2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA
Sept 25-26, 2014
Hotel Park Plaza
Bangalore, India

UVM , VMM and Native SV: Enabling Full Random Verification at System Level

Ashok Chandran, Project Manager, Analog Devices, Bangalore, India (*ashok.chandran@analog.com*)

***Abstract*— This paper describes an approach which enables full random verification at system level. This approach is methodology agnostic and can use blocks from UVM, VMM and SV. Paper also describes the general capabilities and advantages of various approaches used**

Keywords—System Level, SoC, Random, UVM, VMM, System Verilog

I. INTRODUCTION (STYLE: HEADING 1)

The Blackfin SoC Family caters to a very wide range of markets. While some are targeted applications, we also cater to a wide variety of general purpose customers. The variety of use cases that need to be validated to meet such a wide market is very huge. We believe that having full random verification capability at system level has helped us achieve robust quality for the past many silicon – reaching customer sampling with first silicon every time. The paper describes the verification approach used on BF70x – the latest Blackfin based processor SoC.

Our team had built up a rich set of legacy verification IPs in Verilog and VMM over the past projects. With the recent adoption of UVM and better practices, there was a strong desire to migrate to UVM. The interesting question was how to ensure that the existing verification IP (some being very complex) can be reused within an UVM framework and still work with its full random capabilities and controllability. We also (like others) had an interesting problem on how to reuse the stimulus or the sequences from a block level UVM testbench efficiently at system level under the resource limitations. The paper describes an efficient architecture which merges all the methodologies and provides a full random capability for SoC level verification.

The broad charter while developing the testbench were:

- Enable full reuse of all block level testbenches – VMM/UVM/Verilog
- Full randomization capability
- System Resource Management
- Interconnect and SoC Performance Validation
- Ease of Testbench Integration
- Reuse across projects

II. TESTBENCH ARCHITECTURE

A. Core BFM Approach

The testbench drives on core to system Interface Bridge via protocol drivers/ BFM. However, the core is not completely bypassed. There is an arbitration scheme between the core and the BFM. To verify the core-system interaction, the testbench also randomizes the instructions inside the core to achieve various scenarios that affect the system. This include cache fills, memory access, exceptions, code execution etc. Memory mapped register access is abstracted to a higher level by using UVM register package.

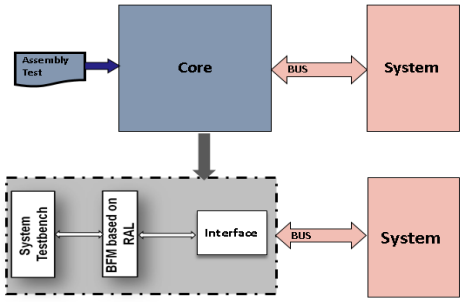

Figure 1- Core BFM Approach

B. Automated Testbench Build

To enable ease of reuse of platform across projects, it was planned that as much as possible, the data required for testbench build should directly come from the spec. The spec maintained in xml was easy to query and generate config db / other files required to direct testbench build. The Verilog side is auto stitched based on perl directed rule file, chip pinmux spec and templates for each peripheral. The template is instantiated for each instance as a separate module and with different hierarchical taps.

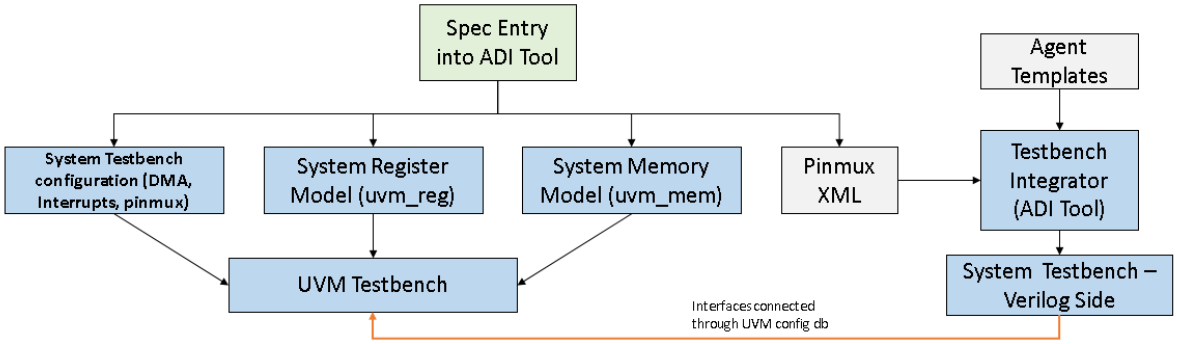

Figure 2- Automated Testbench Build

C. System Resource Mapping

Across projects, the standard infrastructure – like DMA, Pinmux Scheme, and Interrupts etc. remains same. However, the mapping of blocks to each of these keeps changing. As a result, we need a dynamic way of associating each block with the resource at UVM build time. This is achieved by a table maintained in UVM config db. The component hierarchy builds automatically by querying the config db.

To ensure that peripheral to dma channel/ interrupt is kept dynamic, a TLM router was added. On top of it a request ack protocol works to register masters (peripherals) and slaves (dma, interrupt etc). This way we can dynamically route the transaction from a master to the slave and send the response back. This approach also scales well for multi-core where you can map the resource to a different core as well.

D. Block to System Component Reuse

Reuse of UVM components in a toplevel testbench is somewhat straightforward in normal case. However, the complexity here is in terms of providing the resources (like DMA, Interrupt monitoring). To provide such support, an intermediate system level UVM component level is added. This level instantiates the UVM block level testbench, adds system level driver, interrupt monitor etc. There is also a virtual sequencer which connects to block level virtual/normal sequencer and to the system level driver.

VMM subenv/xactors are instantiated inside a UVM wrapper component. The UVM wrapper component provides system level hooks which allow the VMM testbench to request for resources. The phase objection of UVM is made to wait for the consensus from VMM. Also, the transactor / subenv start is done inside the run phase of the system level wrapper component. This way the VMM phasing is aligned to UVM.

Legacy system Verilog/Verilog agents used to read from files to generate stimulus. To incorporate these agents into the UVM environment and allow controllability; we map the file read memories as uvm mem models. They are then associated with a agent register model. Through backdoor poke() , we can now initialize and control the external Verilog agent. To do end of test data checks, we can peek() into these memories and check against expected data.

E. Block to System Stimulus Reuse

UVM methodology enables strong block to system reuse for monitors, scoreboards etc. However, stimulus porting to system level and ability to retain full randomness is somewhat limited. The virtual sequencer based layered approach is used to manage both block level and system level sequencer. To ease the process of integration of multiple sequences, a macro based approach is done.

VMM scenarios and multi-stream scenarios are reused via UVM based wrappers. The UVM sequences so created are executed on a sequencer. The sequencer provides pointers to the Multi-Stream generators.

F. Resource Allocation and Randomization

In an SoC, there are several resource limitations – like some pins are shared, memories limited etc. Hence, to ensure that we hit all combinations we need to generate random sequences or orders. It is simple to generate random sequencer orders in UVM and random scenario election in VMM. When you deal with both together, it

needs ad-hoc methods to ensure randomness across UVM and VMM under a constraint which limits maximum number of peripherals active in a test. This is enabled by customized multi-peripheral scenario and generator scheme. This allows you to register UVM/VMM sequences and allow random election.

G. Layered Approach

The usefulness of layered approach was described in detail for a VMM based approach. See ee-times article for detailed view on this approach. This is continued in UVM as well, with extensive usage of UVM register and memory packages.

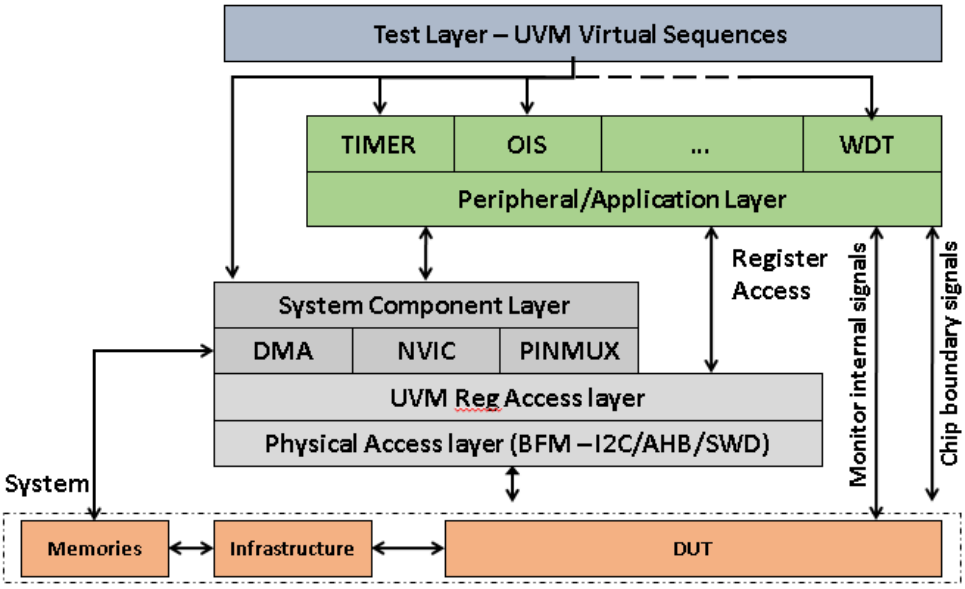

Figure 3-Layered Approach

III. SUMMARY

The above verification approaches resulted in robust silicon quality which enabled sampling to customers as per planned schedule. This platform is being reused across multiple product families within ADI.

ACKNOWLEDGMENT

Author would like to thank the Analog Devices India - Blackfin team for the contributions towards this platform.