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

Vaibhav Mahimkar, Akshit Dayal, Tomas Huynh, Van Huynh, Erwin Hermanto.
EP-Processors, Texas Instruments Incorporated, Stafford, Texas

***Abstract*— 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.**

Index Terms—Specman/e, eRM, ARM, CortexA15, Configurable Testbench.

I. INTRODUCTION

Considering the various configurations of the design (DUV) and its variants, the verification environment capable of verifying these different configurations of the design subsystem and be scalable enough to verify future generation variants of the design was needed. Successive iterations of the designs can have modified functional behavior and such variations were to be supported by the same testbench verification environment while simultaneously maintaining backward compatibility with older versions of design. This enables running the same set of tests across all configurations as well as future variants of design.

Once a regression test suit is created that is robust and comprehensive enough to hit all the required functional scenarios and coverage bins then it can be ported over to a different design variant allowing coverage collection against same set of scenarios.

In the simplest form design feature variations involved changes to number cores, power control features, debug hardware, proprietary and legacy interfaces and their scaling with respect to the number of cores.

Having such an all-encompassing verification environment would allow using common test cases and other common verification assets developed across all design variants.

We wanted these components which are developed at IP level environment to be reused at System on chip (SoC) level environment as well. In order to the do this the system level programming view as seen by the design at IP level environment had to be compatible and identical to what it would see in the actual SoC that it was intended to be used in. This would allow the software test cases and basic library components to be plugged into SoC environment giving a significant reduction in integration testing time in those environments and software development. In addition the components such as custom checkers and assertions from IP level could be ported over to SoC environment as well.

In order to be able to simulate the verification scenarios on Hardware emulator or FPGA based platforms the environments needed to have synthesizable components. This would allow easy reproduction of test scenarios across various simulation platforms. Irrespective of the platform the system level topological view seen by the design had to be identical.

We wanted the environment to support high level verification language (HVL). Use of HVL would provide advanced capabilities such as randomization of tests and complex sequences and scenario generation. This would also provide coverage driven verification environment and thus allow gauging the completeness of the verification effort. Standard verification components and Bus Functionality Modules (BFM) available for HVL languages from third party vendors as well as the proprietary ones created internally can be integrated easily into the environment and ensure easy interaction with each other. HVL provides a standardized way for test development which allows multiple people to work on test development for different features of the design simultaneously.

II. CHALLENGES OF IMPLEMENTATION

The requirements mentioned above are somewhat diverse in nature and exclusive with respect to each other. For instance having system level view presented to design that resembles the one faced inside the SoC is essential for various reasons mentioned above sections like software, testcases, assets development and reuse across IP level and SoC level view. Additionally, having such capabilities at the IP environment allows easy replication of SoC level bugs for faster debug. However, having such capability at the IP environment makes it complex, as a result the verification engineer has to comprehend the details of how the testbench is implemented in addition to the design understanding. Similarly, having a testbench which would provide the portability across various simulator platforms with identical system view and at the same

time having HVL components for advanced verification features are somewhat contradicting since the HVL components are non synthesizable and the portability to hardware and FPGA platforms need synthesizable elements to be present in the testbench. Our testbench architecture has distinct synthesizable and non synthesizable component layers which enable portability across multiple platforms.

III. Environment Architecture and Implementation

To achieve these capabilities a layered testbench approach was adopted. The different layers are partitioned according to the level of abstraction they represent as explained below.

Layer 1:

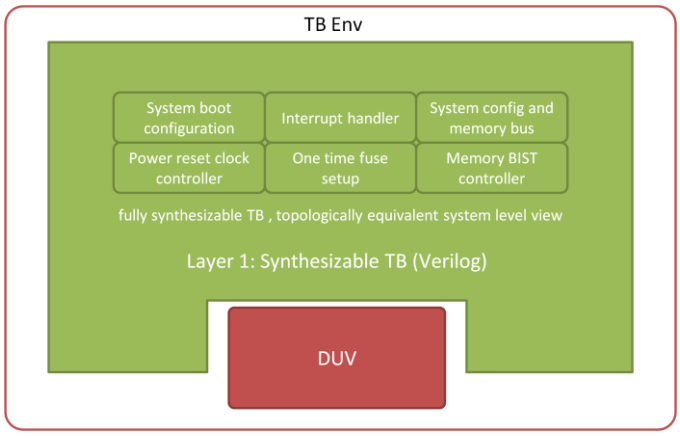

Figure 1. Layer 1

This is the primary/base layer and is placed closest to the DUV. Figure *1*. Layer 1 shows this layer. It interacts directly with DUV's interfaces at protocol/physical level and represents the signals activity such as control, handshake, bus values. Components of this layer are cycle accurate representation of components that the DUV would be interfaced with inside the SoC. Blocks such as power, reset, clock, fuse controllers are example of such components and these provide the setup and initialization capabilities of the DUV via Layer 1. In our case, since the design is a processor based subsystem additional blocks like boot controller, interrupt handlers, bus and memory system controllers around the design provide the complete SoC identical topology from a Software programming perspective. All the components of this layer are synthesizable and can be ported over to various platforms allowing the DUV to have a consistent system level view across all platforms.

Layer 2:

Higher level layers represent functional abstraction of equivalent protocol/physical activity carried out at base layer (layer 1). Second layer surrounds the synthesizable layer 1 and is implemented using HVL, in our case using Specman/e. See Figure *2*. Layer 2. This layer provides system level calls which in turn interact with protocol specific components of Layer one in sequenced manner. For example a power on reset could be implemented as a system call in this layer, which when invoked would configure the reset, clock, boot components of layer 1 to create a valid power on reset activity on the DUV. Such functions hide the details of the implementations of physical layer from the higher layer. Any changes to design at protocol/physical level to be absorbed by the functions of this layer. As a result this intermediate layer serves as a programming interface to the physical Layer1 and hides all the protocol level details and sequencing from the higher layer.

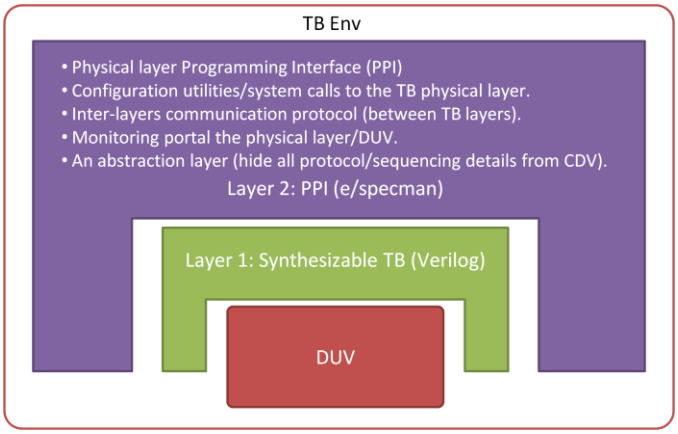

Figure 2. Layer 2

Layer 3:

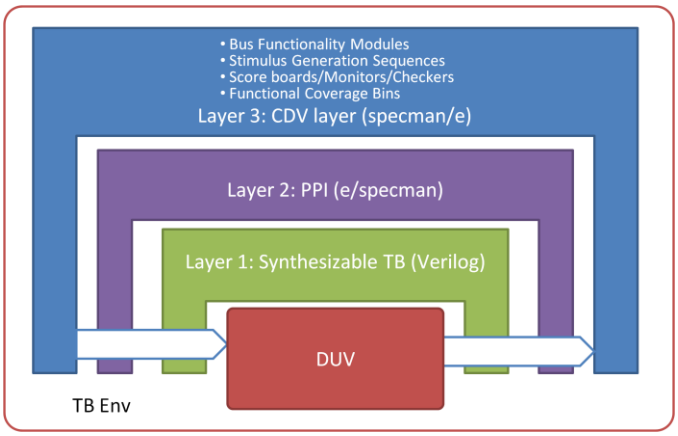

Figure 3. Layer 3

Third layer is the top most layer that provides the HVL features such as coverage metrics and scoreboarding and is implemented in Specman/e. See Figure *3*. Layer 3. This is where standard/proprietary BFMs are integrated and used to create test sequences and custom checkers specific to the feature being tested. Verification engineers can use the BFM sequences specific to the functionality they are testing in addition to high level system calls provided by layer 2. This allows simultaneous and independent test development targeted towards the specific functionality.

Each of these layer can be configured at setup time to customize for different scenarios of a particular design variant as well as variations in behavior between two design variants. Moving from one design to other requires a configuration change which is achieved by feeding in the specific configuration file.

IEEE 1647 ***e*** language facilitates having clean partitioning between different layered components necessary for coexistence of synthesizable and non-synthesizable views of the environment.
Combination of various layers above provide the test writer a high level and seamless way to interact with the design allowing reuse of developed assets as well as coverage driven metrics in one common environment.

IV. Memory Coherency and Consistency

Memory operations in a uniprocessor are assumed to execute sequentially wherein the load operation returns the last store to the same location and memory operations execute in order as specified in program **[2]**. In a heterogeneous processor system, the same sequential order needs to be preserved to the same memory location without any software overhead or added complexity in executed program. Even though some or all the processors in a system can execute instructions out of order (as long as they do not cause any hazards), the illusion of sequential consistency of memory operations to the same location needs to be preserved. What this implies is that the memory view by each processor in the system is same as if it was the only processor in the system and the stores complete in sequential order specified in program and loads return the last value written to a memory location by any coherent master.

The hardware feature which suffices the rules outlined by the memory consistency model is complex and that added hardware complexity percolates in the verification process. One such protocol which works within the framework of the memory consistency model is the ARM AMBA ACE protocol and is the focus of our verification targets. We further discuss a reusable DV infrastructure across a configurable multicore A15 subsystem and its portability for verification in a heterogeneous cache coherent multiprocessor system on chip.

V. Memory Coherency Verification Requirements

The requirement for coherency is outlined at a subsystem and SoC level to appreciate the requirements for each and how requirements map back from SoC to subsystem verification effort.

A. Transaction Generation

The ACE protocol supports 29 different type of ACE transactions on the read, write and snoop channel. Each depends on the page table attribute of the region the cacheline/block belongs to and the state of the block within the local cache of a cached master i.e. whether it is in Modified, owned, exclusive, shared or invalid state. Furthermore, the type of coherent transaction also is determined by the fact if it's a full vs a partial cacheline access by the master. Lastly the type of transaction varies if the initiating coherent master can cache the accessed line or not. These represent the various knobs to tweak while configuring a BFM to mimic the transaction by a coherent master via an equivalent VIP in the unit level verification environment or as parameters if the transactor is RTL representation of a coherent master in a SoC verification environment. The various groups or classes of transaction are shown in the diagram below, which the transactor must be able to produce.

Fig. 4. Read/Write/Snoop channel transactions)

B. Response Generation

Response generation alludes to the generation of all the degrees of freedom of responses for every permitted transaction by the master. This again depends of the state of the block of cache memory accessed. For instance there is a possibility of 5C1 responses for each of the snoop transactions and are dependent on the state of the cacheline being snooped.

C. Concurrent Memory Collision Generation

The prior points indicate the requirement for every coherent master which might harbor a local cache or not (for performance improvement). In addition there should be scenarios which check the validity of the protocol or the master complying with the protocol by generating conflict cases, wherein, the crux of the memory consistency model tested. This implies concurrently accessing the same block of memory while preserving store order and observability of the last store upon a read operation by the coherent.

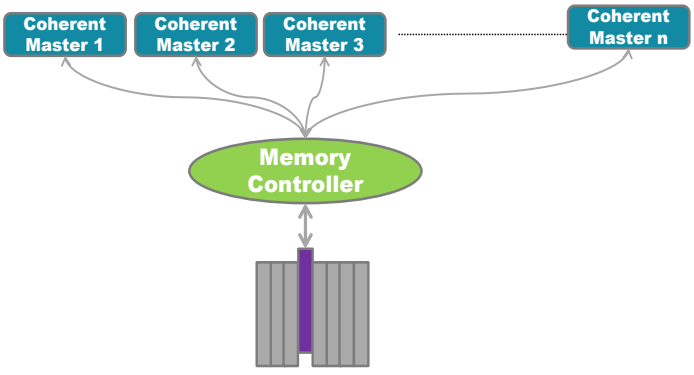

Fig. 5. Concurrent memory access by coherent masters.

VI. Coherency verification challenges in SoC

Coherency verification imposes many challenges when verifying in a multi-coherent master heterogeneous processor system. The complexity increases many fold and controllability is greatly reduced upon swapping a BFM with actual HDL described design component. The challenges play a vital role in molding the verification strategy.

1) Protocol Permitted Possible Scenarios: Meeting coherency verification requirements become manifold in a system with multiple coherent masters. Complexity apart, number of all possible scenarios increases linearly with number of masters. The possible number of ACE crossings for an 'N' Coherent master system is roughly (7000*N) [1].

2) ACE Transaction and Response generation: Generation of ACE transactions using the real design of coherent master vs BFM is non trivial and is more involved than using a BFM to thrash out the ACE transactions. It requires a lot of Software overhead and added effort for verification in a SoC. In SoC's this overhead arises due to software sequencing for setting cache warming, page tables setup, LPAE setup, setup of the global snoop monitor in the system. Additionally, software sequence to program the other coherent master in the system is required to generate all the flavors of ACE transaction and coercing all combination of responses.

3) Timing and Controllability of concurrent transaction generation: Unlike unit level testbenches the transaction generation is not fine grained enough to have a cycle by cycle control over scenario generation. For instance if one wants to test out the overlap of ReadOnce snoop after a WACK for a WriteClean is received but before the slave returns the BRESP for it, the window of opportunity is just a few cycles. Thus it becomes very difficult to control boundary cases of overlapped scenarios.

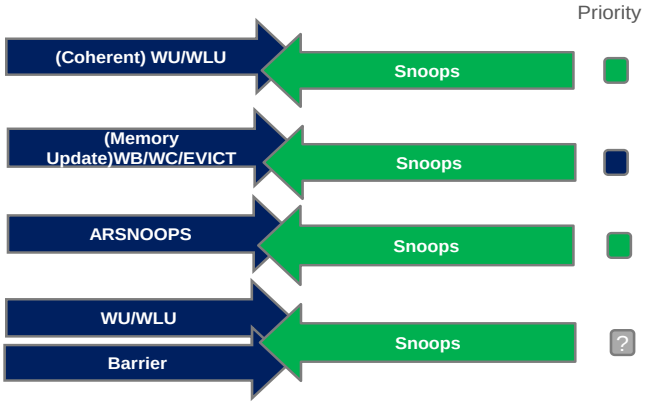

Fig. 6. Memory Access conflict scenarios.

4) Data integrity: The nature of memory consistency verification requires the generation of concurrent accesses to shared memory location. Most SoC tests rely on self-checking as one of the correctness models for test scenarios, which is nullified by the nondeterministic nature of concurrent access. Due to concurrent access and the lack of fine grain controllability in SoC, it is difficult to control the order of concurrent accesses; hence it is not known which coherent master was able to order memory access ahead of others. This is one of the key components of the correctness model without which verification of memory consistency model is incomplete.

VII. Coherence verification: joining dots from Subsystem to Soc

We laid out the case of coherence verification and the challenges within the scope which increases linearly with coherent masters [1] within the verification scope. We enumerate the steps geared towards tackling them.

1) Reusable-Configurable Testbench: The reusable and configurable testbench has an API layer which enabled the DUT to interact with BFM's and also supplies configuration information to monitors which are fanned out to other DV assests. This API layer enabled us to reuse our tests and testbench DV assests like functional coverage, protocol checkers, assertions and monitors. It also enabled us to auto-generate our tests which are pseudo-random in nature. The API carries encoded information to the external coherent masters at SoC or PPI layer at the subsytem layer. The encoded information describes the current state of cacheline, page table shareability attribute, miss policy, allocation hint, the external coherent master id, randomization of the type and byte-size of the ACE transaction. This API layer based test generation ensures reusability of test, software library, and various test bench components. Furthermore, the test and software library were directly portable with minimal change from subsystem, to SoC pre-silicon verification and even post silicon validation. The reusability is a key component to avoid reinventing the wheel and reworking the verification effort across multiple A15 configurations and also in a heterogeneous processing system constituted of ARM CPUs and TI C6x DSP's.

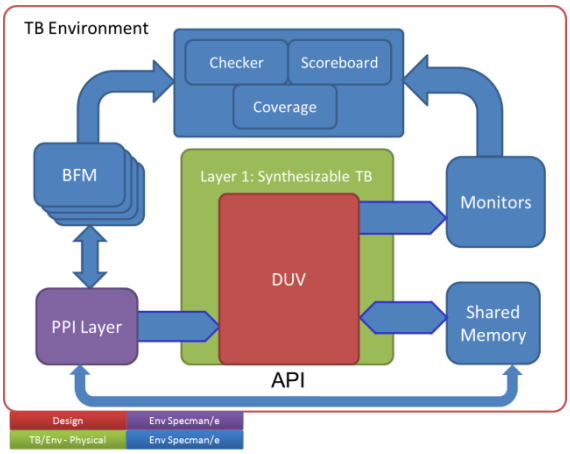

Fig. 7. Unit Level TB: API layer interacting with PPI Layer

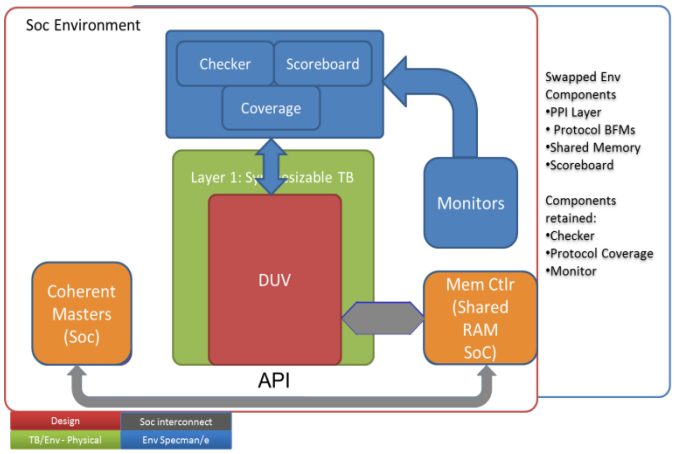

Fig. 8. SoC level TB: API layer interacting over switch fabric to other coherent masters

2) Functional Coverage: Functional coverage was a tracking metric to ensure that we are hitting all possible scenarios and corner cases. The specman-e based functional coverage was directly portable from unit level to SoC testbench env. Furthermore, specific assertions targeting timing based conflict cases were used for the cover property and incorporated into functional coverage. This ensured that timing based non-deterministic, conflict cases are hit and triggered a failure if a specific directed test was unable to meet the test objective.

3) *Correctness Model:* The correctness model for memory consistency problem should be two fold primarily. For each, shared memory location the order of stores should be observed in program order and each load should return the last value written at that memory location. The ordering aspect of the memory consistency problem is solved by writing protocol checkers that checks the compliance of the coherent master to the ACE protocol and the same is reused at the SoC env for pre-silicon verification. For ordering to be observed true sharing is considered sufficient enough however something more involved is required to ensure data integrity. True sharing requires the processors involved to explicitly synchronize with each other to ensure program correctness**[3].**

The second aspect of the memory consist deals with validating that the last store is read back. This involves data checking which is very difficult to predict if multiple coherent masters are thrashing the same memory location concurrently. This part of the problem is solved by using false sharing amongst all the coherent masters in the system. Each coherent master owns a byte of the same cacheline and uses a read modify pattern to increment its own byte. Thus observation of an incrementing pattern for each byte wherein the coherent master not only checks the incrementing pattern for its own byte but all bytes in the cache line. This partial line RMW (read modify write) and an observation of incrementing pattern for each byte ensures the data integrity and a coherent master can employ a self-checking mechanism. Hence, enabling portability of test scenarios beyond unit level verification environment. The diagram below illustrates the false sharing mechanism. Here M n is a cached master and Cp is a coherent non-cached I/O master, typical of modern day heterogeneous processing systems and in our case is a mixed ARM and TI DSM based system and other coherent I/O's talking to memory shared memory controller.

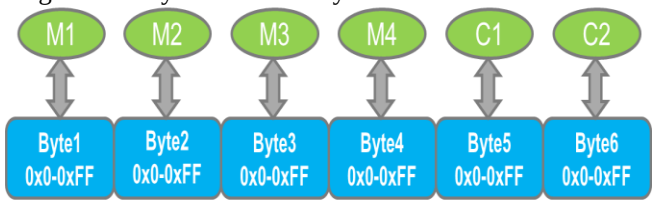

Fig. 9. False sharing

VIII. Getting it Done with Specman/e

Use of specman/e allowed us to have clean partitioning between synthesizable and non-synthesizable views of the testbench environment and decoupled the compile flows based on the view. Layer 2 and layer 3 overlay on layer 1, and these can be pulled in by inclusion of one file during compile for the required view. For alternate view layer 1 is sufficient enough for complete simulation runs.

Specman/e compiler works in compile as well interpreter mode and permits addition of new functions or modifications to existing structures at simulation time without having to recompile the entire testbench environment. This was very helpful in initial stages of development for debugging environment code on-the-fly during simulation.

Certain sections of environment code such as coverage bins, checkers had to reused with slight modifications for different structures of design and 'defined as' macro feature of specman/e was very helpful in generating such repetitive code with required parameters. An example for replicating coverage bin with an equivalent parameterized macro in specman/e is shown below.

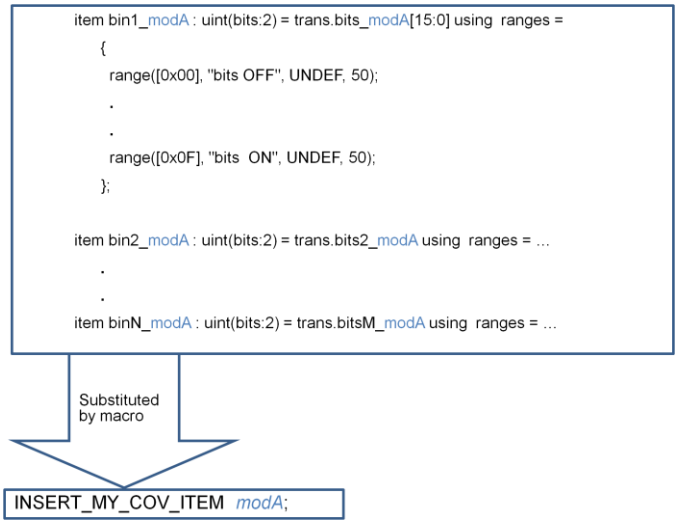

Fig. 10. Macro usage

IX. Summary

1) 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		

2) We created a unified DV environment that supported quad-core, dual-core, and single-core ARM SS.
3) Reuse Metrics

Components	IP to SoC
Test cases	~50%
Functional coverage bins	~4K
Checkers	~300
Software Library Functions	100%

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

X. Conclusion

A layered hierarchical testbench environment implemented with combination of HDL and Specman/e components allowed us to create a scalable verification environment for ARM Cortex A15 processor based subsystem at IP level. Moreover, it also allowed us to achieve coverage goals across various design configurations and variants at the IP level with maximal reuse. It also facilitated reuse of verification assets and software library for SoC validation. Furthermore, the layered approach also enabled us to port the synthesizable components of the testbench to the FPGA and portability of tests, from subsystem all the way onto evaluation boards for further testing and characterization. Novelty resides in the way verification environment is structured to encompass all the above features in single environment.

References

[1] Multicore ARM SoCs Face Cache Coherency Dilemma : Featured Techtalk, Mirit Fromovich, Cadence.

[2] Adve, S. V. and K. Gharachorloo, " Shared memory consistency models: A tutorial," IEEE Computer, December 1996, pp. 66-76.

[3] http://people.csail.mit.edu/saman/papers/anderson95/node2.html