Reusable On-Chip System Level Verification for Simulation, Emulation and Silicon

Avishay Maman, Sharon Goldschlager, Hillel Miller, David Bell, Rob Slater, Oded Ben-Moshe, Nissan Levi, Hagit Gilboa

23 Oct 2006

Freescale Semiconductor Israel
Agenda

► ROC methodology
► ROC specific implementation on the MSC8144
► Coverage and bugs found
► Comparison with other approaches
ROC Methodology for System Level Verification

System Level Verification
► Validate the integration of independently-verified components.
► Simultaneous activation of peripherals.
► Overall functionality.

ROC methodology
► Light-weight operating system written in C.
► User defined tasks.
► Exercising several HW blocks.
ROC Components

- Memory
- Resources
- OS Scheduler
- Memory Manager
- Resource Manager
- Task Launcher
- Interrupt Handler
- Random Library

System
User
External
ROC Components - Task

- A software module that verifies the functionality of a specific HW block.
- A task consists of fixed logical execution phases.
- Scheduler interleaves tasks’ execution phases.

For a DMA block:

Init
- Checking conditions for startup.

Run
- Resource allocation: source, destination memories, DMA channel interrupt.
- Initialization: Data buffers, DMA block registers.

Iterate
- Waiting for finish signal by an interrupt.

Check
- Checking operation results validity - data comparison of source and destination buffers, report status and finish.
Different Verification Environments

Standalone Debugging

Chip Level

Emulation

Silicon

PLI interface
ROC Implementation

**MSC8144 block diagram**

Chip-level integration and connectivity, system-level verification
Out of Reset Flow

- Reset is asserted at the beginning or during simulation
- Generate seeds per core
- Randomize clock ratios between the 7 clock domains
- Add random delay to all of the synchronizers
- Randomize I/O pins configuration and relevant tasks
- Bootstrap - initialize the chip’s configuration registers
- Initialize the ROC OS by each core
- Initiating Scheduler, calling task launching routine
Coverage

1. Connectivity Coverage
   - Data bus connectivity, pipeline depth, access alignment.

2. System Level Coverage for the CLASS interconnect switch.
   - Read and write accesses from each initiator to each target.
   - Combinations of simultaneous accesses to different targets.
   - Committed accesses per target per time interval.

3. ROC task's random parameters coverage (emulation and silicon)
   - Data rate measure.
   - Cross of peripheral, source and destination memory segments.
   - Cover hardware blocks configurations.
Bugs Found

Different types of bugs were found.

System Level configuration bugs
  ► Arbitration related (PCI).
  ► Wrap accesses (M3).
  ► Architectural (PCI).

Corner-case (exhaustive)
  ► Recovery from reset (DDR).
  ► Access alignment (DMA).

Block level (unexpected)
  ► Module internal (DMA).
Comparison With Other Approaches

Verification Proprietary Tools
+ Have an on-the-fly constraint based random generation engine and coverage tracking mechanism.
+ Have a well defined reuse methodology, but only for simulation.
- No reuse of testbench in emulation and post-silicon.
- No leverage of the SoC's C based software drivers.

Pre-generation
+ Can be potentially run on emulation and post-silicon.
- Inherently no on-the-fly biasing to increase coverage.
- No reuse of C based SW drivers on all environments.
ROC Methodology -’s

- No 100% coverage guaranteed – not a formal method.
- Core stub – inaccurate timing.
- Porting to emulation requires a set up effort.
- Task visibility is limited to core accessed components.
ROC Methodology +’s

+ Reuse of code over project life-cycle - verification environments.
+ On the fly biasing supporting coverage driven approach.
   ROC OS and ROC tasks are written in C which inherently provides
   the following advantages:
+ No need for a new specification or modeling language.
+ Reuse of code over different SoC's, and cores.
+ Seamless activation of C based software drivers (to be used by
  customers).
+ Code in simulation can be linked with other tools having a C API
  and not be dependent on a specific simulator or environment.
+ Use of existing tools for the development and debugging of C
  programs.
+ Use of C compiler optimizations, otherwise very time consuming.
Freescale MSC8144 – World’s Most Powerful DSP

Launched on May 2006
Thank you!

Bon Appetit!

Sharon@freescale.com