Low Power Validation on Emulation Using Portable Stimulus Standard

Joydeep Maitra, SRR3, (joydeep.maitra@intel.com)
Deepinder Singh Mohoora, SRR3, (deepinder.singh.mohoora@intel.com)
Vikash Kumar Singh, SRR3, (vikash.kumar.singh@intel.com)

Abstract

Today's modem SoCs have heterogeneous processor architectures containing several clusters with multiple cores per cluster. The power logic is implemented across numerous power domains involving interaction between different clusters and distributed memories – all communicating through a network of control signals and a complex NOC architecture. To achieve high quality silicon, it is imperative to stress the architecture rigorously yet predictably at the pre-silicon stage of the project.

Our innovation lies in creating a scalable test generation framework which allows us to create complex multithreaded low power test scenarios and their corresponding checking methods on emulation simultaneously using Portable Stimulus Standard (PSS) **[1]** – a powerful language for capturing system level test intent. The level of test complexity we have achieved was not attainable by manual test development that we have done in earlier projects. Also without the automated checking mechanism, it would be practically impossible to sign-off on the test execution. This has led to significant savings in time and effort for test generation and debug.

Keywords— Low Power Validation, Emulation, Portable Stimulus Standard, Simics

1 Introduction

Modern low power architectures usually have a distributed implementation to support sophisticated power saving schemes across heterogeneous processor clusters to meet challenging power KPIs that are expected by our customers. For instance, our paper refers to a SoC where there are multiple ARC® core clusters, custom 3rd party core clusters and multiple Tensilica® cores each with its own low power architecture – integrated with the overall system low power scheme.

Pre-silicon validation of such an architecture involves developing system level test scenarios in bare-metal OS-less software (which can be executed on an RTL accurate platform) where multiple SoC components are constantly shuffling in and out of their respective power states simultaneously. Functional bugs and design weaknesses that manifest during such entry-exit sequences alone have contributed to the majority of low power issues in multiple previous generation SoCs. Thus a robust testing technique that stresses the complex power architecture scheme to its corners is crucial for detecting both power related HW and SW/FW issues in the pre-silicon phase of the project.

Our methodology combines the capability of multi-threaded bare-metal test generation of PSS, RTL accuracy and debug ability of the Emulation HW and the capabilities of Simics **[2]** emulation testbench into an automated environment to create highly randomized, stressful, self-checking SoC low power test scenarios.

2 Stressing low power architecture using PSS on Emulation

2.1 Problem context

Let us consider a simplified architecture for a typical modem SoC as depicted in Figure 1. It has four CPU clusters – ARC® (4 cores), A custom 3rd party core cluster (4 cores referred to as CCA-Cx in the figure) and two Tensilica® clusters with one core each. Each core and cluster has its own low power states. The overall SoC has a DeepSleep state where various power domains are switched off and clocks are gated or switched to slower clocks for reducing power consumption. Exit from the low power states can be triggered by wakeup events such as timer expiry or interrupts from various active sources. All external memory accesses happen through the NOC which has multiple power switches too for optimizing power consumption.

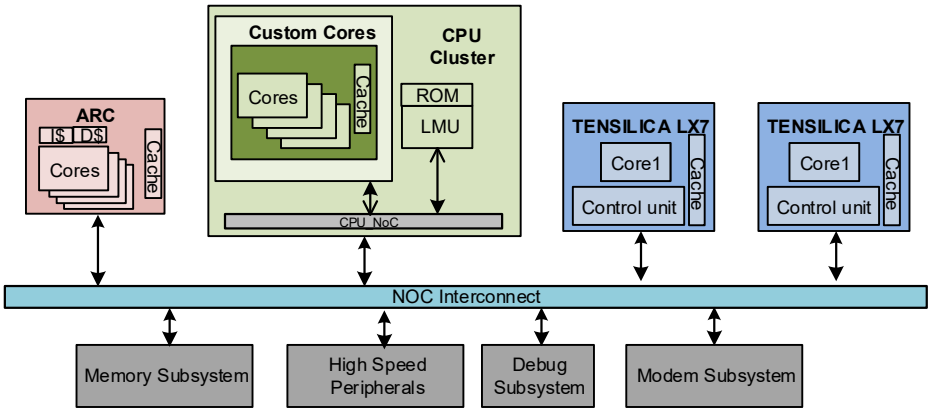

Figure 1: Simplified SoC Architecture Block Diagram

For brevity of explanation, let us limit the possible power states as follows:

Cluster Name	Core Power States	Cluster Power States
Custom Cores (CCA)	WFI, Shutdown	PS1, PS2, PS3, OFF
ARC®	WFI, Shutdown	ON , OFF
Tensilica®	WAIT, Dynamic Retention, Shutdown	ON, OFF
System Components	**Power States**	
NOC	LOW, MED, HIGH	
SOC TOP	ON, DEEPSLEEP	

Table 1: Power states in the system

Our primary goal here is to ensure exhaustive coverage of all low power states of individual components and stressing the overall SOC DeepSleep. The test should exercise all legal combinations of core, cluster and NOC power states while functional transactions are ongoing in parallel threads. Secondly, we should ascertain that all such state transitions strictly respect the latency requirements laid down in the architecture specification.

Figure 2 describes scenarios that can run functional threads (e.g. memory & peripheral operations) on all cores and corresponding power threads for low power states (of individual cores as well as the system DeepSleep) while randomizing system parameters like operating frequency, wakeup sources etc. The test should execute for several loops where in certain system parameters like memory addresses, wakeup options will get randomized at runtime. Since the aim is to test the hardware, this multithreaded code has to be lightweight and OS-free to ensure no software errors are inadvertently introduced in the code. Also such minimum overhead test code allows fast execution and easier debugging on emulation platforms.

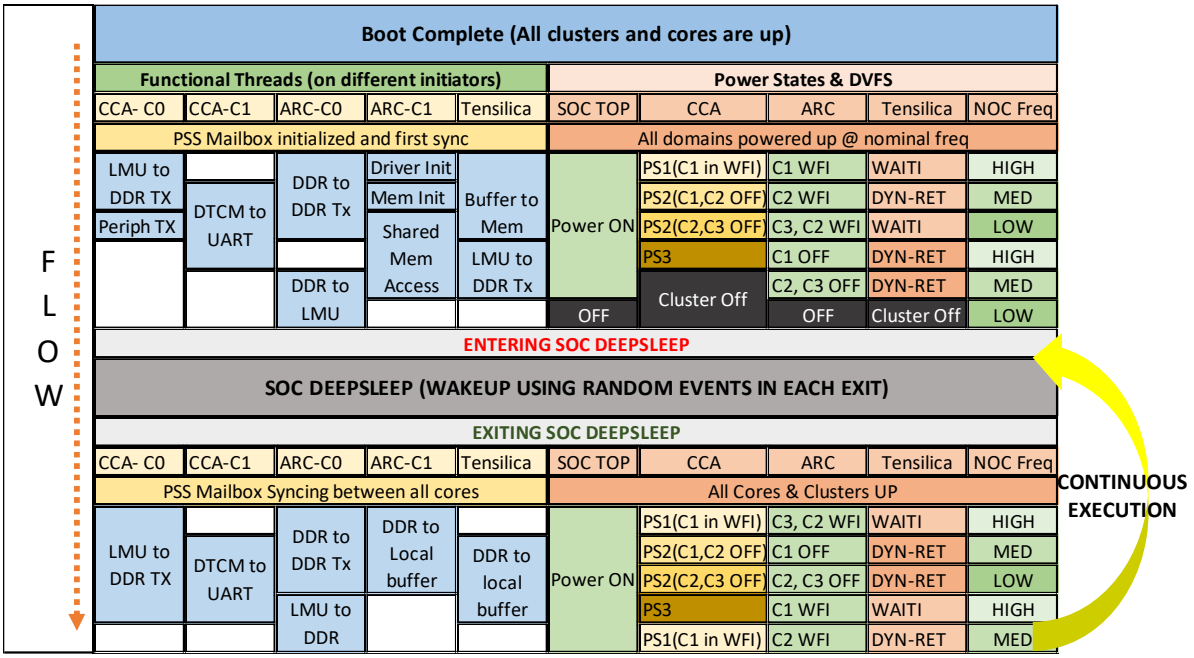

Figure 2: Multiple functional threads on different clusters along with randomized low power events

One obvious challenge is scheduling multiple threads and syncing them with multiple power threads (while respecting the architectural constraints) without using an OS scheduler. How we solve this by using PSS is described in section 2.2.

The functional operations (see blue boxes in above figure) can be checked in DUT code itself – say a memory write checked with a corresponding read action. However, the way to check whether actual low power events have occurred will require taking a design dump and checking the waveforms. A complex system flow like above can execute for long duration on emulation. It would require prohibitively large amount of memory and time if we were to capture the scenario on a waveform. Not to mention that analyzing the waveforms for the possible number of test scenario combinations manually would be practically impossible.

To automate this checking, we shall treat Simics testbench as an active system component and create actions for it which are closely coupled with the generated test flow. These scheduled actions will allow the Simics testbench to control the test flow, check for occurrence of relevant HW events and provide runtime stimulus.

Another problem area lies in cases where latencies and signal transition sequences get violated by a very thin margin and only when certain set of threads are concurrent in the system. The need for reproducibility of the failure case with ample visibility into the design becomes paramount.

The next section covers how we use PSS to model the DUT, schedule actions on the Simics emulation testbench and generate the requisite C and Simics test scripts.

2.2 Using PSS for test generation and test-bench automation

PSS provides a declarative environment to model system level components with their behavioral descriptions and compose test cases with data and control flows. The primary behavior abstraction mechanism in PSS is an "action", which represents a particular behavior or set of behaviors. Actions combine to form the scenario(s) that represent(s) the verification intent. If the system level constraints are correctly captured during modelling, the constraint solver in a PSS composer can produce a range of legal test scenarios for different execution platforms. Moreover if the verification intent is incomplete (partial), the PSS tool can infer the additional actions and model elements to complete the test scenario. This leads to greater randomization of test cases which otherwise would have been missed in manual generation.
Finally in the test generation phase, the PSS tool picks up the relevant code templates (exec blocks) corresponding to each action and creates the necessary C test code and testbench script as desired.

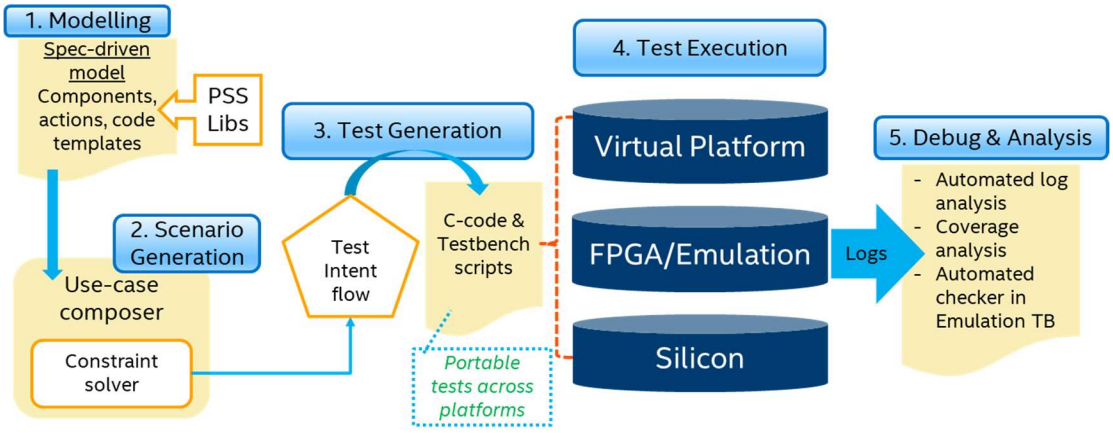

Figure 3: PSS usage flow

Let us explain the approach with an example of a simple power down sequence involving only the CCA core cluster containing 4 cores.
Say we want to create a test where after completion of boot, each core either enters a "WFI" or "Shutdown" state in parallel. The atomic actions we can create are "enter_wfi" and "enter_shutdown". We allow the scenario solver to randomly assign these actions to the four cores and generate a parallel test flow (Core 3 and Core 1 enters WFI, Core 2 and Core 0 enter Shutdown). Figure 4 shows the generated test intent flow.

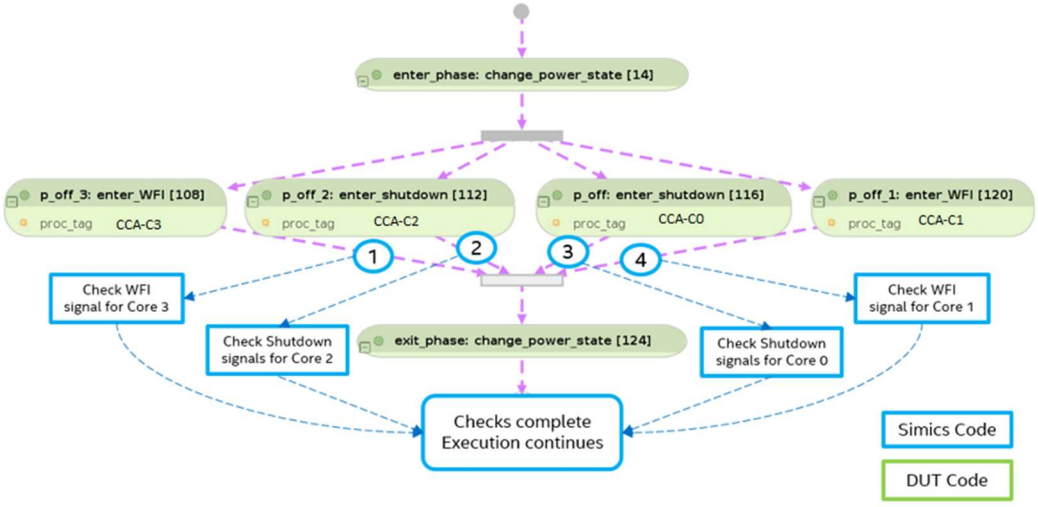

Figure 4: Example of Multiple Core Low Power Entry with corresponding Checker flow in Simics

For each DUT execution thread (marked 1,2,3,4 in the figure) there exists a "checker" action which executes on the Simics testbench. These checker actions will monitor and compare relevant signals in the design against expected values and ratify the test execution. Only when the checks are complete, the rest of the test will execute. This provides a foolproof method to carry out run-time checks for both SW & HW on emulation without having to depend on text logs or sifting through huge waveform dumps.

The checker code on a single Simics thread looks like this:

```
//Define a breakpoint on one core's WFI signal
Local $breakpoint=
(emu.break "emulation_tb/inst_soc/path_to_core_CCA-C1/WFI_signal =1")

//The design continues to execute till the breakpoint is hit
 wait-for-breakpoint $breakpoint
 echo "Core CCA-C1 WFI state reached"
```

So when Core 3 executes the WFI instruction and the WFI signal in the design is asserted, the above checker will hit the breakpoint and halt the execution. The rest of the checker threads will execute in parallel on Simics testbench using script branches:

```
script-branch
{Checker for CCA-C3 WFI state reached}

script-branch
{Checker for CCA-C0 SHUTDOWN state reached}

script-branch
{Checker for CCA-C2 SHUTDOWN state reached}
```

When all the four low power events are ratified, the design runs again and test flow continues.

There are scenarios other than the above power down sequence where the same principles are extended.

External Stimulus: In this case, the Simics emulation testbench becomes an active enabler of the test flow and not just a passive listener on system events. We generate external stimulus like wakeup events or asynchronous resets through the testbench.

Latency Measurements: To ensure that all the transitions into and out of different power states are adhering to the latency specifications, we use the testbench to capture the timestamp of relevant events depending on the test scenario and calculate the pure HW latency.

Fail case analysis: Another key capability in the framework is a "Fail-case Automatic capture" utility – if a scenario where a desired checker fails in the testbench, it can enable deeper debug visibility by capturing a waveform dump for only the relevant time window automatically. This has been invaluable in localizing issues in regression tests.

3 Results

With our methodology, we are able to create test cases and corresponding checker mechanisms for highly complex scenarios at the pre-silicon component verification stage itself (example in the Figure 5), which involves a high number of parallel threads running on both, the DUT & emulation testbench.

Though a significant amount of effort was needed to develop this test framework, but this one time effort led to a pull-in of the test development activity by **four man-months** and got almost 100% re-use of the infrastructure for the derivative projects.

<table>
<tr><td colspan="5">Executing on DUT</td><td colspan="4">Executing on Simics Testbench</td></tr>
<tr><td colspan="5">All domains powered up @ nominal freq</td><td colspan="4">Simics Threads</td></tr>
<tr><td>SOC TOP</td><td>CCA</td><td>ARC</td><td>Tensilica</td><td>NOC Freq</td><td>for CCA</td><td>for ARC</td><td>for Tensilica</td><td>for NOC</td></tr>
<tr><td rowspan="5">Power ON</td><td>PS1(C1 in WFI)</td><td>C1 WFI</td><td>WAITI</td><td>HIGH</td><td>Check PS1
Trigger Wakeup</td><td>Check C1 WFI</td><td>Check WAITI</td><td>Check HIGH Freq</td></tr>
<tr><td>PS2(C1,C2 OFF)</td><td>C2 WFI</td><td>DYN-RET</td><td>MED</td><td>Check PS2</td><td>Check C2 WFI</td><td>Check DYN-RET</td><td>Check MED Freq</td></tr>
<tr><td>PS2(C2,C3 OFF)</td><td>C3, C2 WFI</td><td>WAITI</td><td>LOW</td><td>Check PS2</td><td>Check C2,C3 WFI</td><td>Check WAITI
Trigger Exit</td><td>Check LOW Freq</td></tr>
<tr><td>PS3</td><td>C1 OFF</td><td>DYN-RET</td><td>HIGH</td><td>Check PS3</td><td>Check C1 Off</td><td>Check DYN-RET</td><td>Check HIGH Freq</td></tr>
<tr><td rowspan="2">Cluster Off</td><td>C2, C3 OFF</td><td>DYN-RET</td><td>MED</td><td rowspan="2">Check Cluster Off</td><td>Check C2,C3 OFF</td><td>Check DYN-RET</td><td>Check MED Freq</td></tr>
<tr><td>OFF</td><td>Cluster Off</td><td>LOW</td><td>Check Cluster Off</td><td>Check Cluster Off</td><td>Check LOW Freq</td></tr>
<tr><td colspan="5">Entering DeepSleep</td><td colspan="4">Check Status of Sleep Blockers</td></tr>
<tr><td colspan="5">SOC DEEP SLEEP ENTERED</td><td colspan="4">Check DeepSleep relevant signals</td></tr>
<tr><td colspan="5">Exiting DeepSleep</td><td colspan="4">Trigger Wakeup Event on External GPIO</td></tr>
<tr><td colspan="5">All Cores & Clusters UP</td><td colspan="4">Check Power State of all cores</td></tr>
</table>

Figure 5: An example of Power threads on DUT and Simics Testbench actions

Though we could complete and deploy this framework only very close to the tape-in of the SoC, we found a few critical SoC level design bugs in the first few runs itself. We describe one such discovery below which had to be fixed by an ECO before tape-in.

In this scenario a Tensilica® core tries to access the LMU in the CCA cluster when the latter is off (refer Figure 6). The design requires this to trigger a wakeup for the cluster and enable the access to the memory. No

issue was seen in SoC simulation level test flows. However when our test framework randomly changed the NOC clock down to reference clock frequency while it had scheduled the access, the test failed.

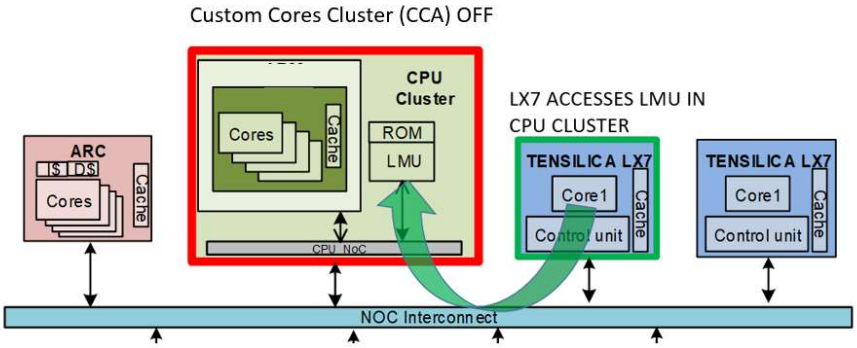

Figure 6: LX7 accessing LMU in Custom Cores Cluster with NOC @ Reference Clock

This frequently used flow would have rendered many top level software functions unusable. Without a multithreaded test that also randomizes system parameters like NOC frequency, we were unable to hit such corner cases before this.

Hitting these kind of issues in a test scenario requires a high level of parallelism and randomization in the flow which is often impractical to achieve in a simulation level or single threaded test. In fact, the lack of a multithreaded modular framework had made it very difficult to manually develop such tests for pre or post-silicon validation in both the current and parent projects. As a result, these kind of issues got reported much later either by system software teams or worse still – by a customer.

4 Summary

The test framework provides hooks and levers to increase test complexity, randomness and iterations out-of-the-box. This will allow the existing test scenarios to be scaled and executed as-is on silicon without the need of any additional development. In other words, the time and effort needed to revisit pre-silicon test code and refactor it for post-silicon will tend to be zero.

Though we have started with applying the methodology to the area of low power validation, its scope can be extended to other areas as well. Since the test intents are written at a high level of abstraction, the same framework can be leveraged across different SoC programs in the industry.

.

References

[1] "Portable Test and Stimulus Standard 1.0," [Online]. Available: https://accellera.org/images/downloads/standards/pss/Portable_Test_Stimulus_Standard_v1.0.pdf.

[2] "Wind River Simics ® Release Note Version 5," [Online]. Available: http://simics-download.intel.com/simics-5/docs/RELEASENOTES.pdf