# Test-driving PSS for System Low-Power Validation

Prabhat Gupta
Prabhat.Gupta@amd.com

Matan Vax
Matan@cadence.com

***Abstract*-This paper describes the application of the new Portable Stimulus Standard (PSS) to post-silicon validation, exercising power behaviors of the system, hardware and firmware, in different usage profiles. The solution pattern we describe is simple but quite general. We share lessons learned with respect to the PSS language and its expressive power for specifying tests in this domain. We also have encouraging proof points concerning the level of automation and reuse that can be achieved by a tool processing these descriptions. We believe our experience with PSS contains valuable observations for validation teams and the standard committee alike.**

## I. Introduction

Power is important in many SoC designs. Very complex logic, both in hardware and firmware, is put in place with the objective of reducing power consumption in a variety of usage profiles of the system. It is important to functionally validate this logic, as well as analyze the actual power consumption, of these use cases at full-chip level. Such analysis could uncover “power-bugs” in hardware or firmware and suggest tuning of firmware to optimize power usage. Validation process should ideally start with virtual prototype, and subsequently continue with the real design in both pre-silicon and post-silicon environments.

Unfortunately, coding up tests for low power scenarios to run on multi-core, multi-cluster processors, in a synthetic embedded execution environment, is not an easy task. Even worse is maintaining many different test variants in the face of ever changing requirements and a multitude of execution environments. AMD has been looking into ways to improve the quality of tests in this domain, and to increase productivity in their development. Moreover, AMD has the long-time objective to reduce the cost of implementing such tests for different platforms and environments.

In 2015, Accellera has formed a working group to define a standard language for capturing test intent that is portable across different platforms and environments – the Portable Stimulus Working Group (PSWG). Much of its focus was devoted to the problem of describing system-level use-cases. Low-power verification/validation was one of the recurring themes in both design objectives and usage examples brought in front of the group. In July 2018, the official standard – PSS1.0 – was announced [1]. AMD is one of the first companies to adopt the new standard and apply it to real problems. This paper describes the early experience with PSS in this application domain using the Cadence® Perspec™ System Verifier tool.

There are multiple public reports on the use of “Portable Stimulus” in real application context. To the best of our knowledge, few are strictly concerned with the Accellera PSS (the term ‘portable stimulus’ is used liberally to refer to different solutions, feeding some confusion in the industry). Even fewer publications show and discuss concrete PSS code. We are not aware of such a report covering our field of application. The contribution of this paper is in putting forward a practical solution pattern for a common validation problem in terms of the new PSS standard, including concrete code snippets and explanations. As a corollary, we share lessons learned about PSS and directions for future enhancements.

## II. Problem Statement

The following outlines a fictional high-level architecture to use as an example. A system, or a “package”, consists of one or more CPU-complexes and a GPU (and other components not discussed here). In a CPU-complex there are four cores, and each core executes two HW threads. The power states are hierarchically organized along these same lines. For this example, we are defining that for a core to begin its transition into low-power state, execution on both its threads needs to be halted. Initially it transitions to state Core_C1, and subsequently, to the deeper states like Core_Cn. In general, each transition starts after the expiration of a timer and has some latency. Power collapse on

CPU-complex starts after all cores are in Core_Cn, and similarly full package power collapse starts after both CPU and GPU are down. The diagram in Figure 1 shows our example architectures different states and their dependencies.

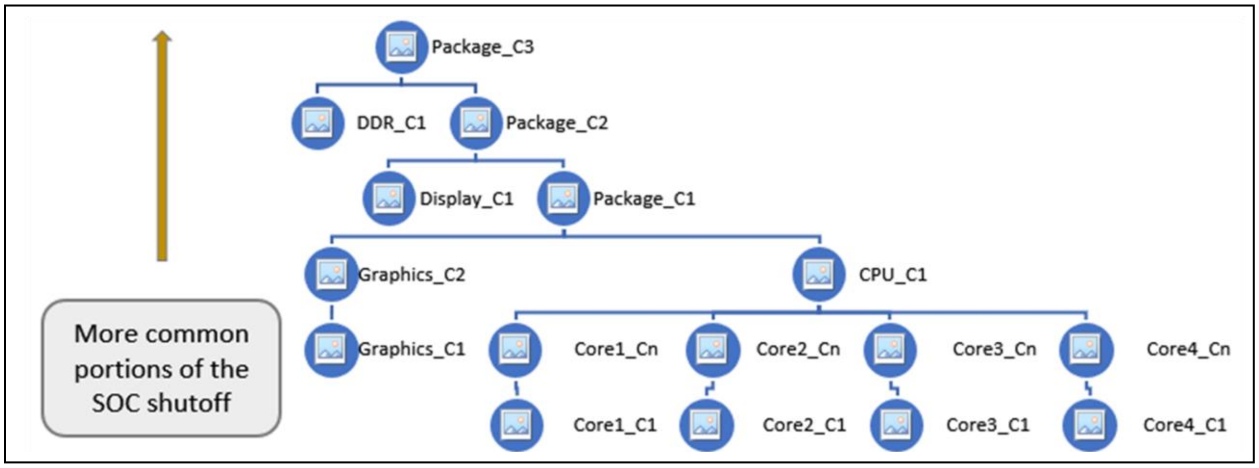

**Figure 1. System power-states**

Several challenges can be pointed out with respect to capturing the space of scenarios and automating scenario test-code generation. For example, in some of the flows, timer-based interrupt should be set before cores execution is halted. Values should be such that the timer expires at the right point relative to the power-sequence stage. For some purposes, steady power-state should be maintained long enough to enable measurement. For other purposes wakeup should come in at different points during the transition, or closely after low-power state was reached, to enable analysis of power behavior and trade-offs.

Separate challenges have to do with the underlying implementation of test scenarios. One example is guaranteeing thread-safe synchronization for the multiple test activities across the different hardware threads and cores. This problem is aggravated in our case by the fact that the very same threads are suspended and resumed as part of the overall flow, and yet implementation of dependencies must be deadlock-free.

## III. Solution Overview

The PSS language includes native constructs to fork behaviors in concurrent branches and capture control-flow and dependencies between operations performed by the test and the underlying system. Such units of behavior are called 'actions'. Higher-level actions call lower-level actions in some specified control/data flow called 'activity'. Atomic actions – the ultimate leaves of the behavioral hierarchy – are directly mapped to test code in the target language. This description of behavior is declarative, and algebraic constraints can be applied to elements of the behavior across different levels of the hierarchy.

### *A. Modeling Preliminaries*

In our post-silicon validation environment test code is executed by CPU cores within the system. We assume no operating-system kernel or higher software layers in the loop. Generated tests use "bare-metal" code mostly in C, with some assembly code. The code is cross-compiled, loaded onto the system memory, and executed on the silicon board in the lab.

Being able to control the operation of specific cores is an essential part of the task. For our purposes it's convenient to represent this association of actions with threads using just two numeric attributes – 'core_num' and 'thread_num'. Ultimately these attributes jointly correspond to a unique thread id, such that the target code mapping of atomic actions would show up under the respective entry-point (thread "main" function). In order not to duplicate these declarations, we factor them out to an action base type.

To initiate core power collapse, we must suspend instruction execution on both its hardware threads. There are different ways in which threads can be suspended, but for our present purposes it's enough to generate a halt instruction (HLT). We encapsulate this with an action called 'halt'. Before each thread is suspended, we need to set up a timer to send an interrupt and thus wake up the thread after a certain period expires. This is performed by action 'set_timer_interrupt'. Here too there are alternative ways to wake up a thread/core, but for simplicity of this example we focus on timer-based interrupt. Prior to suspending execution, we need to enable a certain power state for that core. This is the purpose of action ‘enable_cstate’. See Figure 2 for the PSS code with these definitions.

## *B. Scenario Building Blocks*

We move on to define the flow for powering-down a core in terms of the basic operations defined in the previous section. Again, the flow is the following: enable the goal state for the selected core (on any one of its two threads), and then for both threads concurrently, set up the timer-based interrupt and halt the execution. The code for compound action ‘power_down_core’ with its activity is shown in Figure 4. The same activity is also nicely depicted in the diagram in Figure 3. Such graphics can be presented by the tool given the PSS description as input.

Power collapse is a flow implemented by the underlying hardware. The longer the cores (and other engines) remain idle, the deeper the sleep state they reach. Given that a certain power state is enabled for a core, the ultimate state that would be reached becomes a function of the time that elapses from the suspension of both its

```
package core_defs_pkg {
  abstract action core_op {
    rand int in [0..NUM_CORES-1] core_num;
    rand int in [0,1] thread_num;
  };
};

component power_ops_c {
  import core_defs_pkg::*;

  enum cstate_e {Core_C0,…,Core_Cn, CPU_C1};

  action enable_cstate : core_op {
    rand cstate_e in[Core_C1,…CPU_Cn] goal_cstate;
    rand int in [0..1000] Core_C1_delay;
    …
    rand int in [0..1000] CPU_Cn_delay;
  };

  action set_timer_interrupt : core_op {
    rand int in [1..1000000] wu_delay;
  };

  action  halt : core_op {
    exec body C = """
      asm("hlt");
    """;
  };
  ...
};
```

**Figure 2: Base level actions**

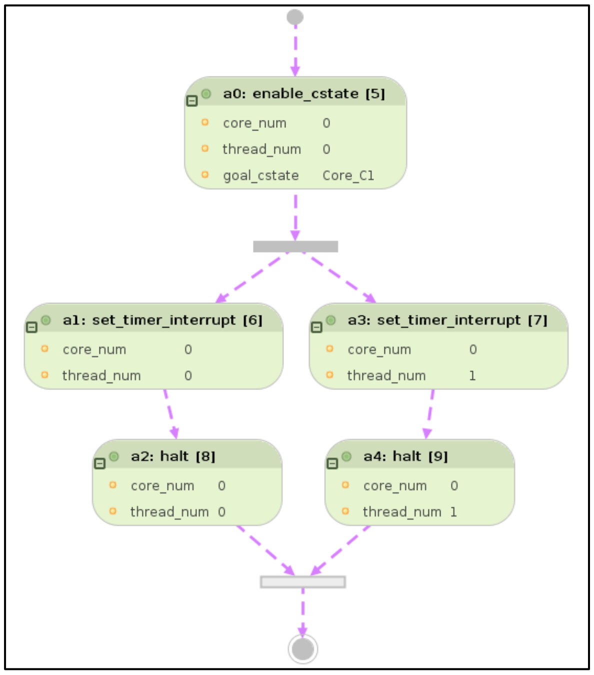

**Figure 3: Activity diagram for action *power_down_core***

```
action power_down_core {
  rand int in [0..NUM_CORES-1] core_num;
  rand cstate_e goal_cstate;
  rand int in [1..1000000] min_wu_delay;
  ...
  activity {
    do enable_cstate with {
      core_num == this.core_num;
      goal_cstate == this.goal_cstate;
    };
    parallel {
      thread0_seq: sequence {
        do set_timer_interrupt with {
          core_num == this.core_num;
          thread_num == 0;
          wu_delay >= this.min_wu_delay;
        };
        do halt with {
          core_num == this.core_num;
          thread_num == 0;
        };
      }
      thread1_seq: sequence {
        ... // similar to thread0_seq
      }
    …
};
```

**Figure 4: Code for action *power_down_core***

threads to the event that wakes it up. Thus, if we want to complete the transition and reach the goal state we need to make sure that the core is idle for enough time. In other words, we need to set up long enough wakeup delay. For this purpose, we declare a numeric attribute of action 'power_down_core' representing the minimum wake-up delay and constrain the timer setup actions accordingly.

The minimum wakeup delay corresponds to the transition we want to affect. The formula is straightforward. The transition from Core_C0 to Core_C1 would take the Core_C1 delay we configure at runtime, determining the time from idle state to beginning of the power collapse, plus the actual Core_C1 entry latency. Similarly, the transition from Core_C0 to Core_Cn takes the sum of configured delays and latencies for all core C-states. Figure 5 depicts the different phases during core power-collapse sequence, with different stages, beginning with the halt instruction, on to Core_C1 timer expiration, point-of-no-return, Core_C1 completion, and similarly for other Core C-states.

We intend to reuse action 'power_down_core' as a building block in the context of different use cases. In some cases we want the power transition to complete and the goal state to be reached, and in other cases we want to observe the behavior (and implications) of a power collapse sequence that is interrupted at different stages before it is completed, as indicated by red arrows in Figure 5. To support this requirement, we introduce a control-knob that represents the abstract intent and correlate it to the actual wake-up timer delays we choose.

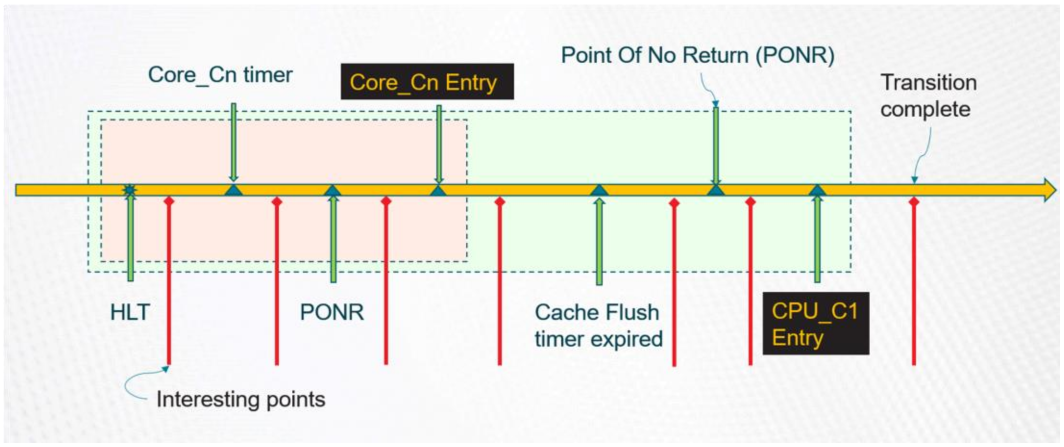

**Figure 5: Core C-state transition**

PSS features full algebraic constraint language, not very different from other hardware verification languages, such as SystemVerilog [2]. Formulas like ours can be directly expressed with constraints. The control-knob itself is declared as an enumerated attribute called 'transition_case', and its value correspond to the state of the transition we target. Figure 6 shows the code we use to do that.

```
action power_down_core {
  rand power_transition_case_e transition_case;
  rand int in [0..1000] Core_C1_delay;
  …
  rand int in [0..1000] Core_Cn_delay;
  rand int in [1..1000000] min_wu_delay;

  constraint transition_case == COMPLETE && goal_cstate == Core_C1 ->
    min_wu_delay == Core_C1_delay + Core_C1_ENTRY_LATENCY;

  constraint transition_case == COMPLETE && goal_cstate == Core_Cn ->
    min_wu_delay  ==  Core_C1_delay  +  Core_C1_ENTRY_LATENCY  +  …  +  Core_Cn_delay  +
Core_Cn_ENTRY_LATENCY;

  activity { ... }
};
```

**Figure 5: Attributes and constraints for *power_down_core***

## C. *Capturing a Use Case*

The process of power collapse and wake-up itself incurs significant power overhead. Transitions that are interrupted before they complete clearly have negative impact on the overall power consumption. However, even in cases where steady power state is reached, too short a time in the low-power state would not compensate for the overhead of the transitions to and from it. One of the first tasks for our PSS model is to help measure and chart power consumption behavior for one or more cores as a function of the duration of their idle time. For this purpose, we devise an action with an activity describing a top-level flow that would let measure in the lab the data required for this analysis. We do it in terms of action 'power_down_core' described in previous section.

The flow is as follows: power-down all cores except the one(s) subject to measurement, and then power that core(s) down, and back up, iteratively, with growing wakeup delays. For each wakeup delay value, loop many times, to enable reliable measurement statistics. The goal low-power state for these transitions is CPU_C1, but again, this state is actually reached only in phases that allow enough idle time. We call this flow a "sweep" scenario. We expose the sweep parameters as top-level attributes – the overall range in which we vary the delay, and the delay increments at each step. The code for action 'sweep_scenario' is shown in Figure 8. Note that in this code we use one construct which is currently not part of the PSS1.0 language reference manual – the 'replicate' statement. This is a topic for the discussion in the next section.

Figure 7 is a partial snip of the activity diagram of a specific instance of action 'sweep_scenario'. In it, core 1 is targeted and therefore core 0, 2, and 3 are powered down at the beginning. Then, core 1 is powered down in a loop, each phase with growing wake-up delay time (only the first two steps out of 10 are shown). Note that in the activity description there are two nested repeat statements. In

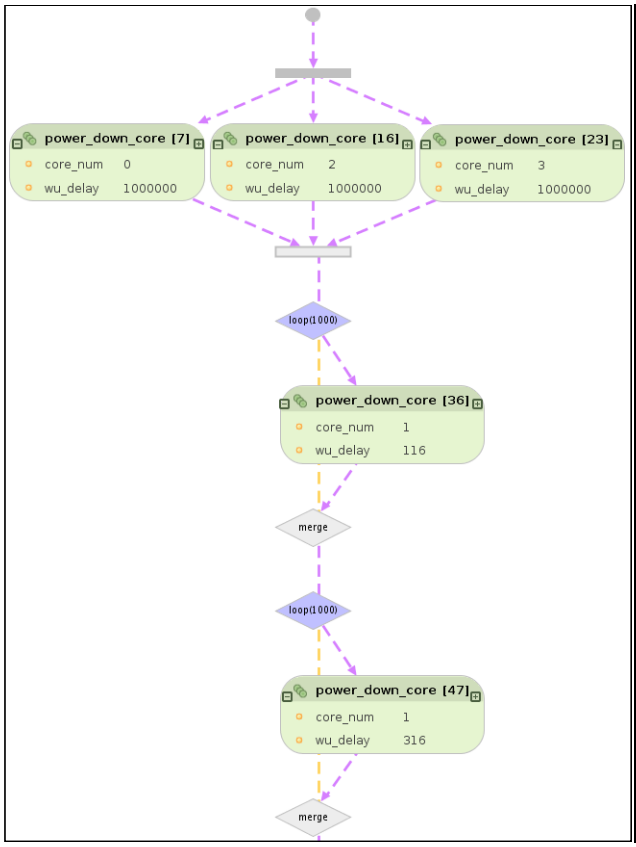

**Figure 7: Activity diagram for *sweep_scenario***

```
action sweep_scenario {
  rand int in [0..NUM_CORES-1] core_num;

  rand int in [1..10000] range;
  rand int in [1..1000] step;
  rand int in [1..1000] base_delay;

  activity {
    parallel {
      replicate (i: NUM_CORES) {
        if (i != this.core_num) {
          do power_down_core with {
            core_num == i;
          };
        }
      }
    }
    repeat (phase: range / step) {
      repeat (1000) {
        do power_down_core with {
          core_num == this.core_num;
          goal_cstate == CPU_C1;
          transition_case == SWEEP;
          wu_delay ==
            base_delay + phase * step;
        };
      }
    }
  };
};
```

**Figure 8: Code for action *sweep_scenario***

the diagram, the outer one is unrolled to show the incremental additions attribute ‘wu_delay’ while the inner loop with the high iteration count is kept rolled up.

## IV. Discussion: Language Limitations

We generally found the PSS language concepts very apt for the task at hand, and its specific constructs captured naturally our intent. However, we did run into a few limitations in expressive power which are perhaps not major, but still not easy to overcome. We mention here two such challenges.

Consider again the code in Figure 8 above. The scenario starts by powering down all cores that are not the target of the measurement for the entire duration of the run. Of course, we could power- down the cores one after the other, by iterating over all cores with a ‘repeat’ statement, which implies sequential execution. But there is no reason to do so sequentially and waste cycles. In many other cases it is a part of the intent to specify some activity on all cores (or some subset of the cores) in parallel. Indeed, as we generalize action ‘sweep_scenario’ per the requirement above, to target multiple cores concurrently, it becomes mandatory to duplicate the body of the main loop for all the targeted cores in parallel.

We could think of two alternative ways to achieve this with existing PSS constructs:

a. Explicitly duplicate the activity code block for each of the cores, hard-coding the core number in each. However, this is too rigid, as the number of cores in our system varies from one configuration to another. We must keep it as a parameter of this description.
b. Encapsulate the replicated activity block in a new action type and use an action array under the ‘parallel’ statement. The parallel scheduling would apply to the multiple action handles in the array. The down side here is the need to define an “intermediate” action, which is artificial and ad-hoc, being only used in this specific context. If applied in all cases it would make the code more verbose and obscure the flow.

Instead, the code we present here uses a ‘replicate’ statement, with macro-like semantics. It lets us describes iterative expansion of an activity block which is otherwise taking the semantics of the enclosing statement – ‘parallel’ in this case. This is an approach that will be brought for consideration into PSWG.

Another shortcoming of the language we ran into was in not providing a construct to specify default values for data attributes. Consider again the definition of ‘power_down_core’. The attributes ‘Core_C1_delay’ to ‘Core_Cn_delay’ are used as control knobs when this action is called from a higher-level action. However, in many cases the higher-level action would not require any delay, so these attributes would need to assume value 0. Because these attributes are randomized, they must be constrained to 0 explicitly whenever this is the expected outcome. This generally leads to lack of abstraction, where every attribute for any possible use of an action needs to be taken into consideration in its use.

We could not find a workaround for this limitation within the PSS1.0 LRM. Other constraint languages in the domain of verification, such as SystemVerilog, feature “soft” constraints. Soft constraints are used, among other purposes, to capture default values for random variables. In the absence of regular “hard” constraints dictating otherwise, they are satisfied in all solutions.

Our goal is to help evolve Accellera PSS standard with strong engagement with the Portable Stimulus Working group. Our suggestions and comments in this paper align with this goal.

## V. Conclusion

We used PSS strategy described in this paper to a real AMD project with excellent results. We have been able to put together, in a short time and with relatively few lines of code, a PSS model for the scenarios we set as our evaluation goal. The language constructs and modes of expression served the task at hand well. The ability to compose behaviors in arbitrary sequential and parallel scheduling, and to describe conditional and iterative behaviors, provided natural formalization for the flows we had in mind. Data attributes and algebraic constraints over these behaviors allowed us capture abstract properties and controls. Most importantly, the combination of high-level language and tool (Cadence’s Perspec$^{TM}$) kept us focused on the use-cases in the terms described by the architect, without being limited or even concerned with low-level test implementation details.

In addition, graphical representation of the scenarios proved very valuable in reasoning about the scenarios and communicating with the architect and other stakeholders. These parties did not have any prior knowledge of PSS, nor did they need any, in order to grasp the semantics of this graphical language. This is due to the close correspondence of PSS constructs to universally recognized behavioral modeling terms. A team working in lab with minimal exposure to PSS was able to generate variations of scenarios as well as new scenarios quickly to test on silicon.

Test code we generated was initially executed on virtual platform and emulator and shortly afterwards on the silicon board in the lab. We were able to verify that the tests indeed realized the intent we expressed in the high-level language. If we had to implement similar functionality manually in this bare-metal environment, with the complex threading, synchronization, state retention, and other runtime considerations, it would have been an effort of a completely different scale.

We did note a few missing features in the language, that were hard to work around. We elaborated on a couple in this paper. We realize that the Portable Stimulus standard is evolving, and we are looking forward to enhancements that would make it even stronger and easier to use.

## REFERENCES

[1] PSS1.0 Language Reference Manual, Accellera site, http://accellera.org/downloads/standards/portable-stimulus
[2] SystemVerilog 2012 Language Reference Manual, https://standards.ieee.org/standard/1800-2012.html