity becomes critical when coherent masters modify the same cache line at the same time.
      - How to track data ordering on concurrent cacheline access, is there a workaround?

---

\(^1\) Multicore ARM SoCs Face Cache Coherency Dilemma : Featured Techtalk, Mirit Fromovitch, Cadence
Coherency Verification Challenges in SoC

- **Timing of overlapped accesses:**
  - Generating conflict case of ACE transaction overlappings.
  - How to ensure that the overlapped scenarios was correctly generated?

![Diagram showing priorities](Image)
Coherency Verification: From SS to SoC

Soc Environment

Layer 1: Synthesizable TB

Checker
Scoreboard
Coverage

BFM

Coherent Masters (Soc)

DUV

Monitors

Mem Ctrlr (Shared RAM SoC)

API

Specman/e Component

Swapped Env Components
- PPI Layer
- Protocol BFMs
- Shared Memory
- Scoreboard

Components retained:
- Checker
- Protocol Coverage
- Monitor
Coherency Verification: From SS to SoC

- **Functional Coverage**
  - Number of Scenarios
  - Timing of conflict cases
    Assertion based cover property to ensure that timing based non-deterministic, conflict cases are hit.

- **Correctness Model**
  - Protocol checkers
  - False Sharing
    Data Integrity during basic concurrent access: Use false sharing and read modify write to overcome the issue of data checks for concurrent writes
    - Each Coherent master owns a portion of the line does a partial line write
    - Checker written to observe a incrementing pattern for each byte of the line.

```
+--------+--------+        +--------+--------+
| M1     | M2     |        | M3     | M4     |
| Byte1  | Byte2  |        | Byte3  | Byte4  |
| 0x0-0xFF | 0x0-0xFF |        | 0x0-0xFF | 0x0-0xFF |
```

```
+--------+        +--------+        +--------+        +--------+        +--------+
| C1     |        | C2     |        | M1     |        | M2     |        | M3     |        |
| Byte5  |        | Byte6  |        | Byte1  |        | Byte2  |        | Byte3  |        |
| 0x0-0xFF |        | 0x0-0xFF |        | 0x0-0xFF |        | 0x0-0xFF |        | 0x0-0xFF |
```
Getting It Done with Specman/e

• **Integration Flexibility**
  – Coexistence of synthesizable and non-synthesizable view.
    • Clean topology partitioning: synthesizable TB and Specman/e components.
    • Decoupled compile flow.
  – Port mapping of verification component based on TB view.

• **Debug efficiency**
  – Modifying Specman/e code without TB recompile
  – On-the-fly override of existing methods, events, sequences.

• **Use of ‘defined as’ macro**
  – Duplication of coverage bins: minimal coverage codes.
  – Conditional commenting of pre-string instead of if-def construct

• **Well Integrated with Cadence Incisive**
  – Sys Object browser: hierarchical view of instantiated e structure, hookup, state.
  – Event waveform allow execution flow trace. (activity across simulation timeline)
  – Command prompt allows execution function/methods in between simulation.
Getting It Done with Specman/e
Getting It Done with Specman/e

item bin1_modA : uint(bits:2) = trans.bits_modA[15:0] using ranges =
{
    range([0x00], "bits OFF", UNDEF, 50);
    range([0x0F], "bits ON", UNDEF, 50);
};

item bin2_modA : uint(bits:2) = trans.bits2_modA using ranges = …

item binN_modA : uint(bits:2) = trans.bitsM_modA using ranges = …

Substituted by macro

INSERT_MY_COV_ITEM  modA;

‘define as computed’ Macro usage
Summary

- We successfully met coverage requirements with Specman-e.
  
  Coverage Type | Total Points/Bins | Covered *
  --- | --- | ---
  Functional | ~85K | 97 %
  Block/Line | ~700K | 83 %
  Toggle | ~3M | 99 %

  * Covered bins with excludes and review

- We created a unified DV environment that supported quad-core, dual-core, and single-core ARM SS.
- Reuse Metrics.
  
  Components | IP to SoC
  --- | ---
  Test cases | ~50%
  Functional coverage bins | ~4K
  Checkers | ~300
  Software Library Functions | 100%

- We delivered plug and play ARM Cortex-A15 SW library and tests to SoC teams.