Low Power Static Verification- Beyond Linting and Corruption Semantics

Kaustav Guha [#1] , Ankush Bagotra [#2], Neha Bajaj [#3]
[#]*Synopsys India Pvt Ltd*
Bangalore, India
[1]kaustav@synopsys.com
[2]ankushb@synopsys.com
[3]nehab@synopsys.com

***Abstract*— UPF2.0 [1], with its ability to define power states and corruption semantics on them, has made low power verification flows powerful. This powerful flow provides more flexibility to a verification engineer to define sophisticated assertions, enabling them to isolate more low power issues in the design. The traditional approach of verifying designs relies on functional state space localized in various blocks, for example an SoC. However, when verifying low power designs, power states transcend hierarchies and are at a Chip Level. Converting existing functional verification platforms into a powerful power aware one is time consuming and error prone. It is not scalable with the number of power domains and with the number and types of low power sequencing checks that one must ideally perform. Reconciling the architecture-wide low power issues and the functionally self-contained design blocks should be the first step, before one can design or conceive a verification flow. Normally, we end up generating a huge set of assertions with very little contextual understanding of the design, both in terms of functionality or power. Even if the context is understood, the process of conceiving assertions is time consuming.**

In this paper, we will highlight how traditional verification flows based on an assertion library framework falls short in low power design verification. To address this issue, we will demonstrate how a static verification tool and UPF 2.0 can be leveraged to conceive powerful low power assertions that are context sensitive, resulting in effective low power verification flows. Until now, the static tools were typically utilized for language semantic checks and structural low power checks. However, we will broaden the scope of static tools and extend it to formulating verification methodologies. We will deal with aspects as to how the assertions can be statically generated and how the coverage of various assertions (generated or otherwise) can be organized.

***Index Terms*—Low Power Verification, Static Verification**

I. INTRODUCTION

A. Low Power Verification

The demand for low power verification led to the formalization of UPF that allows the specification of power semantics that was not possible in HDL. There have been static verification offerings at the HDL level, and low power verification space also provided its own solution. These checks were in addition to linting and other formal verification techniques. With the formalization of UPF 2.0, and most notably the add_power_state semantic, the ability to specify the power states in the design became much more powerful. This has opened many opportunities to verify more corner case scenarios in a standard way.

So far the approach adopted to verify a low power design [2][3][4][5] is to use the low power assertion library, which is packaged with a simulator. The library contains various assertion IPs, which would assert when any illegal low power sequencing occurs in a design. This helps the verification engineer to discover low power issues in the design. However, as the design becomes more complicated, the number of different assertion IPs active, lead to lot more assertions getting fired during verification and the number can explode. The adopted approach for this scenario is to divide and conquer where the assertion IPs are used to verify in a bottom-up approach. The block is white box tested with the assertion IP and then is black/grey boxed when integrating into a higher hierarchy. This effectively manages the situation. But we believe this process can be improved upon by exploiting static verification and UPF 2.0, more effectively. Low power verification is not only a functional issue, but it is a structural issue as well. The dependency between design elements does not respect the hierarchical boundaries, and the issues do not get contextualized in a bottom-up verification approach. A static tool can provide better intelligence to the verification engineer about the cross hierarchy dependency between design elements. It is imperative that with increasing design complexity, an engineer must have better clarity across different reports generated by various tools. For this we adopt the approach of aligning all the tools, simulator in conjunction with all static tools, and achieve a clear set of objectives to ensure a bug free design.

Different static verification tools in conjunction with the simulator do not have a standard semantic framework. We need to correlate different tool runs to align with a global verification objective. For example, the static tool that specializes in clock domain crossing [6] may not be aware of a power context understood by a low power rule checker [7]. This makes any issue that is related to both CDC and low

power difficult to detect. Also, with the power states defined on a domain using the add_power_state of UPF2.0 (henceforth referred to as "power states"), we might want to verify specific relationships among Power States of various domains, functional properties, assertions, and the design rules of a static tool. The multitude of properties/assertions/rules would complicate the verification process, unless there is a way to manage them. As explained earlier, the bottom-up approach may not help in isolating design issues upfront. The ability to correlate different properties (static and functional) will help contextualizing different issues in the design. Having this correlation will ease the debug process, bringing down the turnaround time, as any issue detected has a clear context to debug with.

B. Static Verification

Before we proceed further, let us know the definition of Static Verification.

Definition

Static Verification is a verification of all specified desirable invariant properties and absence of all specified undesirable invariant properties in both the structure and functionality of the design.

The most significant statement to make here is about a property being "invariant", which means a property that always "makes sense". A property is invariant, only if the context of the property and the property itself is made clear in its specification. This is however not true, as we have witnessed in the evolution of low power verification. An HDL assertion is invariant when we verify the design in a functional context. However, with the advent of low power verification, an HDL assertion has to take into account a power context in order to make sense. The HDL semantic space has not kept up with the power semantics required to verify the design, in the context of low power. As a result, the HDL assertion makes sense in context when the logic block is fully powered on, and does not make sense when the block is powered off, the property ceases to be an invariant. A static property asserted by one tool can also cease to be an invariant when some other static property is asserted by another or the same tool. What this means is, unless the static properties of various tool are contextualized, it will be very difficult to make sense of the health of the design given the multitude of report logs.

C. Objective

The objective of this paper is to describe a methodology, wherein the various HDL assertions, static rules, and low power states are contextualized. Once this is done, this methodology provides semantics to formally describe various verification objectives that would help determining the root cause of design issues quickly. We can clearly demonstrate the state of a verification cycle on a given design, by this methodology, in terms of quality, coverage, and signoff.

II. HIGH LEVEL LOW POWER VERIFICATION FRAMEWORK

We introduce High Level Low Power Verification (HLLPV) Framework which is a holistic approach to address low power verification flows and enable non low power verification flows into power-aware ones. The framework brings together assertions from HDL specification, simulation tool, power states, and the static rules of different point tools to ensure that the low power flow is complete. We do this by specifying a "Verification Objective" in terms of logical, low power states, and the static rules. Thus, we introduce the notion of a power_operation. The following code example introduces various components that together constitute a power_operation.

```
power_operation <Name>
   on_definition {domain_list}
   off_definition      {domain state list}
   complete_upon  {all_on |domain state expression | all_off}
   apply_when    -domain_state {<domain state list>}
                 –operation_completed {<power operation list >}
   don't_apply_when  {<list of domain expression>}
   property_with {<domain state expression>,
                  <domain state expression>)
   property_after  {<domain state expression>,
                    <domain state expression>}
   property_before {<domain state expression>,
                    <domain state expression>}
   property_until {<domain state expression>,
                    <domain state expression>}
   assert_on {<domain state expression>}
   user_property_when -state<domain state expression>
                    -checker  <checker file name>
                    –bind_to <domain instance>
   user_assert_when –state <domain state expression>
                          -assertion <assertion file>
                          -bind_to <domain instance>
   disregard_user_property_when      –state      <domain      state
expression>
                    -checker  <checker file name>
                    –bind_to <domain instance>
   disregard_user_assert_when      –state      <domain      state
expression>
                    -checker  <checker file name>
                    –bind_to <domain instance>
  assert_on_rule –rule <rule name> -tool <tool name>
  cover_rule –rule <rule name> -tool <tool name>
end_operation
```

A. Definition:power_operation

A named **power_operation** has a start condition and a terminating condition. The start and terminating condition can be a logical/low power property or a start/stop of other **power_operation**s specified.

One can consider **power_operation** to denote the advancement of verification objectives during verification. However, the onus is on the verification engineer to design and specify the **power_operation** in such a manner, so that the resulting **power_operation** sequence completes the global verification objectives. **power_operation** will be a means of

prioritizing and categorizing high level design states into verification objectives.

A. *Definition: on_definition*

on_definition constitutes a set of domains or UPF power states of a domain that are to be considered powered-on in the **power_operation**. The rationale behind this semantic is that one can define what a domain being powered really means. **off_definition** is the converse of **on_definition** in a **power_operation**.

With this, we can define states that are neither ON nor OFF in a given **power_operation**, or in other words irrelevant to the verification objective being verified.

B. *Definition: apply_when*

apply_when captures the start condition for a **power_operation** to become active. A start condition can be a Boolean expression of power domains states whose ON/OFF is defined using the **on_definition**/**off_definition** of the **power_operation** or when other specified **power_operation** are active or have terminated.
apply_when specifies the dependency of the current verification objective of a named **power_operation** against others or certain power state conditions.

C. *Definition: complete_upon*

complete_upon is used to specify the UPF domain state condition, upon which the **power_operation** would terminate. Having achieved this, the verification engineer can use the terminating condition of the **power_operation** to define new ones.

The following definitions will be used to capture verification objectives in terms of power states, HDL specified properties/assertions, and the static rules of various participating tools.

D. *Definition: **property_*** *

property_with is used to specify two power state expressions that have to be true together. **property_after** is used to specify a power state expression that has to be true, after certain power state expression is true. **property_before** is used to specify a power state expression that has to be true, before certain other power state expression is true. **property_until** is used to specify the power state expression that has to be true, until certain other power state expression is true. These definitions are primarily used to define relationship between the power states specified in the UPF.

E. *Definition: user_*_when*

user_property_when is used to specify the validity and binding of a user-defined checker, based on an expression of domain power states that are relevant in a given **power_operation**. **user_assert_when** is used to specify the validity and binding of a user defined assertion, based on an expression of domain power states that are relevant in a given **power_operation**. **disregard_user_property_when** is a converse of **user_property_when,** and so is **disregard_user_assert_when** is of **user_assert_when**.

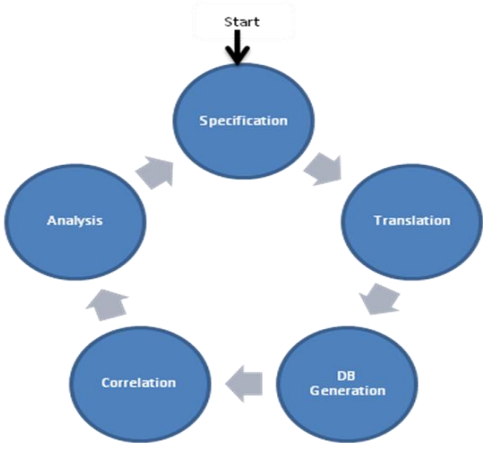
Binding and instrumenting of user specified properties and assertions are powerful ways of laying down verification objectives. This is the heart of the framework and the payload that any **power_operation** carries within. Various **power_operations** can be specified by individual verification teams, all of which can be sorted into a sequence of multiple **power_operation** using the **apply_when** and **complete_upon** semantics. Together, all the **power_operation**s achieve the global verification objective. Depending on how the **power_operation**s are designed, multiple **power_operation**s can be relevant at the same time. A **power_operation** execution can be gated by other **power_operation(s)**. In a single simulation run, various **power_operation(s)** may be active. Depending on which leaf level assertion gets fired, the relevant **power_operation** against which the assertions are specified has to be debugged. Hence, with all the noise generated by the Simulator, the **power_operation** can correlate all the assertion and properties. With multiple teams defining **power_operation** or **power_operation** sequence independently, it leads to more debug parallelism.

F. *Definition: assert_on_rule*

assert_on_rule is used to associate a static rule that is undesirable, when verifying a given power_operation. **cover_rule** is the converse of this semantic, wherein the static rule is necessary to achieve the verification objective of the **power_operation**.

III. FLOW DESCRIPTION

Figure 1 - Various Stages of Execution Cycle in the HLLPV Framework.



A. *Specification*

The verification engineer would use the semantics defined in Section II to describe verification objectives and their

dependency. We term this as the High Level Low Power Verification (HLLPV) intent. Once this intent is specified, the HLLPV Engine (shown in Figure 2) will translate the specification into a Verification flow.

B. Translation

For each verification objective, the tool will generate customized scripts for each point tool to run relevant checks.

C. Database (DB) Generation

The point tools execute the checks and dump their respective DB in an SQL compliant format. The existing point tools, if not SQL complaint, will use an SQL adaptor to dump SQL compliant reports. The DB generation is incremental and re-usable. The DB Generation will also honor the power_operation dependencies laid by the verification engineer.

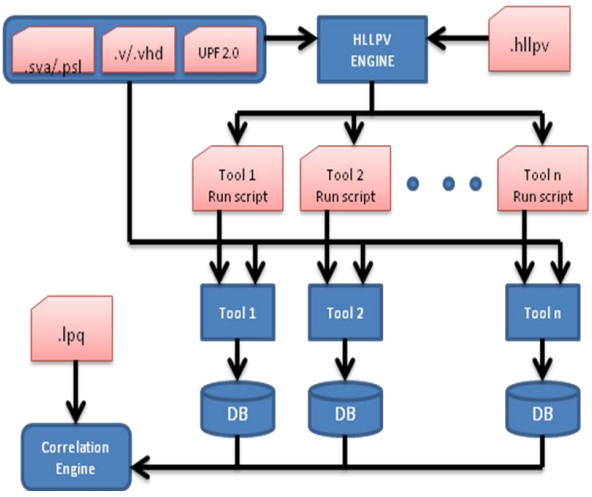
D. Correlation

The DB from various point tools are correlated to present a unified coherent view for analyzing a verification objective.

E. Analysis

Analysis is done on a correlated view using a Query framework, which works over the DB with generic set of queries written on an SQL engine. Each verification goal is validated with the correlated view. In case of a violation, the verification engineer will be able to connect to most debug information using these generic queries. The engineer can query the tool using a .lpq (Low Power Query) file, and can also perform a "what-if" analysis, correlate results with respect to previous results, and isolate the root cause.

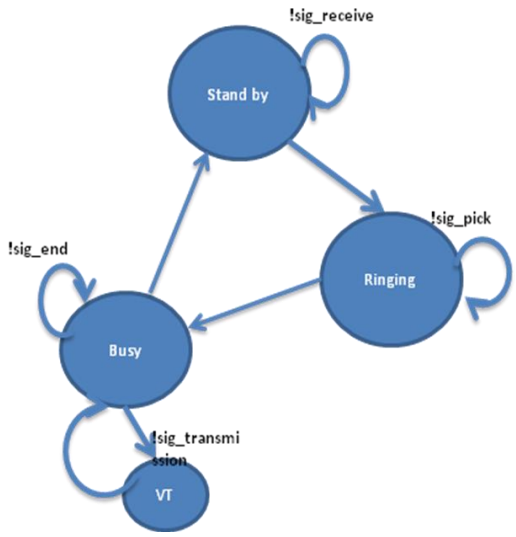
Figure 2: HLLPV Framework



IV. CASE STUDY

Through our case study, we would demonstrate how a design issue is isolated in a cell phone design. The high level Design State Machine (DSM) for this design is shown in Figure 3.

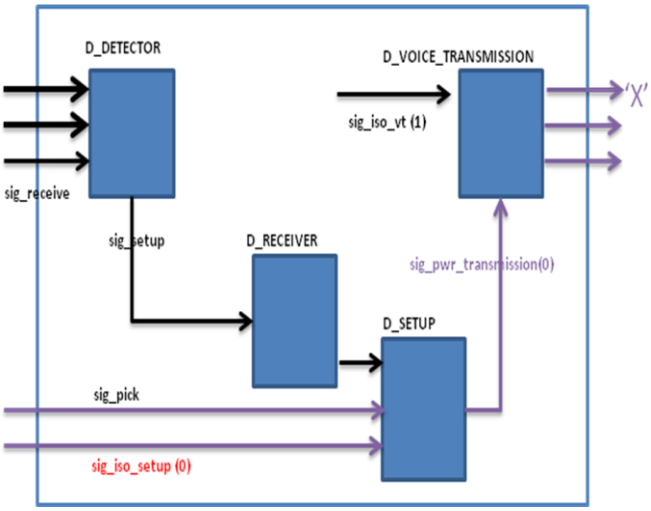
Figure 3: High Level Design State Machine



This state machine is created by the designer, and given to the verification engineer for verification. The framework detected a design issue, which is described as follows:

The voice transmission domain (`D_VOICE_TRANSMISSION`) is powered-off in the "Stand-by" and "Ringing" state. The `D_VOICE_TRANSMISSION` domain has to be powered on when the design is in the "Busy" state. The power switch enable signal `sig_pwr_transmission` for `D_VOICE_TRANSMISSION` domain is getting generated by the `D_SETUP` domain. The isolation enable signal `sig_iso_setup` to the `D_SETUP` domain should be set to logic '1', before the power switch enable signal `sig_pwr_transmission` reaches the `D_VOICE_TRANSMISSION` domain. But `sig_iso_setup` got set to '0', and thereby sending the wrong value ('0') through `sig_pwr_transmission` to the `D_VOICE_TRANSMISSION` domain. Hence, `D_VOICE_TRANSMISSION` domain did not power on when it is supposed to be, resulting in X propagation happening from the outputs of this domain, as the isolation at the output of the domain was set to 1 (as per the correct functionality). This is shown in Figure 4. There were functional assertions within the `D_VOICE_TRANSMISSION`, and they all failed.

Figure 4: Corruption of D_VOICE_TRANSMISSION



We exploit the add_power_state construct from UPF 2.0 to define intended state semantic for each of the domains.

Following add_power_state syntax is defined in the UPF:

```
add_power_state D_SETUP -state {ON_MODE -
supply_eq {VDDI_setup == `{FULL_ON,1.5}} -
state {RUN_MODE -logic_eq sig_iso_setup}
```

This syntax defines that the domain `D_SETUP` is ON when the supply `VDDI_setup` is in `FULL_ON` state, and in `RUN_MODE` when the isolation signal is '1'.

```
add_power_state D_VOICE_TRANSMISSION -state
{STDBY -supply_eq {VDDI_VT == `{OFF}} -state {
ON_MODE -logic_eq sig_pwr_transmission}
```

This syntax defines D_VOICE_TRANSMISSION is in STDBY with supply being OFF, and in ON_MODE when the power signal sig_pwr_transmission is set to 1.

```
add_power_state D_RECEIVER -state {ON_MODE -
simstate NORMAL} -state {SLEEP_MODE -logic_eq
sig_setup}
```

This syntax defines the `SLEEP_MODE` and `ON_MODE` for `D_RECEIVE`.

All the above UPF add_power_state constructs capture the dependency of various signals on the respective domains, as shown in Figure 4.

Although, add_power_state semantic helped in depicting state of the domain in the design, it still did not portray any compliance to the DSM shown in Figure 3. To specify the dependency of various domain states and capture the intent of DSM for verification objective (Vg), following HLLPV intent is defined.

```
power_operation u_standby
     apply_when -domain_state {!D_RECEIVER}
     on_definition {D_RECEIVER.SLEEP_MODE}
     complete_upon { D_RECEIVER.ON_MODE}
end_power_operation
```

This power operation defines when in "Standby" state the `D_RECEIVER` domain should be in `SLEEP_MODE` and when this domain goes to `ON_MODE`, the operation is considered complete.

```
power_operation u_ringing
     apply_when -operation_completed u_standby
      on_definition {D_RECEIVER.ON_MODE}
      property_after {D_SETUP.ON_MODE,
D_RECEIVER.ON_MODE}
end_power_operation
```

This power operation defines that the operation starts only when the `u_standby` power operation is complete. Also, in this power operation, `D_RECEIVER` should be in `ON_MODE` and that the `D_SETUP` should be in `ON_MODE`, only after `D_RECEIVER` is in `ON_MODE`.

```
power_operation u_busy
     apply_when -operation_completed u_ringing
-domain_state {D_VOICE_TRANSMISSION.ON_MODE}
     property_after {
D_VOICE_TRANSMISSION.ON_MODE,
D_SETUP.RUN_MODE}
     assert_on_rule -rule {WRONG_SENSE} -tool
MVRC
     user_property_when -state
{D_VOICE_TRANSMISSION.ON_MODE} -checker
vcs_checker.sva -bind_to
{D_VOICE_TRANSMISSION}
end_power_operation
```

This power_operation defines that it will start only when the `u_ringing` operation is complete and should start also satisfying that `D_VOICE_TRANSMISSION` domain is in `ON_MODE`. This operation is also associated with the MVRC rule `WRONG_SENSE`. Figure 5 displays the relationship among the power_operations defined.

Figure 5: power_operations state machine

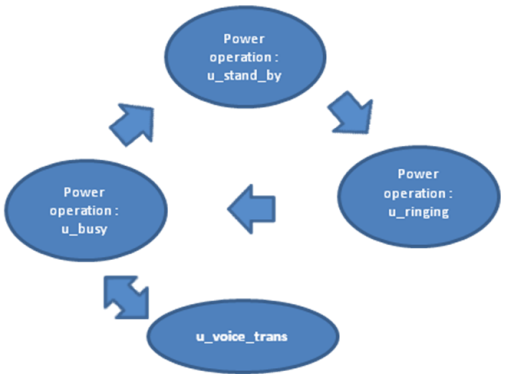
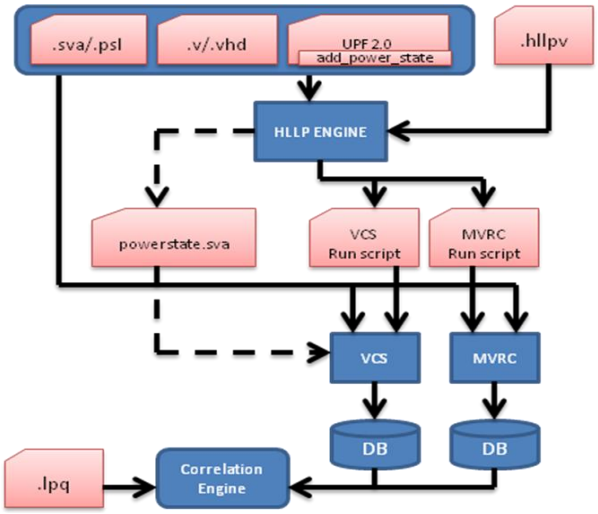


Figure 6: HLLPV flow in Case Study



The HLLPV engine generates a customized script for all the point tools used, in our case VCS [8] and MVRC, and also the powerstate.sva assertion file. The powerstate.sva file contains the power state assertions defined in the .hllpv file, instrumented in the form of a System Verilog Assertions (SVA) [9] which is understood by VCS. In order to debug from a functional point of view, traditionally, one has to trace back from all the outputs that have 'X' propagation, instead, `WRONG_SENSE` rule in MVRC has pointed out the root cause of the above issue- "The power switch signal powering on the `D_VOICE_TRANSMISSION` domain has a `WRONG_SENSE`, because of the powering off of `D_SETUP` domain". The Correlation Engine constructs a coherent view using the VCS and MVRC DBs, and provides reports for the status and coverage of Vg. This is the advantage that HLLPV brings to the low power verification. Figure 6 shows the flow used while achieving Vg.

The .lpq file (Figure 6) includes all the queries we had pertaining to status of Vg. We queried for the status of various power_operation formulate quality, coverage, and signoff metrics for the design. Thanks to dependencies defined among power_operations.

Table 1: Reference designs verification results

Design	# Domain	# Known Issues	# Root Issues caught by MVRC	# Root Issues caught by assertions in VCS	# Root Issues caught by HLLPV
Ref_design1	5	37	10	20	7
Ref_deisgn2	9	63	21	30	12

Our flow was used to validate some reference designs. The results of the respective runs are shown in Table 1. The table highlights the value that HLLPV brings to the low power verification. It represents issues, which were discovered using MVRC, VCS, and HLLPV. The root causes were isolated faster than the traditional flow, since we could correlate both the VCS simulation data and the MVRC logs.

V. Conclusion

In this paper, we have demonstrated the HLLPV framework. It can leverage different Static Verification tools in conjunction with the Assertion Based Verification flows, in the context of UPF2.0 (add_power_state). This provides a holistic solution to the verification problem of low power designs. The framework also provides the ability to define verification objectives that can be used to define metrics in terms of quality, coverage, and signoff. In doing so, it brings down the turnaround time for fixing a design issue and effectively portrays the accurate status of the verification exercise.

VI. Future Work

Our future goal is to incorporate other static verification tools in HLLPV framework. We are looking into the potential value at the Translation stage of HLLPV intent, wherein we critique and optimize the global structure of various power_operations defined.

References

[1] *Unified Power Format (UPF 2.0) Standard*; IEEE Draft Standard for Design and Verification of Low Power Integrated Circuits, IEEE P1801/D18; 23rd October, 2008.

[2] Khan N. and Winkeler W; *Power Assertions and Coverage for Improving Quality of Low Power Verification and Closure of Power Intent*; In the Proceedings of DVCon, pp. 53-58, 2008.

[3] Chidolue G. and Ramanandin B.; *Upping Verification Productivity of Low Power Designs*; In the Proceedings of DVCon, pp. 3-10, 2008.

[4] Jadcherla S., Bergeron J., Inoue Y., Flynn D.; *Verification Methodology Manual for Low Power (VMM-LP)*; February 2009.

[5] Keating M., Flynn D., Aitken R., Gibbons A. and Shi K.; *Low Power Methodology Manual (LPMM) – For System-on-Chip Design*; 2nd Edition, Springer, 2008.

[6] MVRC from Synopsys, http://www.synopsys.com/TOOLS/VERIFICATION/LOWPOWERVERIFICATION/Pages/MVRC.aspx

[7] Leda from Synopsys, http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/Leda.aspx

[8] VCS from Synopsys http://www.synopsys.com/tools/verification/functionalverification/pages/vcs.aspx

[9] *System Verilog from Accellera*; http://www.systemverilog.org/.