

UPF Code Coverage and Corresponding Power Domain Hierarchical Tree for Debugging

Shang-Wei Tu MediaTek Inc. +886-3-5670766#23493 kuma.tu@mediatek.com	Tom Lin Synopsys Inc. +886-3-5581860 tomlin@synopsys.com	Archie Feng Synopsys Inc. +1-650-5844915 archie.f@synopsys.com	Chen Ya Ping Synopsys Inc. +86-592-3012457 lukechen@synopsys.com
---	---	---	---

Abstract- The functional coverage together with the code coverage is essential for completely verifying a design. This is also true for the low power verification. However, traditionally, verification engineers only consider the RTL code for generating and collecting the code coverage. Nowadays, a design usually comprises two parts. One is the functional part described in the hardware language such as Verilog or VHDL, and the other is the power management part described in the power format language such as UPF or CPF. Hence, to completely verify the design, we have to adopt the power-aware simulation instead of the pure RTL simulation. However, we found the low power verification is incomplete without a suitable coverage model to cover the UPF part. In this paper, we proposed the concept of the UPF code coverage to fill in the missing piece of the low power verification. To implement and prove the concept, we collaborated with Synopsys to implement this concept into their simulator. In addition, the Power Domain Hierarchical Tree is proposed to facilitate debugging and reviewing the UPF code coverage. The result is also demonstrated in this paper.

Keywords—low power verification; UPF; code coverage; power-aware simulation; power domain hierarchical tree

I. INTRODUCTION

When the process technology and the design complexity continue advancing, the leakage power consumes over 50% of the total power for portable devices from year 2012 as shown in **Figure 1**. However, traditional low power techniques such as the clock gating, the logic optimization, and the multi-Vt optimization cannot handle the leakage power effectively [2]. Therefore, various advanced low power techniques are employed in the industry [3]. This further increases the complexity of the hardware and the corresponding software, and both of them become highly risky when not being well verified. This is why we require the low power verification methodology to minimize the risk of both the hardware and the software.

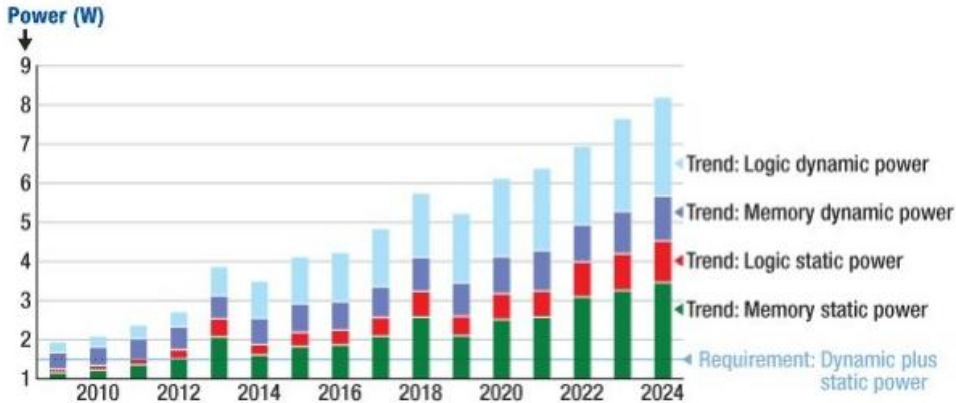


Figure 1. SOC consumer portable power consumption trends [1].

The coverage is an important index for measuring the completeness of the verification, and the same concept can also be applied to the low power verification as well. There are two categories of the coverage. One is the functional coverage, and the other is the code coverage. Then the upcoming question is that “Is either one of them enough for measuring the completeness of the verification?”. The answer is negative. The code coverage measures how thoroughly your tests exercise the “implementation” of the design specification, but not the verification plan. In other words, the code coverage does not provide the measurement of how well the functionality is verified. Consider a case that you have 100% code coverage, but there is one function left unimplemented. For this case, the functional coverage will reveal the missing functionality. On the other hand, the functional coverage is tied to the design intent and is sometimes called “specification coverage”. However, with the functional coverage only, corner cases could be missed even when the functional coverage is 100% covered. Consider a case that you have 100% functional coverage, but there are still some FSM states or some IO ports left not toggled. The code coverage will reveal the uncovered corners. Therefore, we will have high confidence of the verification quality only when both the functional coverage and the code coverage are considered together [4].

Nowadays, a low power design comprises two parts. One is the functional part implemented with the IEEE HDL language—Verilog/VHDL, and the other is the power management part implemented with the IEEE power intent language—UPF. In other words, UPF also describes a part of a design, since it will be synthesized and implemented with the RTL code. Although current vendor tools can generate the code coverage to cover the RTL code very well, they do not address the code coverage of the UPF part or provide the same user interface for the UPF code coverage as that of the RTL code. Besides, from our literature survey, we do not find any literature address this missing piece of the low power verification either. However, the UPF code coverage is essential for the low power verification. We cannot measure how thoroughly the power management design is exercised without coverage directly generated from the UPF code. For example, we don’t have any index to measure if all isolation rules are toggled without the UPF code coverage. Hence, to fill in the missing piece of the low power verification, we proposed the concept of the UPF code coverage and collaborated with Synopsys to implement into their simulator and prove the concept with the real project execution. In addition, the Power Domain Hierarchical Tree is proposed to facilitate debugging and reviewing the UPF code coverage. The result is also demonstrated in this paper.

The rest of this paper is organized as follows. Section II describes the features of the code coverage. Section III details the definition of the UPF code coverage and the corresponding interface provided by VCS NLP. In Section IV, the concept of the Power Domain Hierarchical Tree is briefly explained. Following, we demonstrate the debugging solutions implemented with the Power Domain Hierarchical Tree for the UPF code coverage. Finally, Section VI concludes this paper and discusses the plan for future work.

II. FEATURES OF CODE COVERAGE

Before giving the detailed definition of the UPF code coverage, we should first review the features that should be contained by the code coverage. Those features are key indices for us to inspect if the defined UPF code coverage is similar to the RTL code coverage. The similarity is crucial for the verification engineer to adopt the UPF code coverage seamlessly just like RTL code coverage, since as mentioned in [5], the efficiency is also a key factor of the successful verification. From our experience on the execution with the RTL code coverage, we list the features of the code coverage below:

- (1). Have clear definition
- (2). Directly derived from source code
- (3). Coverage automatically collected by tool
- (4). Can be selectively turned on
- (5). Have debugging scheme
- (6). Have user friendly coverage report
- (7). Easy to exclude and merge

We believe that if all above features are contained in the UPF code coverage, we can easily adopt the UPF code coverage to measure the completeness of the low power verification. In Section III, we will cover feature (1) to (4), and in Section V, we will cover feature (5) and (6). However, feature (7) is not covered in this paper, since these are still under discussion with Synopsys.

III. DEFINITION OF UPF CODE COVERAGE

In following subsections, we will give detailed definitions of the UPF code coverage and the corresponding VCS options to turn on.

A. Port State Coverage

The port state coverage is defined for below UPF command:

```
add_port_state port_name {-state {name value}}*
```

and the corresponding compile-time option is:

```
-power=cov_port_state
```

After turning on the port state coverage, there are two coverage types generated. One is the state coverage of each state, and the other is the transition coverage between the states. For example, if we have below UPF command and turn on the port state coverage, we will get 3 state coverage points and 6 transition coverage points generated:

```
add_port_state DVDD_DVFS -state {ON 1.0} -state {HI 1.2} -state {LO 0.8}
```

With the port state coverage, we can check if our testbench can toggle all defined port states. For some extreme cases, if we have some port states can never be toggled, it could be either redundant states or even missing connection for the corresponding ports.

B. Power Switch Coverage

The power switch coverage is defined for below UPF command:

```
create_power_switch switch_name [-domain domain_name]
  -output_supply_port {port_name [supply_net_name]}
  {-input_supply_port {port_name [supply_net_name]}}*
  {-control_port {port_name [net_name]}}*
  [-ack_port {port_name net_name [logic_value]}}*
  {-on_state {state_name input_supply_port {boolean_expr}}}*
  [-off_state {state_name {boolean_expr}}]*
```

and the corresponding compile-time option is:

```
-power=cov_psw
```

After turning on the power switch coverage, there are two coverage types generated for the power switch states (i.e., **-on_state** and **-off_state**), the control ports, and the acknowledge ports. One is the state/level coverage of each state/port, and the other is the transition coverage between the states/levels. For example, if we have below UPF command and turn on the power switch coverage, we will get 2 power switch state coverage points, 2 level coverage points of the control port, 2 level coverage points of the acknowledge port, and 2 transition coverage points generated for each of them:

```
create_power_switch PSW -domain PD
  -output_supply_port {vout DVDDO}
  -input_supply_port {vin DVDDI}
  -control_port {ctrl pwr_on}
```

```

-ack_port {ack pwr_ack}
-on_state {PSW_ON vin {ctrl}}
-off_state {PSW_OFF {!ctrl}}

```

With the power switch coverage, we can examine if our testbench have toggled all power switch rules. This coverage is extremely important for the power-aware simulation, since many corner cases can be covered by reviewing this coverage. In our project execution, we consider this power switch coverage must be fully covered in the power-aware simulation.

C. Power State Table Coverage

The Power State Table (PST) coverage is defined for below UPF commands:

```

create_pst table_name -supplies supply_list
add_pst_state state_name -pst table_name -state supply_states

```

and the corresponding compile-time options are:

```

-power=cov_pst
-power=cov_pst_state
-power=cov_pst_transition

```

“-power=cov_pst_state” enables only the state coverage of the PST, and “-power=cov_pst_transition” enables only the transition coverage of the PST. “-power=cov_pst” enables both the state and transition coverage of the PST. For example, if we have below UPF command and turn on the PST coverage (i.e., -power=cov_pst), we will get 3 state coverage points and 6 transition coverage points generated:

```

create_pst PST_M -supplies {VDD VDD_SW VDDDB VSS}
add_pst_state Normal -pst PST_M -state {ON ON ON GND}
add_pst_state Sleep -pst PST_M -state {ON OFF ON GND}
add_pst_state OFF -pst PST_M -state {OFF OFF ON GND}

```

With the PST coverage, we can review if all the legal power modes are fully covered. This coverage is also extremely important for the low power verification, since the power modes represent the use cases which are defined by the power architect for the software engineers to use. In other words, all allowable power modes are defined in the PST. The software engineers are not supposed to use any power mode beyond the PST. In addition, for the macros and IPs, we are interested in the correctness of the integration in the top design. The PST coverage also gives us an index for measuring the correctness of the integration. Hence, in our project execution, we also consider this PST coverage must be fully covered in the power-aware simulation.

D. Retention Coverage

The retention coverage is defined for below UPF commands:

```

set_retention retention_name -domain domain_name
[-elements element_list]
[-retention_power_net net_name]
[-retention_ground_net net_name]
set_retention_control retention_name -domain domain_name
[-save_signal {logic_net <high/low/posedge/negedge>}]
[-restore_signal {logic_net <high/low/posedge/negedge>}]
[-save_condition {boolean_expression}]
[-restore_condition {boolean_expression}]

```

and the corresponding compile-time option is:

```

-power=cov_ret

```

After turning on the retention coverage, there are two coverage types generated. One is the active and inactive coverage of each retention signal/condition, and the other is the transition coverage between the active and inactive state. For example, if we have below UPF command and turn on the retention coverage, we will get 4 active and inactive coverage points and 4 transition coverage points generated for the save and restore signal:

```
set_retention RR -domain PD
  -retention_power_net DVDD_BCK
  -retention_ground_net DVSS
set_retention_control RR -domain PD
  -save_signal {save high}
  -restore_signal {restore high}
```

With the retention coverage, we can measure the verification completeness of the retention rules defined in the UPF file. On the other hand, without this coverage, we could have some retention logics left unverified and, hence, this could result in some verification holes.

E. Isolation Coverage

The isolation coverage is defined for below UPF commands:

```
set_isolation isolation_name -domain domain_name
  [-elements element_list]
  [-applies_to <inputs/outputs/both>]
  [-source source_supply_ref]
  [-sink sink_supply_ref]
  [-clamp_value <0/1/latch/Z>]
  [-isolation_power_net net_name]
  [-isolation_ground_net net_name]
set_isolation_control isolation_name -domain domain_name
  -isolation_signal signal_name
  [-isolation_sense <high/low>]
  [-location <self/parent>]
```

and the corresponding compile-time option is:

```
-power=cov_iso
```

After turning on the isolation coverage, there are two coverage types generated. One is the active and inactive coverage of the isolation control signal, and the other is the transition coverage between the active and inactive state. For example, if we have below UPF command and turn on the isolation coverage, we will get active and inactive coverage point and 2 transition coverage points generated for the isolation control signal:

```
set_isolation ISO -domain PD
  -isolation_power_net DVDD_BCK
  -isolation_ground_net DVSS
  -clamp_value 0
  -applies_to outputs
set_isolation_control ISO -domain PD
  -isolation_signal iso
  -isolation_sense high
  -location self
```

With the isolation coverage, we can measure the verification completeness of the isolation rules defined in the UPF file. On the other hand, without this coverage, we could have some isolation logics left unverified and, hence, this could result in some verification holes.

IV. POWER DOMAIN HIERARCHICAL TREE

A. Background and Assumption

Among all existing tools, we found that they are either flattening view or design-hierarchy-based view for reviewing the power domains defined in UPF. To make the power domain view structuralized, we propose the Power Domain Hierarchical Tree in this paper. However, we have a basic assumption for this solution. The assumption is that real projects will adopt the hierarchical UPF coding style. In other words, most reusable IPs and sub-systems will be delivered with corresponding UPF files for top integrators to integrate. Hence, top integrators can easily integrate those UPF files with below UPF command:

load_upf "IP.upf" -scope some_hier/ip0

We think the hierarchical UPF coding style is a common practice in real projects due to several factors. First is the increasing complexity of the design power intent, so it is not realistic to create a flattening UPF for a SoC. Second is the reuse of IPs. IP providers should provide both Verilog and UPF for customers to integrate, verify, and implement. Third is the trend of the bottom-up hierarchical verification and implementation flow, so stand along UPF files are required for this flow.

Based on the assumption of the hierarchical UPF coding style, we propose the Power Domain Hierarchical Tree [6] to systematically organize the power domain structure. With the structuralized power domain hierarchy, designers can review and debug their power intent efficiently.

B. Criteria for Creating Power Domain Hierarchical Tree

Below are criteria for creating the Power Domain Hierarchical Tree:

- 1). Power domain hierarchy should base on the design hierarchy
- 2). Branch node will be created only when “changing scope” and “creating base domain” happen concurrently
- 3). Every branch node should record its current scope and parent scope
- 4). If *PD_X* is defined with *create_power_domain PD_X -elements {list_of_inst}*, then *PD_X* is a member of current branch node
 - i. Use current scope as a hash key to find out the branch node which it belongs to
 - ii. If cannot find a branch node with current scope, then use parent scope to find
- 5). To determine which branch node a given PST (power state table) belongs to, we can also adopt the same method listed in i. and ii.

The branch node represents the tree node in the Power Domain Hierarchical Tree, and the members of a branch node can be non-default power domains and PSTs.

C. Example

In this subsection, we use a simple example as illustrated in **Figure 2** to demonstrate the creation of the Power Domain Hierarchical Tree with the criteria listed in previous subsection.

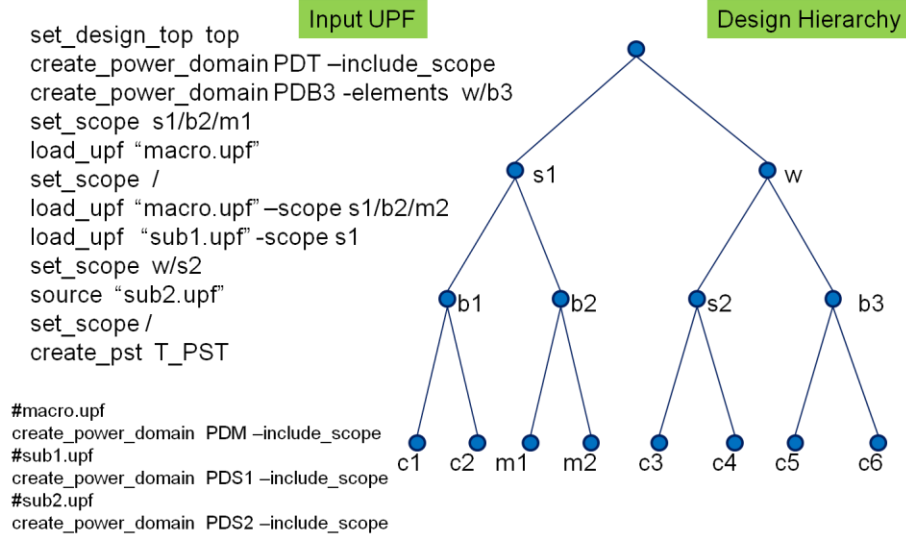


Figure 2. Example UPF code and design hierarchy.

When parsing first 2 UPF commands, criterion 2) will be applied, since a new scope with a new default domain *PDT* are created. Hence, a branch node *PDT* will be created as shown in **Figure 3**. Then, after parsing the third UPF command, criterion 4) will be applied, since a normal power domain is created. Therefore, a member *PDB3* belonging to branch node *PDT* will be created and the data structure will also be updated as demonstrated in **Figure 3**.

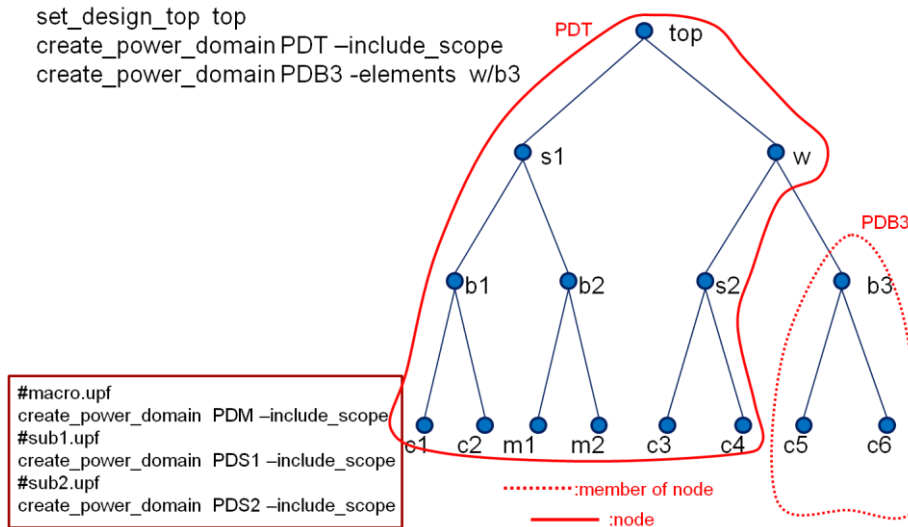


Figure 3. Data structure after parsing first 3 UPF commands.

Next, after parsing successive 4 UPF commands, criterion 1) and 2) will be applied, since two scope changes (i.e., *set_scope* and *-scope*) with two creations of new default domains *PDM* (introduced by *load_upf*) happen respectively. Hence, two new branch nodes with name *PDM* will be created under *PDT* and the data structure will also be updated accordingly as demonstrated in **Figure 4**. In addition, criterion 3) will be applied and the parent scope “/” (i.e., *top*) will be recorded for the new branch nodes.

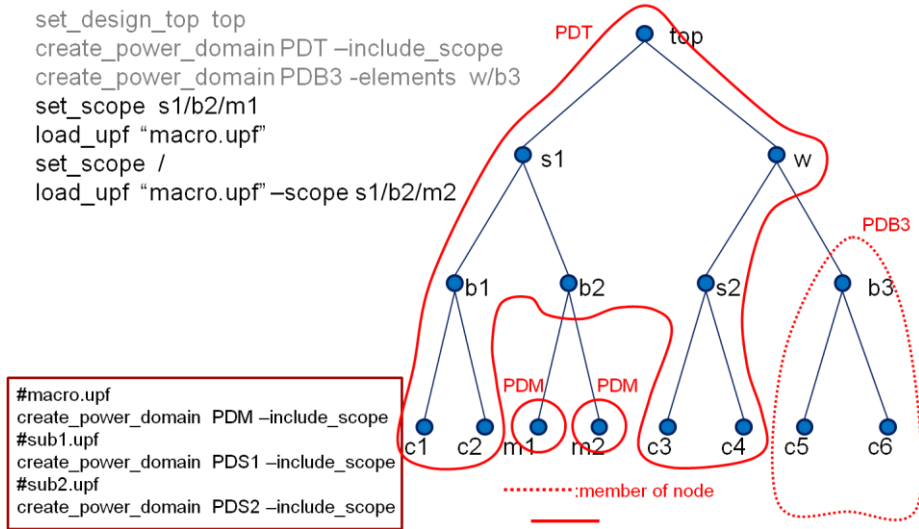


Figure 4. Data structure after parsing following 4 UPF commands.

Still next, after parsing following 3 UPF commands, criterion 1) and 2) will be applied, since two scope changes (i.e., *-scope* and *set_scope*) with two creations of new default domains *PDS1* and *PDS2* (introduced by *load_upf* and *source*) happen respectively. Hence, two new branch nodes *PDS1* and *PDS2* will be created as demonstrated in **Figure 5**. Then, criterion 3) will be applied. The parent scope “/” will be recorded for the new branch nodes, and the parent scope of two branch nodes *PDM* will be updated to “*s1*” to keep the hierarchy correct. Finally, after parsing last 2 UPF commands, criterion 5) will be applied. Since current scope is changed to “/”, *T_PST* will be created as a member of the branch node *PDT* as demonstrated in **Figure 5**.

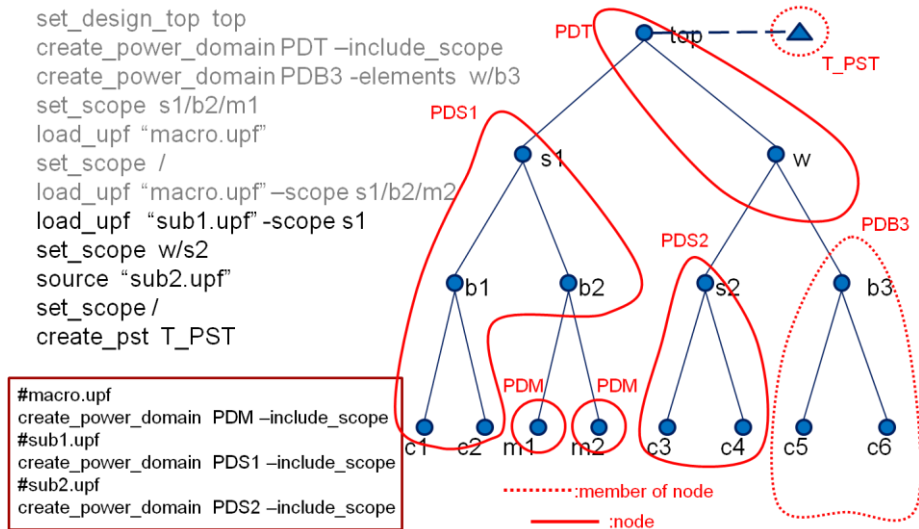


Figure 5. Data structure after parsing last 5 UPF commands.

After all UPF commands are parsed, we can easily draw the Power Domain Hierarchical Tree as demonstrated in **Figure 6** from the latest data structure in **Figure 5**. Since there could be some IPs’ or macros’ UPF loaded several times, you will see several branch nodes with the same domain name but different scope such as *PDM* in **Figure 6**. For simplicity, we can also group them as a supper branch node. With this hierarchical tree, we can easily review and debugging our power intent comparing to a flattening view.

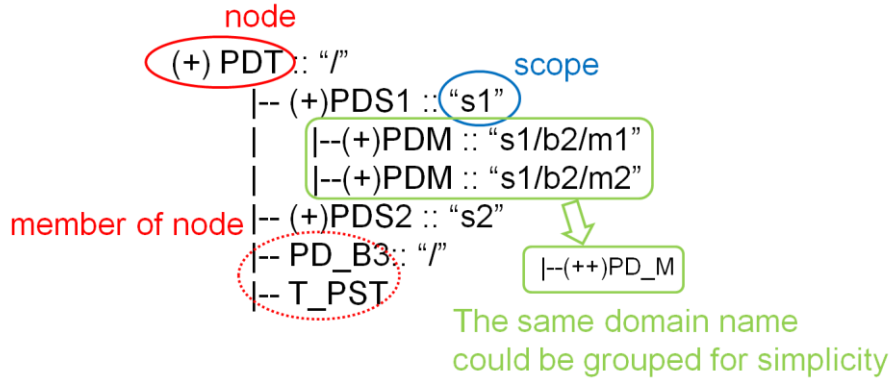


Figure 6. Generated Power Domain Hierarchical Tree.

V. APPLICATION OF POWER DOMAIN HIERARCHICAL TREE

The concepts of the UPF code coverage and the Power Domain Hierarchical tree are both tool independent. In other words, they can be implemented in any tool. In this section, we demonstrate the results of applying the Power Domain Hierarchical Tree to commercial tools. We collaborate with Synopsys to implement this concept into URG low power coverage report and Verdi coverage debugging GUI for the UPF code coverage. In addition, this concept can also be applied to power intent debugging GUI. The result of Verdi-PD (Verdi Power-aware Debug) with the Power Domain Hierarchical Tree is also demonstrated in this section.

A. URG Low Power Coverage Report

One solution provided by Synopsys for reviewing the UPF code coverage is the HTML report generated by URG. The original HTML report generated by URG is demonstrated in **Figure 7**. This is just a kind of raw data, and engineers are difficult to review the UPF coverage with the raw data.

SCORE	WEIGHT	GOAL	NAME
12.50	1	100	tb/top0/WRAP0_PST.pst_state
12.50	1	100	tb/top0/WRAP1_PST.pst_state
12.50	1	100	tb/top0/WRAP2_PST.pst_state
12.50	1	100	tb/top0/WRAP3_PST.pst_state
12.50	1	100	tb/top0/PSW_WRAP0/vout.supply_state
12.50	1	100	tb/top0/PSW_WRAP1/vout.supply_state
12.50	1	100	tb/top0/PSW_WRAP2/vout.supply_state
12.50	1	100	tb/top0/PSW_WRAP3/vout.supply_state
12.50	1	100	tb/top0/SS_DVFS.power_state
16.67	1	100	tb/top0/TOP_PST.pst_state
16.67	1	100	tb/top0/sub10/wrap0/stop0/aaa0/bbb0/cc0/macro0/MACRO_PST.pst_state

Figure 7. Original HTML report generated by URG for the UPF coverage.

After applying the Power Domain Hierarchical Tree and the concept of the UPF code coverage, the new HTML report is demonstrated in **Figure 8**. Obviously, the structuralized coverage data is easier for engineers to review. In addition, the coverage report structure is analogous to the RTL code coverage report demonstrated in **Figure 9**. Engineers will quickly get familiar with the similar report style and do the UPF code coverage review as well as the RTL code coverage review.

LP Hierarchy

[dashboard](#) | [hierarchy](#) | [modlist](#) | [groups](#) | [tests](#) | [asserts](#) | [UPF LP](#)

NAME	SCORE	PST	PSW	SUPPLY_STATE	PD	ISO	RET	ESS_SIMSTATE
PD_TOP:/	23.58	18.06	26.46	27.03	26.14	25.28	25.45	16.67
PD_MACRO								
PD_SUB2::sub20	43.76	45.06	51.23	50.00	50.00	31.25	35.00	
PD_IP								
PD_IP::sub20/ip0	24.07	17.59	26.85	25.00	25.00	25.00	25.00	
PD_IP::sub20/ip1	24.07	17.59	26.85	25.00	25.00	25.00	25.00	
PD_MACRO								
PD_MACRO::sub20/ip1/func0/macro0	23.61	16.67	25.00	25.00	25.00	25.00	25.00	
PD_MACRO::sub20/ip1/func0/macro1	23.61	16.67	25.00	25.00	25.00	25.00	25.00	
PD_MACRO::sub20/ip1/func0/macro2	23.61	16.67	25.00	25.00	25.00	25.00	25.00	
PD_MACRO::sub20/ip1/func0/macro3	23.61	16.67	25.00	25.00	25.00	25.00	25.00	
PD_MACRO::sub20/ip1/func1/macro0	23.61	16.67	25.00	25.00	25.00	25.00	25.00	
PD_MACRO::sub20/ip1/func1/macro1	23.61	16.67	25.00	25.00	25.00	25.00	25.00	
PD_MACRO::sub20/ip1/func1/macro2	23.61	16.67	25.00	25.00	25.00	25.00	25.00	
PD_MACRO::sub20/ip1/func1/macro3	23.61	16.67	25.00	25.00	25.00	25.00	25.00	
PD_WRAP0:/	25.00				25.00	25.00	25.00	
PD_WRAP1:/	25.00				25.00	25.00	25.00	
PD_WRAP2:/	25.00				25.00	25.00	25.00	
PD_WRAP3:/	25.00				25.00	25.00	25.00	

Figure 8. Enhanced HTML report with the Power Domain Hierarchical Tree and the concept of the UPF code coverage.











































Summary														 	
Hierarchy Modules Groups Asserts Statistics															
		Name	Score	Line	T	U	Toggle	T	U	FSM	T	U	Metrics		
		top	 28.69%	 47.25%	4626	2440	 28.62%	8006	5715	 10.19%	540	485	<input checked="" type="checkbox"/> All		
		dut	 28.66%	 47.22%	4623	2440	 28.56%	8000	5715	 10.19%	540	485	<input checked="" type="checkbox"/> Line		
		l2c_th_top0	 28.58%	 47.19%	924	488	 28.38%	1596	1143	 10.19%	108	97	<input checked="" type="checkbox"/> Toggle		
		l2c_th_top1	 28.58%	 47.19%	924	488	 28.38%	1596	1143	 10.19%	108	97	<input checked="" type="checkbox"/> FSM		
		l2c_th_top2	 28.58%	 47.19%	924	488	 28.38%	1596	1143	 10.19%	108	97	<input type="checkbox"/> Condition		
		l2c_th_top3	 28.58%	 47.19%	924	488	 28.38%	1596	1143	 10.19%	108	97	<input type="checkbox"/> Branch		
		l2c_th_top4	 28.58%	 47.19%	924	488	 28.38%	1596	1143	 10.19%	108	97	<input type="checkbox"/> Assert		
		l2c0	 100.00%				 100.00%	4	0				Cover Items		

Figure 9. Example of the RTL code coverage report.

B. Verdi Coverage GUI for UPF

Another solution provided by Synopsys for reviewing the UPF code coverage is the Verdi Coverage GUI. The original solution is DVE GUI coverage view as shown in **Figure 10**. This is also just a kind of raw data, and engineers are difficult to review or debug the UPF coverage with the raw data.

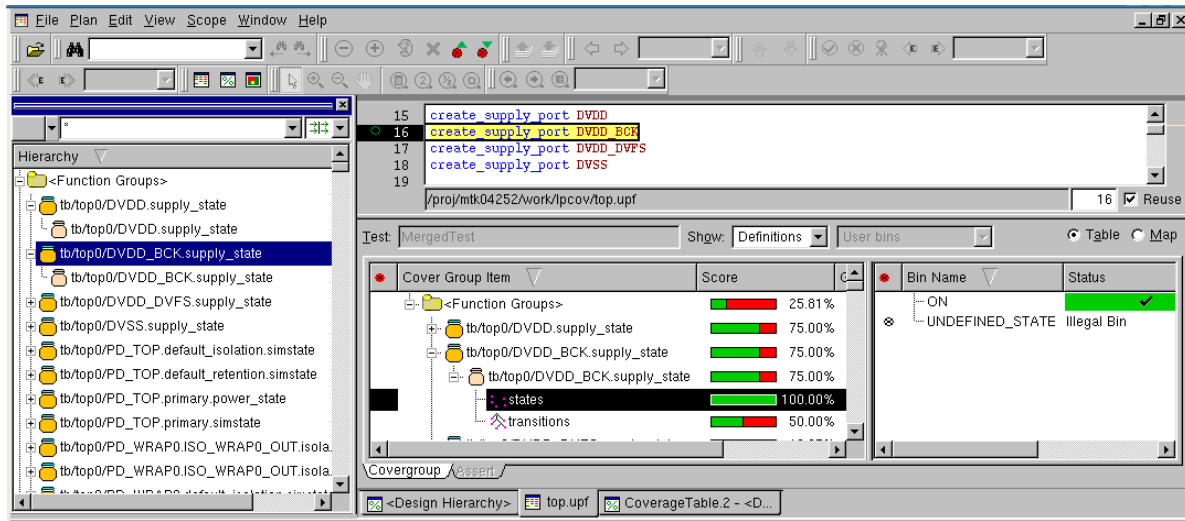


Figure 10. DVE GUI coverage view for the UPF coverage.

After applying the Power Domain Hierarchical Tree and the concept of the UPF code coverage, the new Verdi coverage GUI is demonstrated in **Figure 11**. Obviously, the structuralized coverage data is easier for engineers to review and debug. In addition, the coverage report structure is analogous to the RTL code coverage report demonstrated in **Figure 9**. Engineers will find no difficulty to get familiar with the similar data structure and do the UPF code coverage review and debugging efficiently.

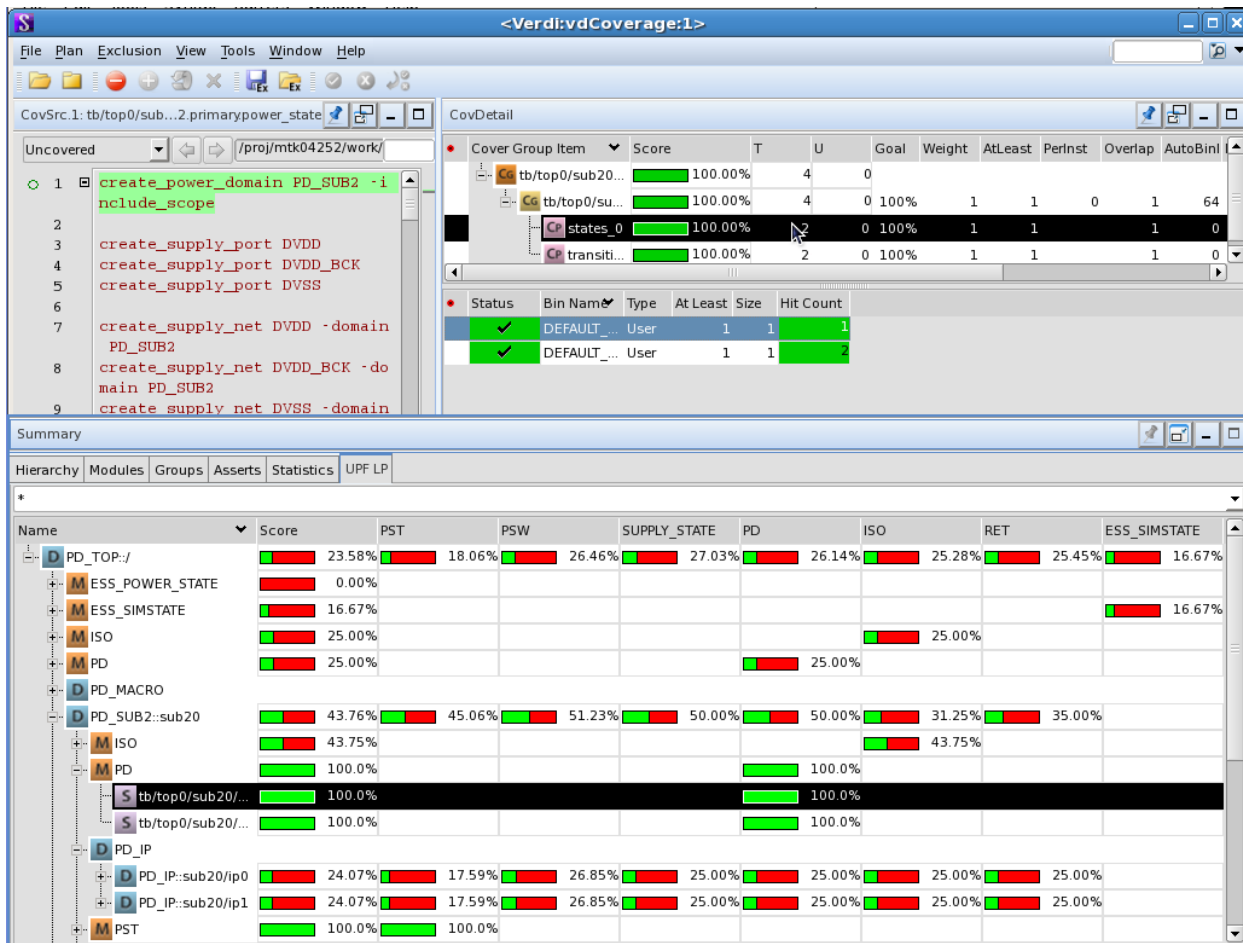


Figure 11. Enhanced Verdi coverage GUI with the Power Domain Hierarchical Tree and the concept of the UPF code coverage.

C. Verdi-PD Debugging GUI for UPF

When we want to review our power intent defined in UPF or debug power-aware simulation failures, we usually open a debugging GUI and hope it can help us do the job efficiently. However, the solutions provided by vendors are all kinds of a flatten view of UPF domain as shown in **Figure 12**. It is difficult for us to find a certain power domain in the flattening view and there is no any power domain hierarchy relation provided.

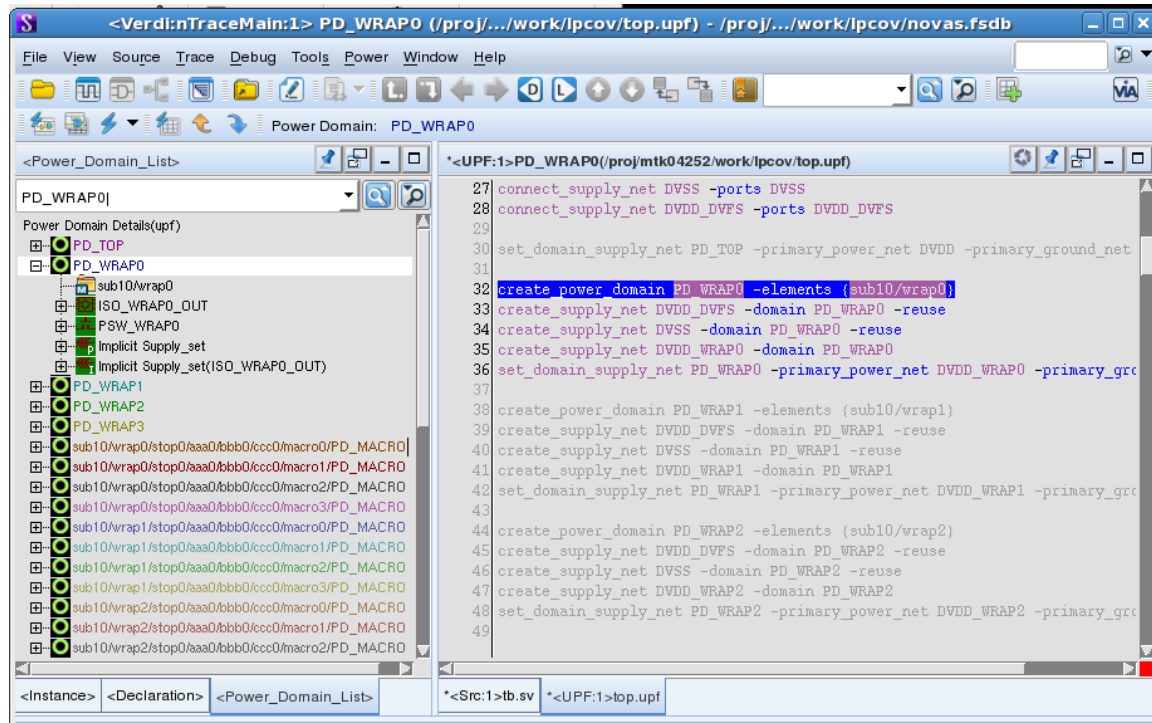


Figure 12. Original Verdi-PD debugging GUI for UPF.

After applying the Power Domain Hierarchical Tree to Verdi-PD, the new UPF debugging GUI is demonstrated in **Figure 13**. Obviously, the well organized tree structure is easier for engineers to review or debug than the previous one. In addition, the multiple macro power domains are grouped for simplicity. The enhancement of the debugging GUI is obvious.

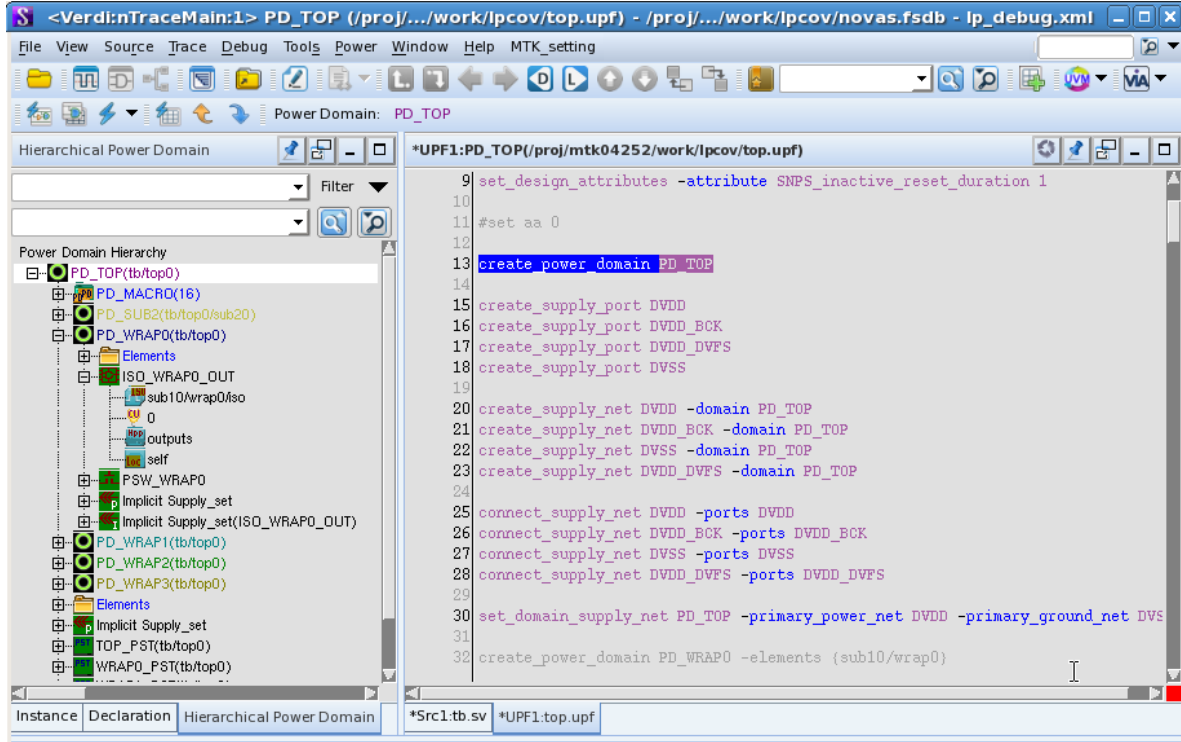


Figure 13. Enhanced Verdi-PD GUI with the Power Domain Hierarchical Tree for UPF debugging.

VI. CONCLUSION AND FUTURE WORK

In the paper, we propose the concept of the UPF code coverage to fill the missing piece of the low power verification methodology. To make this concept practical, we collaborate with Synopsys to implement this concept into their power-aware tools including MVSIM NLP, URG, and Verdi. In addition, this concept is general and it is not supposed to be tool specific. It can also be implemented in the simulators of other vendors.

The Power Domain Hierarchical Tree is also proposed in this paper for the practical consideration of executing the UPF code coverage in real projects. The results are demonstrated in this paper as well. Again, this concept is general and is not supposed to be tool specific.

There are three future works to be planned. First is the UPF code coverage definition for covering full UPF commands. Second is the UPF code coverage exclusion mechanism. The last one is applying the Power Domain Hierarchical Tree to a UPF editor.

ACKNOWLEDGMENT

The authors wish to thank the following for their contributions and support.

- Tiger Hsu of Synopsys Inc. provides the supportive and thorough reviews.
- Sean Lin of Synopsys Inc. bridges the customer and the research and development engineers of Synopsys.
- Research and development engineers of Synopsys Inc. implement these ideas.

REFERENCES

- [1] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors*, 2010; <http://public.itrs.net>.
- [2] Synopsys presentation slide, *Low Power Design: Galaxy Power Implementation*, 2013.
- [3] Srikanth Jadcherla, Janick Bergeron, Yoshio Inoue, and David Flynn, *Verification Methodology Manual for Low Power*, Synopsys, 2009.
- [4] Chris Spear and Greg Tumbush, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, 3rd edition, Springer, 2012.
- [5] Iman Sasan, *Step-by-step Functional Verification with SystemVerilog and OVM*, Hansen Brown, 2008.
- [6] Shang-Wei Tu, Tom Lin, and Archie Feng, "Power Domain Hierarchical Tree", DAC Designer Track, 2014.