Debug Challenges in Low-Power Design and Verification

Durgesh Prasad, Mentor Graphics (durgesh_prasad@mentor.com)

Madhur Bhargava, Mentor Graphics (madhur_bhargava@mentor.com)

Jitesh Bansal, Mentor Graphics (jitesh_bansal@mentor.com)

Chuck Seeley, Mentor Graphics (chuck_seeley@mentor.com)

***Abstract* - One of the toughest challenges faced in semiconductor chip design and verification is debugging. Various studies have shown that a significant amount of engineering time and effort for a project is typically spent on debug. For low-power design and verification, these debug challenges are further complicated as a result of the sophisticated power management architectures and techniques that are used. Moreover, the traditional debug technology and methods focus on issues found in a design working in always-on mode and fails to address the new and complex power-related issues thereby consuming more engineering time. In this paper, we will provide an in-depth analysis of various debug challenges and problems faced in low-power design and verification. By using relevant examples we will demonstrate how these issues can be either avoided or solved. We will also highlight some of the common pitfalls that low-power designers can avoid, which otherwise can lead to complex low-power issues that are difficult to debug at later stages of the design cycle.**

Keywords - Power Management, Power Aware Verification, Debugging Challenges in Low-Power Designs.

I. Introduction

Most engineers involved in power management design and verification would likely agree that the advent of the Unified Power Format (UPF) Standard has probably been the single biggest factor in increasing productivity when it comes to verifying the power management behavior of today's complex SoC designs. It's also very likely that this same group of engineers would agree that UPF alone has done very little to lessen the complexity of such a difficult task. Anyone who has spent any time trying to debug power management related simulation failures of a large SoC, realizes just how difficult of a job it can be.

There can be no doubt that the evolution of the UPF standard, from the original Accellera 2007 UPF 1.0 LRM, followed by the initial IEEE 1801-2009 UPF 2.0 LRM, and updated by the release of IEEE 1801-2013 UPF 2.1 LRM, has provided many new capabilities that have eased the power intent specification process as well as enabled new power management verification flows aligned with the needs of IP based SoC design today. Unfortunately the evolution of the UPF LRM alone has not lessened the complexity of the power management verification task whatsoever. Debugging power managed designs still requires a combination of the brightest and best technical resources companies have available and requires world class power management-centric debug environments to perform the task.

The difficulty of power management debug is reflected in the large number of EDA vendors that provide not only power aware front-end simulation and emulation capabilities but also by those focused on power management verification tools ranging from static analysis and rule based power checks, to power aware logic equivalency checking and layout tools. While there are debug challenges in verifying the behavior of a power managed SoC from the front-end design phase all the way through the back-end implementation phase, this paper focuses on the simulation based debug challenges that are likely to be encountered.

Power management debug has permeated all aspects of traditional HDL based simulation and nearly all major EDA vendors provide the following power specific debug features: visualization of UPF created objects, HDL signal colorization and highlighting in RTL source and wave windows, dynamic informational/status power related messages, automatic power specific assertion generation and power report generation. Below are some of the low-power debug challenges divided into various groups. While none of the debug challenges discussed in this paper will be new to those involved in power management verification, the intent of this paper is to provide a practical guide to the uninitiated based on actual user experiences.

II. Problems related to Power Intent Specification

The specification of power intent for power management of low power designs has been addressed by the UPF (Unified Power Format); however the UPF standard is still evolving with new features, concepts and clarifications being added over the releases. It often poses problems related to backward compatibility, differences and migration issues which are then difficult to debug.

A. UPF 2.0 Migration Issues

Many verification and design engineers involved in UPF 1.0 based power aware simulations have unknowingly relied on the fact that in UPF 1.0, the UPF supplies defaulted to the ON state. As a result there has been a tendency to not use the UPF package defined `supply_on` function to explicitly turn them on. As a result, many engineers might be surprised to find that a previously passing UPF 1.0 based simulation might fail after switching to UPF 2.0 based power aware simulation semantics. A common reason for many of these failures is that in UPF 2.0, the UPF supplies default to the OFF state causing all power domains to be in a CORRUPT simstate. This common migration issue can easily be avoided by using the UPF package defined `supply_on` function to explicitly set both UPF state and voltage values for all the created UPF supplies. The `supply_on` and `supply_off` function, as well as other functions defined in the UPF package, are available for use by placing the import statement in the simulation test bench, as shown below.

HDL Code

```
module tb;
import UPF::*;
…
initial begin
   supply_on ("tb/dut_inst/VDD", 1.1);
   supply_on ("tb/dut_inst/GND", 0.0);
end
…
dut dut_inst (…);
…
endmodule
```

As shown above, the `supply_on` functions are commonly placed in an initial block. About the only issue involved with the use of the UPF defined `supply_on` function is passing the proper arguments to it. The `supply_on` function takes two arguments; the first is a string type that contains the full hierarchical path to the desired UPF supply_port or supply_net that needs to be turned on while the second argument is the real type voltage value. The code example above, assumes that both the "`VDD`" and "`GND`" UPF supply_ports have been created in the UPF scope corresponding to the `dut_inst`. Another common mistake, even if the `supply_on` function is used to turn on the power supply_port, is not using a `supply_on` function to also turn on the ground supply port. Don't forget that both power and ground supplies must have their UPF state set to FULL_ON in order for a power domains simstate to be in the Normal simstate.

One other potential UPF 2.0 migration issue is due to isolation of lower boundary power domain ports. In UPF 1.0, the `set_isolation -applies_to inputs/outputs` port filters only considered power domain ports that were aligned on a module boundary. In other words only the input and output ports of modules were isolated. In UPF 2.0, these isolation port filters have been extended to also include the lower boundary (child module instances in different power domains) input and output power domain ports as well. The concept of lower boundary power domain port isolation can potentially cause simulation failures as a result of unintended back-to-back isolation cells inferred by tools, especially if isolation strategies were included for any lower-level power domains.

B. UPF2.0 list/wildcard expansion issues

Another debug challenge in power intent specification arises because of the usage of list/wildcard expansion in various UPF commands. It often happens that an incorrect list of signals is created as a side effect of usage of the wrong pattern in wildcards. This problem can be avoided if the user relies on the UPF command "`find_objects`" to create a list of signals and uses the tcl command "`puts`" to print the contents of the element list. Another way to get the list of expanded signals is to use the "`save_upf`" command. Consider the following design example:

HDL Code

```
module dut;
…
Ip_module my_ip1();
Ip_module my_ip2();
Ip_module my_ip3();
endmodule
```

In this particular case `my_ip1` and `my_ip2` have same power requirements; however `my_ip3` has some different power requirements. The intended UPF usage is to add `my_ip1` and `my_ip2` to one power domain, while creating another domain for `my_ip3`.

The following incorrect and Non-LRM usage of a wildcard can result in the addition of all three instances of `Ip_module` to same power domain.

UPF Code

```
create_power_domain pd_dut -elements {my_ip*}
```

Though correction of such an issue requires a change in the UPF itself, however the first and foremost problem is to know what all elements have been added to `pd_dut`. In certain cases, some verification tool may explicitly list out the expanded elements; however that is tool dependent and may not be supported by all the tools.

One of the ways to know which elements are being added to power domain `pd_dut` is to use the "`save_upf`" command in the UPF file. With this UPF command, the verification tool will dump out interpreted commands in a new UPF file. This UPF file will contain lists of expanded elements.

Another way could be to avoid using Non-LRM wildcard usage and rely on UPF command `find_objects`. As shown in the below UPF code, the user can get the list of expanded elements in a tcl variable and print out that variable to know the expanded list.

UPF Code

```
set my_element_list [find_objects . -pattern {my_ip*}]
puts "Elements added to pd_dut : $my_element_list"
create_power_domain pd_dut -elements $my_element_list
```

Stdio output as a result of processing above commands:

```
Elements added to pd_dut: {my_ip1 my_ip2 my_ip3}
```

C. Incorrect vct specified

In case of a macro model in the design, the power supplies along with the power aware functionality are present inside the model itself. When integrating this macro model in a SOC, the integrator has to connect these HDL supplies to the UPF nets. Since the UPF supplies are of type supply_net_type having state and voltage values and the supplies defined in HDL are of wire type, the connection between UPF and HDL net requires a VCT (value conversion table). The VCT defines the mapping between the state of UPF net and value of HDL port/net. The user can either rely on verification tools to apply a default VCT or explicitly specify which VCT to be used using the UPF command `connect_supply_net -vct`. The problem arises when the same VCT gets used for power/ground/pwell/nwell supply nets. This causes power up failure because ground/nwell supply nets are active low and expect a ground specific vct. Consider the design example:

HDL Code: Power Model

```
module macro_model(in1, in2, out);
…
wire VDD = 1;
wire GND = 0;
if (VDD && !GND)
    out = in1 & in2;
else
    out = 'x';
endmodule
```

UPF Connections

```
connect_supply_net upf_VDD -ports {hm_inst/VDD} -vct UPF2SV_LOGIC
connect_supply_net upf_GND -ports {hm_inst/GND} -vct UPF2SV_LOGIC
```

Here "UPF2SV_LOGIC" is a predefined vct by UPF to convert UPF supply_net value to SV logic value as per following rules:

```
create_upf2hdl_vct UPF2SV_LOGIC
    -hdl_type sv
    -table {{UNDETERMINED X}
           {PARTIAL_ON X}
           {FULL_ON 1}
           {OFF 0}}
```

Since wrong VCT(UPF2SV_LOGIC) is getting applied on ground net (GND), the value driven on HDL net GND is "1" when the UPF supply (upf_GND) is FULL_ON. In order to correct this issue, the user needs to specify the ground specific VCT.

UPF Connection

```
connect_supply_net upf_GND -ports {hm_inst/GND} -vct UPF_GNDZERO2SV_LOGIC
```

If the user is relying on verification tools to apply the VCT, then tools needs to be guided that the particular supply is a "ground" supply. This information can come from the liberty file attribute "`pg_type`" or it can be specified using the UPF command "`set_port_attributes`" as follows:

pg_type example (Liberty Specification):

```
pg_pin (GND) {
    pg_type: primary_ground
}
```

set_port_attribute (UPF Specification):

```
set_port_attributes -pg_type primary_ground -ports {hm_inst/GND}
```

III. Design power up failures

A common debug scenario that commonly occurs is when a low-power design fails to power up after a power down period. The problem can be because of any of the following reasons.

A. Missing/incorrect isolation/level shifter

Today's SoCs have large numbers of power domains where each power domain is interacting with other power domains. Two interacting power domains may also be operating with different voltage ranges. The Voltage representing logic value '1' in driving domain may represent it as '0' in receiving logic. Level-shifters are inserted at a domain boundary to resolve such issues; they translate the logic values to proper voltage ranges. The translation ensures the logic value sent by the driving logic in one domain is correctly received by the receiving logic in the other domain. Interacting domains may also cause floating problems. If the driving logic is powered down, the input to the receiving logic may float between 1 or 0. An un-driven input can also cause functional problems if it floats to an unintended logic value. To avoid this problem, isolation cells are inserted at the boundary of a power domain.

In such scenarios, it becomes necessary to have isolation and level shifter cells at domain boundaries for the proper functioning of the design. Any missing cell can cause lot of functional issues, which would be difficult to debug. The following techniques are used to debug such scenarios.

1) Static Verification at compile time

Many tools statically determine the need for isolation and level shifter cells at domain boundaries from the PSTs and power states described in UPF. The PSTs describe the valid interacting states between two domains. These states clearly define the voltage ranges in which two domains are interacting and also whether one domain is relatively ON to other domain or not. If a signal is going from a domain of low voltage range to high voltage range then there is a need of Level-Shifter with the "`low_to_high`" rule. Similarly there is a need for the "`high_to_low`" rule for signals going from high-voltage to low-voltage. It is an error scenario if no level shifter strategy is specified or a level shifter strategy with a different rule is specified.

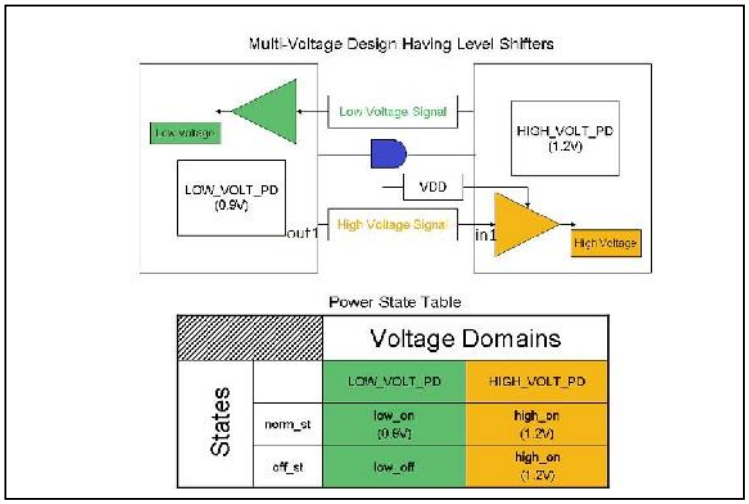

Figure1: Design Example

In the above example, two level shifter strategies are required, one with the rule "`low_to_high`" for the signal going from `LOW_VOLT_PD` to `HIGH_VOLT_PD` and other with the rule "`high_to_low`" for the signal going from `HIGH_VOLT_PD` to `LOW_VOLT_PD`. Also `HIGH_VOLT_PD` is relatively ON to `LOW_VOLT_PD`, so there is a need of an isolation cell.

UPF Snippet :

```
add_pst_state norm_st -pst PST -state {low_on  high_on}
add_pst_state off_st -pst PST -state {low_off high_on}
```

The tool will issue the following type of message :

Missing Isolation Cell :

```
    Source power domain : LOW_VOLT_PD -> Sink power domain: HIGH_VOLT_PD
            1. Source port: out1 -> Sink port: in1
```

2) Tool generated assertions

System Verilog Assertions (SVAs) are a very powerful way to achieve the dynamic verification of low power designs. They can be used to validate power control logic sequences and also ensure that specific requirements are met before and after power mode transitions. Many of the EDA vendors provide automated, tool-generated assertions to check for missing or incorrect Isolation and Level shifter at run time. For each interface signal at the domain boundary an assertion is inserted that would check the need for Isolation and Level Shifter Cells.

The following types of errors are reported at run time:

```
    ** Error: UPF_MISSING_LS_CHK: Time: 90 ns, Missing level shifters for domain boundary,
LOW_VOLT_PD ( Operating Voltage: 5.000000 V ) => HIGH_VOLT_PD ( Operating Voltage: 5.200000 V )
for Source port : out1 -> Sink port: in1
     File: ./src/test.upf, Line: 32, Power Domain: LOW_VOLT_PD
```

3) UPF Bind Checker

UPF also provide a way to add custom assertions using the bind_checker command. The user can write their own assertions in a module and then bind that module to boundary instances to check whether a signal is isolated or not.

Checker module:

```
module checker_isolation(input op, src_supply, sink_supply) ;
    always@(src_supply)
        assert ( (src_supply == ON) || (sink_supply == OFF) || (op != 'X') )
else $error("Missing Isolation");
endmodule
```

The above checker module checks the value of a signal when Source Domain goes OFF and Sink Domain is ON. If a signal gets corrupted then it is case of Missing Isolation Cell.

B. Initial block re-evaluation

Before the advent of power aware RTL simulation capability, the use of initial blocks in RTL code for register and memory value initialization in digital RTL simulators caused very little simulation related issues. Initial blocks were well understood by both simulation providers and end-users. These blocks needed to execute once, at time 0, and never needed to be re-triggered again during a simulation. Designers inherently knew initial blocks were not synthesizable and that any registers needing to be initialized had to be reset either synchronously or asynchronously resettable in an always block. The only potential simulation issues caused by the use of initial blocks prior to power aware simulations were race conditions. These race conditions typically occurred as a result of an evaluation dependency order between two or more initial blocks when a register value, in one initial block, was initialized to the value of another register initialized in a separate initial block.

In power aware simulations reliance on initial blocks for initialization of register and memory values can be problematic since signals may need to be re-initialized after time 0 as a result of their values getting corrupted as a result of their power domain being powered off and back on. In addition to the re-initialization issue, there is also an inherent time 0 race condition between evaluation of the state of a power domain turning on at the start of a simulation and execution of any initial blocks contained in the RTL code which are in the extent (the set of instances) of the power domain.

This re-initialization requirement is best illustrated by a ROM memory, which in real hardware, has its contents pre-programmed and available on demand whenever power is applied to the system. In RTL simulations a ROM is typically modeled using a 2-D register array with its contents initialized by placing either a `$readmemh` or `$readmemb` system task inside an initial block. The initial block commonly contains a "`for loop`" to load the respective hex or binary values, from the specified external file, into the desired 2-D register array locations. There are two ways to mimic the ROM's hardware behavior in a power aware simulation. The first is to use a retention memory for the ROM while the second option is to simply make the simulator re-evaluate the ROM's initial block every time the power domain is powered back up. Cleary from a simulation user's perspective, pushing the simulator to handle re-evaluation of an initial block at power up is the cleanest solution to this problem since the ROM model likely contains the initial block anyway. UPF LRM doesn't provide any command to specify the re-evaluation of initial blocks, but almost all the EDA vendors provide some non-LRM way to specify initial blocks for re-evaluation.

Another way to handle signals placed in initial blocks in a power aware simulation is to exclude them from power management altogether. Consider the RTL code below where the clk register is not resettable but rather initialized to a 1'b0 in the initial block and toggled by the always block every 10ns.

```
module dut (… );
…
reg clk;
…
initial clk = 1'b0;

always #10 clk = ~clk;
…
endmodule
```

In a non-power aware simulation the clk toggles because the clk register is first initialized to a known value before the always block is executed. In a power aware simulation after a power down sequence the 'clk' would get a value 'x' and will keep that value forever since the initial block will not be triggered by default (under strict LRM based simulation). In those cases, by excluding the signal from active power management would allow the clk register initialization to occur and be toggled by the always block.

C. Retention is not/incorrectly implemented

Retention is the enhanced functionality associated with selected sequential elements or a memory such that memory values can be preserved during the power-down state of the primary supplies. The correct protocol for retention is to save the value of a state element prior to switching off the power of that element, and then restoring the saved value after power-up of that element. UPF provides a command `set_retention` to specify the state elements of a power domain which needs to be retained during power down. The command is also used to specify the control signal/conditions that determine when to save and restore the sequential element values.

In certain cases retention may not be implemented or may be applied incorrectly causing the design to fail.

1) Retention is not specified

Certain critical registers in the design need to be retained so that at the power up they start with a previously saved state. However, if a retention register is not applied on them, then these registers would not be able to come out of 'x' state. To understand the retention behavior, consider the example of a simple counter:

HDL Code - Counter

```
always@(posedge cnt_rst,posedge clk)
begin
    if(cnt_rst)
            cnt <= 16'b0;
    else
            cnt <= cnt + 1;
end
```

At the power shut down, the 'cnt' will get corrupted. If retention is not specified on '`cnt`' then even after power up, '`cnt`' would remain 'x' and never get a proper value. Below is the simulation result of 'cnt' signal.

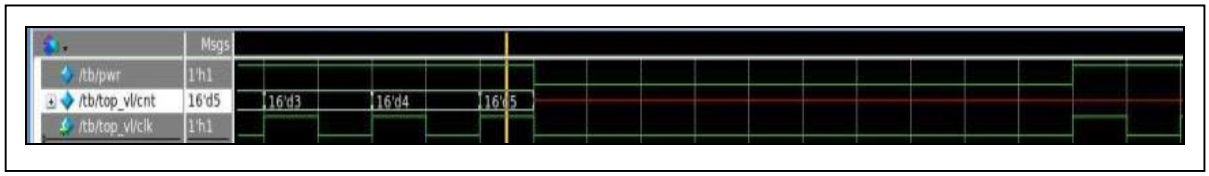

Figure2: Simulation behavior of "cnt" signal without retention

To avoid such an issue, retention needs to be applied on 'cnt' signal. Modifying the UPF as follows:

```
set_retention … -elements {top_vl/cnt}
set_retention_control –save_signal {ret posedge } –restore_signal {ret negedge}
```

The value of 'cnt' is saved during posedge of 'ret' signal and it gets restored during negedge of it.

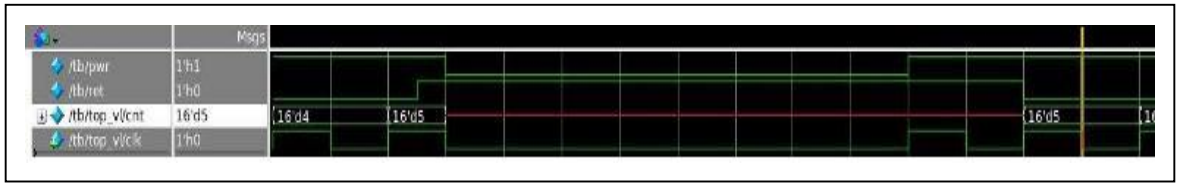

Figure3: Simulation behavior of "cnt" signal with retention

2) Retention protocol is incorrect

Depending upon the type of retention register, "Ballon-style retention" [1] or "Master/salve-alive retention" [1], the save and restore activity can be specified using save/restore control signals or using restore conditions respectively. Typically the proper sequencing of retention and domain power is as follows:

```
Retention save -> power down -> power up -> retention restore.
```

Incorrect retention sequencing can cause an incorrect or 'x' value on the register at power up. Nowadays EDA vendors provide dynamic checks to assure that proper retention protocol is being followed. For example, the following is an incorrect retention sequence:

```
Power down -> retention save -> power up -> retention restore.
```

In such a scenario, verification tools can give an error message.

```
    # ** Error: (vsim-8903) MSPA_RET_OFF_PSO: Time: 64 ns, Retention control (0) for the
following retention elements in scope '/tb/top_vl' of power domain 'pd' is not asserted during
power shut down:
    # cnt.
    # File: test.upf, Line: 41, Power Domain:pd
```

IV. Unwanted X on some signals

An unwanted X on some design signals is the most common debug problem in power aware simulation. Debugging of such problems need some advanced features from verification tools. The most useful features in this regard are as follows.

A. Debugging 'X' values on signal

A major debug issue in low power simulations is root-causing unknown (X's) values on various signals. There can be many reasons why X values appear on signals in low power simulations ranging from incorrect UPF specification (missing isolation/level-shifting retention cells or improper power domain partitioning) to UPF 1.0 to UPF 2.0 simulation sematic differences to use of initial blocks (re-evaluation and races between signal initialization and domain power up). In an effort to help distinguish a normal unknown value on a signal, most simulation tools have the ability to highlight unknown values in wave windows caused by power domain corruption. In normal simulations, unknown X signal values are typically displayed using either a single mid-high red line or a red outlined box around the unknown value region in a wave window. In low power simulations the entire low-high region is filled in using red colored coarse or fine-grain cross-hatching to indicate the X value is a result of direct power domain corruption as shown in the figure below.

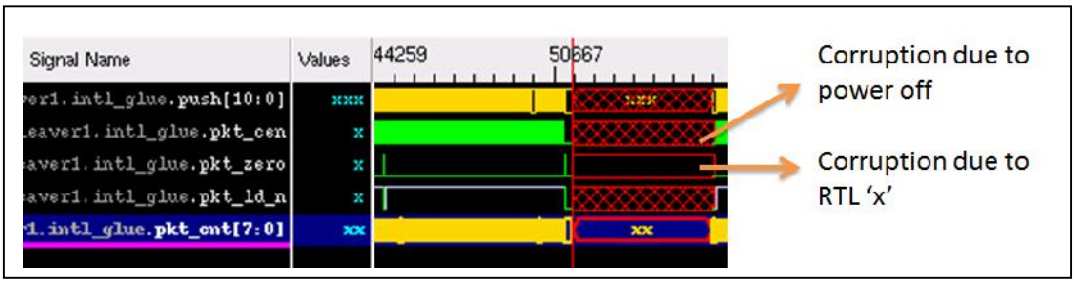

Figure4: Unwanted 'x' waveforms

Typically the corruption highlighting is only visible on the direct output of the driving logic and is not displayed on the outputs of subsequent logic the corrupted net fans out to. Even when an unknown X value signal is not highlighted in the wave, the ability to trace the net connection backwards to where a highlighted signal is visible can greatly simplify the task of determining the origin of the X value.

Most of the EDA vendors provide wave compare tool to identify root cause of these unwanted X values. Consider the code previously shown in the initial block revaluation section where 'X' value on the clk signal is because of power domain getting OFF. In this scenario, the clk can't toggle since the initial value is X instead of a 1'b0. If all the necessary power domains are functional, corruption highlighting won't be of much use in catching the X on the clk. However, wave compare between a normal and low power simulation could easily catch the X on the clk. Another area where wave compare is effective is right after power up a domain especially if retained register values need to be restored for full functionality. It is pretty easy to see differences in signals which remain X as a result of a missing retention element.

If wave compare is not available, another useful technique for catching unwanted X values is correlating their occurrence with changes in power domain sim-states or power states, power control signals including those for power switches, isolation enables, and retention save/restore signals. Most power aware simulation vendors provide the ability to print low power informational related messages. These messages typically are about the change of state in supply nets/ports, power switches, various power control signals, and power domains.

```
Note: UPF_SWITCH_CTRL_INFO: Time: 794845 ns, Power Switch (glue_sw), Control Signal (sw_en),
switched to polarity (1), Power Switch state (FULL_ON)
```

In addition to informational messages many low power simulation vendors also provide the ability to create assertions to help in detecting sources of X values due to issues related to power control sequencing. For instance, turning off the retention supply to a power domain while it's OFF will corrupt the retained register values and cause an X values to be restored to the retention elements after power is restored.

```
Error: UPF_PG_CHK:784845 ns, Power for Retention strategy: 'p_ret' of power domain:
PD_gluelogic' is switched OFF during retention.
```

Besides low power assertion checks to catch possible sources of X values during simulations, some vendors also provide a static analysis capability, when processing the UPF file, to determine if there are any missing, redundant, or invalid isolation/level shifters which can help catch them as well. An un-isolated power domain port will obviously propagate an X value to other power domains when it's powered down.

B. Trace the driver of the signal

Typically in a non-power aware simulation the driver of a signal is some RTL logic. However in the case of a power-aware simulation, it could be anything ranging from RTL logic to UPF inserted cells. For the cases of an unexpected value on a signal, it might be the effect of power-aware activity on that signal or it could be a propagated effect of power-aware activity on some of its driver signal/logic. In these cases the driver-tracing is a useful way to find out the driver signal and its value.

Here is an example from Questa for finding out the driver signal/logic of '`/tb/out2`' which is an assign logic within the hierarchy '`/tb/TOP`'.

Tool Information – drivers /tb/out2

```
# drivers /tb/out2
# Drivers for /tb/out2:
#    StX  : Net /tb/out2
#      StX : Driver /tb/TOP/#ASSIGN#61
```

It is evident from the drivers information that 'x' on out2 is because of an assign statement going to 'x' (preferably due to power aware corruption).

Another useful feature is dataflow/schematic debugging of a signal. It helps in tracing the value of a signal which is being driven from a distant logic. In power-aware debugging it is even more helpful as it also displays the UPF inserted cells in the dataflow path. Here is an example of dataflow from Questa showing an isolation cell and a level shifter encountered in the dataflow path of signal '`/tb/out1`':

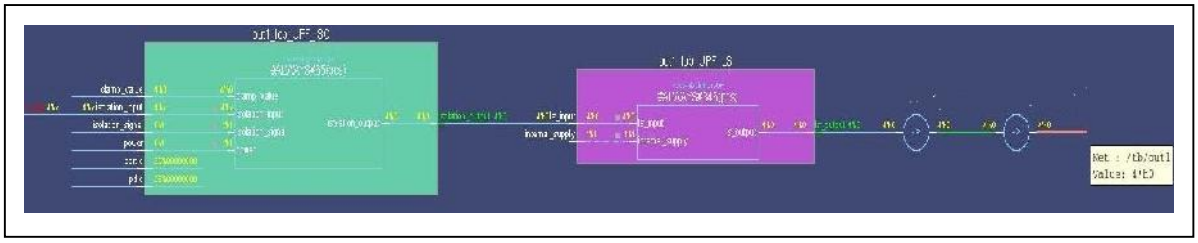

Figure 5: Dataflow of signal /tb/out1

V. Corruption is not showing on some signals

One very common low-power debug issue is that certain part of design fails to switch off and the logic inside that part is never corrupted, although the user expects it to be switched off and show corrupted values. There could be multiple reasons for such behavior, some of which are as follows:

1) Incorrect power domain specification

The design element under concern has been put in a power domain that is not switchable. The first step for debugging such an issue is to identify the power domain to which this region belongs. This can be done either by looking at the power aware textual reports generated by the tool or using the tool's GUI capabilities. The next debugging step is to determine if the power of that domain is being switched off or not. It can be verified using the dynamic messages reported by the tool for whenever a power domain changes its state:

```
# ** Note: (vsim-8902) MSPA_PD_STATUS_INFO: Time: 64 ns, Power domain 'pd' is powered down.
```

2) Exclusion of behavioral model

EDA vendors provide mechanisms to skip the corruption on certain portion of the design. Some of the commonly used methods are by specifying the elements as DONT_TOUCH elements via UPF itself or use a separate file to exclude them from power-aware behavior. In such cases it is advised to look for the tool generated reports to find out if the design element has been excluded by the tool or not.

3) Simulation semantic disabled

Disabling simulation semantics of a design element means the tool will not impart any power shut-off/corruption on that element. The simulation semantics of a region can get disabled because of following reasons:

a) The design element is already a power-aware (All-pins model) [2] and it has its own power pins which are connected via UPF.

b) The design element is an 'always_on' cell as per its liberty specification.

c) The design element is a power-aware cell (isolation/level shifter) as per its liberty/HDL definition however it could not be mapped to any UPF strategy.

In all these cases the tool disables the power aware simulation semantic and hence there would be no tool injected corruption. All such regions can be easily identified by looking at the message thrown by the verification tool:

** Note: (vopt-9693) Power Aware simulation semantics disabled for chip_top/u_hm_top_0/u_ip_1

VI. System transitioning to illegal power state

One of the major debug tasks for any low power design is verification of the design's operational power states. This task requires verifying that each defined power state of every power domain has been covered and functioning properly. It also requires verification of all power state combinations across all domains that comprise each operational power state. Obviously the difficulty of this task will only get harder as designs continue to increase in both the number of power domains and operational power states they support. While the Accellera UPF 1.0 standard supported the creation of power states using the three Power State Table (PST) commands, `add_port_state`, `create_pst`, and `add_pst_state`, there were several limitations that were not addressed in the standard which are:

a) No support for bias states. Only supports the basic OFF (corrupt) and ON (normal) power states

b) No ability to merge multiple PST's together

c) No support for hierarchical power state definition/creation

d) No ability to update power state information

e) No ability to distinguish between legal and illegal power states or detect transitions to and from legal and illegal power states.

Due to the many limitations inherent with UPF 1.0 power states, most low power simulation vendors extended their Finite State Machine (FSM) coverage capabilities to include both power states and power states transitions to do the job.

The IEEE UPF 2.0 standard addressed all these limitations by providing the `add_power_state` and `describe_state_transition` commands. Not only does `add_power_state` support bias states, hierarchical power state creation, and an incremental update capability, it also allows any named power state to be declared as legal or illegal. Likewise, the `describe_state_transition` command allows any transition between two power states to be declared legal or illegal. Using these two commands require low power simulation vendors to issue run-time error messages whenever an illegal power state is reached or any illegal power state transition occurs.

UPF Code

```
add_power_state PD_ALU_SS -state ON4 { -logic_expr { !pwr_alu && !pwr_ram } -simstate CORRUPT
-illegal}
```

Simulation message

```
# ** Error: (vsim-8933) MSPA_UPF_ILLEGAL_STATE_REACHED: Time: 129 ns, Supply set 'PD_ALU_SS'
reached an illegal power state 'ON4'.
# File: src/parser_test22/demo.upf, Line: 73, Power state:ON4
```

The UPF 2.0 standard also stipulates that an unnamed or undefined power state is also illegal.

The detection of undefined power states is especially useful if an unintended power state occurs when transitioning from one defined state to another. The occurrence of an undefined power state during a legal power state transition may result due to a race between changing a UPF supply via the UPF package `supply_on/supply_off` functions and switching of a power control logic signal at the same time. A race induced

undefined power state likely indicates an area where voltage ramp up/down times versus logic switch times must be accounted for in order to ensure proper operation of the design.

VII. Conclusion

Debugging the low-power design is a big challenge in itself. In this paper we have highlighted various scenarios and challenges faced in debugging of a low-power design so that low-power issues can also be detected early in the design phase. With the help of relevant examples, we have also demonstrated how such low-power issues can be either avoided or fixed thereby significantly increasing the productivity of the debug process.

VIII. References

[1] IEEE Std 1801™-2013 for Design and Verification of Low Power Integrated Circuits. IEEE Computer Society, 29 May 2013

[2] Power Aware Models: Overcoming barriers in Power Aware Simulation (Mohit Jain, Amit Singh, J.S.S.S.Bharath, Amit Srivastava, Bharti Jain), DVCon-Europe 2014.

[3] Static and Formal Verification of Power Aware Designs at the RTL Using UPF (Rudra Mukherjee, Amit Srivastava and Stephen Bailey), DVCon 2008.