Low-Power Verification Methodology using UPF Query functions and Bind checkers

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

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

***Abstract*—The increasing demand to minimize power dissipation and offer energy-efficient chips has resulted in the use of highly-advanced power management techniques. Verification of these complex techniques is a challenging task and if not executed efficiently it can waste a lot of design verification cycles. One very powerful way to use low-power verification is with assertions, which can be used to validate power control logic sequences and system-level power management, and also ensure that specific requirements are met before and after power mode transitions. However, modeling these low-power assertions in HDL in conjunction with their power intent is a very complex task, as any change in power intent can completely break the assertions. This leads to a need for an automated way of writing these assertions with considerable immunity from any change in the power intent.**

In this paper, we will highlight the challenges in assertion-based low-power verification and propose a methodology that leverages the capabilities of UPF (Unified Power Format) to specify the power intent and apply efficient and automated low-power assertions and properties. By presenting various examples we will demonstrate how you can use UPF to query the power objects from your power architecture and pass on these objects along with HDL objects to a checker module that can be instantiated in the design without affecting the actual design. We will also propose some of the extensions required in existing UPF commands to use this methodology. We will highlight that with the suggested flow, the verification process can become more automated and the closure can be achieved in less time.

***Keywords*— Power Management, Power Aware Verification, UPF, Assertion Based Verification.**

I. Introduction

A. *Power Aware Designs*

As electronic systems are getting complex, power and heat dissipation are becoming ever more important. There is an increasing demand to reduce the power dissipation and offer energy-efficient chips. This has resulted in the use of highly sophisticated power management architectures. Designers are now employing aggressive power management strategies ranging from clock gating and power gating to multiple voltages and dynamic scaling of voltages and frequencies. It is extremely important to verify the power management because if it is not applied properly, then these strategies can affect the functionality of the design. As the power management considerations start early in the design cycle, catching power management architecture bugs at early stages can save a lot of design verification cycles.

Creating a power aware design involves partitioning of the design into a set of regions or power domains such that each of the power domains can have its own power supplies managed independently. The power architecture is modeled using elements like isolation, retention, level-shifter cells, supply nets, ports, etc. This power architecture is controlled with the help of a power management unit that issues proper control sequences to different elements of the power architecture. At any given time, a power domain may operate at different voltage level or state and it can interact with other power domains.

The increasing need for low-power verification has led to methods for specifying the power intent, i.e., defining the power management architecture. Power intent specification formats were introduced to address this. These formats allow the user to express the power management which can be overlaid on top of HDL. The power intent can be specified much earlier in the design cycle without any modification in the normal design functionality. This specification can be used by various tools to perform verification and implementation of power managed designs.

B. *Power Intent Specification and Basic Concepts of UPF*

IEEE Std 1801™-2013 Unified Power Format (UPF) allows designers to specify the power intent of the design. It is based on Tcl and provides concepts and commands which are necessary to describe the power management requirements for IPs or complete SoCs. A power intent specification in UPF is used throughout the design flow; however it may be refined at various steps in the design cycle.

Some of the important concepts and terminology used in power intent specification are the following:

- Power domain: A collection of HDL module instances and/or library cells that are treated as a group for power management purposes. The instances of a power domain typically, but do not always, share a primary supply set and typically are all in the same power state at a given time. This group of instances is referred to as the extent of a power domain.
- Power state: The state of a supply net, supply port, supply set, or power domain. It is an abstract representation of the voltage and current characteristics of a power supply, and also an abstract representation of the operating mode of the elements of a power domain or of a module instance (e.g., on, off, sleep).
- Isolation Cell: An instance that passes logic values during normal mode operation and clamps its output to some specified logic value when a control signal is asserted. It is required when the driving logic supply is switched off while the receiving logic supply is still on.
- Level Shifter: An instance that translates signal values from an input voltage swing to a different output voltage swing.
- Retention: 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.
- Repeaters: If the distance between driver and receiver is long, special buffers may be required to boost the strength of the signal and to ensure that it stabilizes within the required time. These buffers are typically called repeaters.
- Supply net: an abstraction of a power rail.

II. LOW-POWER VERIFICATION

Verification of power management techniques is a challenging task and if not executed properly it can waste a lot of design verification cycles. Low power verification is generally performed in two steps. Firstly static verification is done to catch all structural errors which include detection of correct placement and connection of power management objects like isolation, level shifter, repeaters and retention. It is followed by dynamic verification where design along with power management architecture is simulated with both functional and power control inputs to detect all control sequences and protocol related errors. This paper focuses on dynamic verification of low-power designs.

A. *Low-power dynamic verification items*

Some of the common dynamic verification items in low-power designs are as follows:

- Protocol checking: In a low-power design, various power management cells like isolation cells, level shifter cells, and retention cells may be inserted into the design affecting its functionality. Therefore it is of high important to check that they are enabled and active at the proper time. One of the protocol checks for isolation is to verify that Isolation enable is triggered before the power of the source logic goes down; it remains active throughout the power down period and until sometime after the power goes up. Similarly if the power domain exhibits retention capability, then another protocol check is to verify that its control signals are triggered at the right time i.e. retention SAVE is triggered before power down and RESTORE is triggered after power up.
- Power Intent checking: Another aspect of low-power verification is related to verifying UPF intent against the implemented power aware design. For example user wants to ensure that during the

active isolation period, the signal on which isolation is applied is being clamped to the correct clamp value as specified in UPF file.

- Power Intent coverage: One of key area of concerns for verification engineer is to ensure that all system states are tested and covered. This can be further achieved by meeting coverage goals of its constituent objects like power domain, supply sets, supply nets, supply ports. For example the coverage of a power domain is defined based on coverage of its strategies, own state and states of its supplies.

There could be many other verification items however those are out of scope of this paper.

B. *Assertion based low-power verification*

One of very powerful way to achieve the dynamic verification of low-power designs is with SystemVerilog assertions (SVAs), which can be used to validate power control logic sequence and also ensure that specific requirements are met before and after power mode transitions. These assertions can monitor and check transitions in the power control signals to verify legal and correct low-power behavior. The assertions also provide coverage data that can be used for verification closure.

Some EDA vendors do provide automated, tool-generated assertions to check common protocol errors. However, in cases where these are not complete, the user has to rely on his own custom low-power assertions to verify the power intent of the design and also make sure that all test scenarios are covered.

Modeling these low-power assertions in HDL in conjunction with their power intent is a very complex task, as any change in power intent can completely break the assertions. This leads to a need for a methodology of writing these assertions such that these assertions remain unaffected from any change in the power intent.

III. Motivation for Methodology

Tool-generated assertions are used widely for low-power verification. However they may not be useful or exhaustive in all the designs, as highlighted by the reasons below:

- A design can have a very specific requirement which is not being provided by the tool-generated assertion.
- The low-power technology is still evolving and hence a new set of protocol appears every now and then, which may require a different set of assertions that is not yet provided by the tool vendor.

Due to the above reasons the user may want to write his custom assertions and coverage items. These items can be grouped in a checker module, and this checker module can be instantiated into the design using UPF command “`bind_checker`”. However the method of instantiation of such a checker module is not trivial because:

- These low-power assertions/coverage items require access to power objects. However, at the early stages of verification these power objects are only present in the UPF file and do not exist in the design. It is therefore not straightforward to pass these UPF objects to a checker instance.
- Some of the property-checking requires access to design/power signals spanning across multiple domains. Such a task is highly error prone and time consuming.
- As the scope and the inputs of these checkers instances are defined in UPF, any change in the UPF or design might break these checkers and they need to be re-written.

The above problems can be averted by use of another set of UPF concepts and commands (query functions) to extract the power object handles and pass on these dynamic UPF objects to `bind_checker`.

The motivation behind this paper is to show a methodological use of `bind_checker` and `query_*` functions to overcome the problem of an evolving UPF and changing design and hence provide a strong way of writing custom assertions and coverage bins.

IV. METHODOLOGY

A. *UPF concepts/commands*

The following UPF commands are the basic building block of our methodology:

1) `bind_checker` command

<u>The sample code snippet:</u>

```
bind_checker checker_instance_name \
    -module checker_module_name \
    -bind_to target_instance
    -ports {{formal_port1_name power_object_handle} \
           {formal_port2_name power_control_signal}}
```

The UPF 2.0/2.1 standard provides the `bind_checker` command that helps to instantiate the checker module '*checker_module_name*' into the design hierarchy '*target_instance*' with the instance name '*checker_instance_name*' without actually modifying the design code or introducing any functional changes. The low-power assertions are modeled in this checker module, which can then monitor the power aware design.

The method of binding this checker module to the design elements relies on the SystemVerilog bind directive. The UPF command `bind_checker` allows specifying the target instance where this checker module needs to be instantiated. It also provides one-to-one port mapping of the checker module to the actual argument, which in this case can be a power object/control signal. Thus the checker module gets access to any UPF object in the HDL scope.

2) `query_*` commands

The UPF 2.0/2.1 standard provides a great toolset of commands, including query (for example `query_power_domain, query_isolation, query_retention`), which can be used to search and get the handle of power management objects, including strategies (isolation/retention/power switch), power domains, supply nets, supply ports etc. These commands follow a hierarchical approach and return the handle of objects which reside in or below the scope where these commands are called.

<u>The sample code snippet:</u>

```
#------------------
# Get list/handle of all power domains defined in this scope or below
#------------------
query_power_domain *

#------------------
# Get handle of isolation strategy 'PD_ISO1' defined in domain 'PD'
#------------------
query_isolation PD_ISO1 -domain PD
```

The return value of `query_isolation` command can be used to get isolation strategy details e.g. isolation enable signal, its elements, isolation power etc.

<u>Note:</u> The purpose of the `query_*` command in the methodology is just to extract out the handles of power/control signals from the power architecture. However any other way apart from `query_*` commands can also be used in the methodology to extract out the same information. In fact UPF 2.0 query function definitions were somewhat ambiguous and they have been moved to an appendix in UPF 2.1. The P1801 working group is working on an information model and API that will serve as the basis for a new set of query functions in UPF 3.0.

3) `find_objects` command

UPF 2.0/2.1 standard provides this command to query the design (HDL) elements. It provides a good deal of filtering support to extract relevant elements.

<u>The sample code snippet:</u>

```
#------------------
# Get all output ports of the instance 'inst1'
#------------------
```

```
find_objects inst1 -pattern * -object_type port -direction out
```

B. *Steps required for methodology*

The recommended methodology revolves around the above mentioned UPF commands. The assertions and covergroup needed for low-power verification are packed in a checker module. This module is then instantiated in the relevant design scope using the UPF command `bind_checker`. For low-power verification, the SVAs and covergroups require access to both the power objects and the control signals. Since these SVAs reside inside the checker module, the required actual design/power signals need to be passed as actual arguments to the checker module. This is achieved by using query functions and bind checkers.

During low-power simulation of the design, whenever these assertions fail it indicates a functional issue or a low-power bug in the design. The coverage bins instantiated within these checkers module collect the coverage items and help achieve verification closure. The methodology interface is Tcl-based therefore you can embed them in the UPF file itself to automate the verification process.

Specifically the methodology proposes following steps:

- Model the protocols/power intent to be checked by a set of SystemVerilog assertions and club them together in checker module.
- Define an interface based on UPF commands `query_*, find_objects, bind_checker` as mentioned below :
  - Determine the required power objects and design signals which are required by the above SVAs. Extract the power/control signals (UPF objects) from the power architecture using the UPF `query_*` commands. If required, extract the design signals (HDL objects) using the UPF command `find_objects`.
  - Pass the above handles to the checker module and instantiate it in the design with the help of the UPF command `bind_checker`.

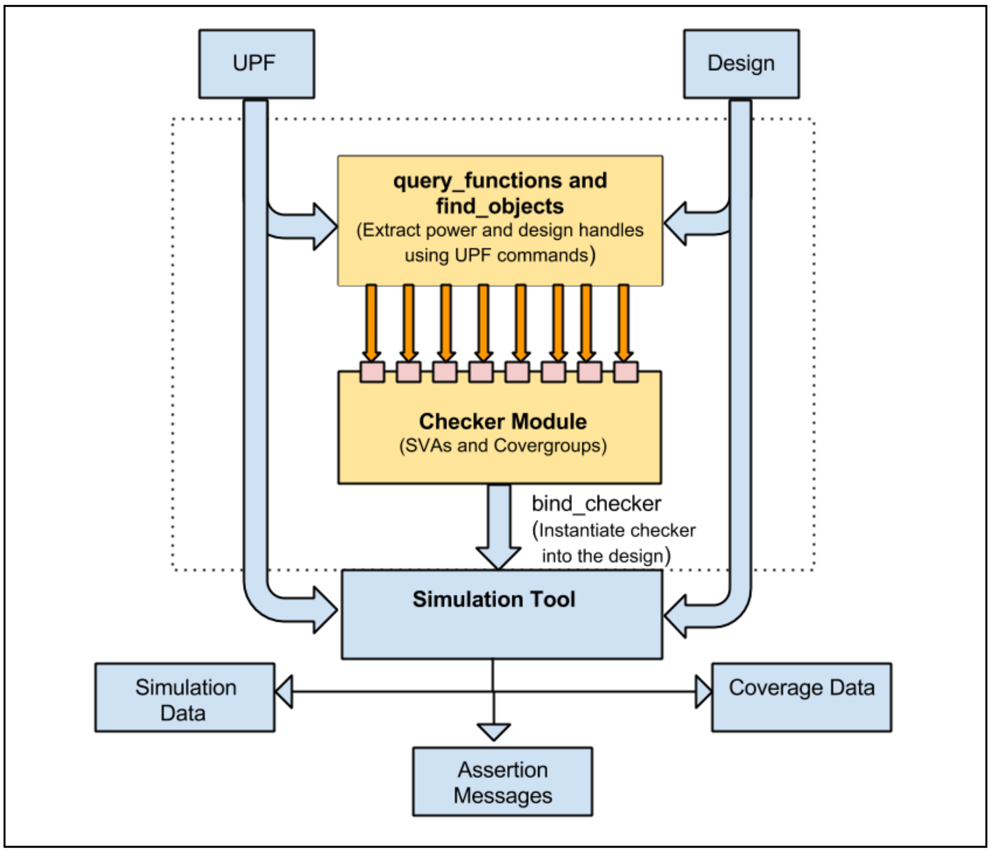
UPF
Design
query_functions and find_objects
(Extract power and design handles using UPF commands)
Checker Module
(SVAs and Covergroups)
bind_checker
(Instantiate checker into the design)
Simulation Tool
Simulation Data
Coverage Data
Assertion Messages

Figure1. Methodology Flow Chart

C. *Extensions required in the UPF concepts/commands*

The methodology relies on UPF concepts and commands to extract out the information from power management and attach it to a checker module. Though the UPF provides a great toolset to support this, there a few extensions required in these commands in order make full use of methodology potential.

1) query_* commands

- As the `bind_checker` can get instantiated anywhere in the design, the handle of power/control signals needs to be the full hierarchical path. All `query_*` commands return the object handle as a name of the object referenced from the active scope. To use `query_*` commands as recommended in the methodology, these commands needs to be extended to return the object handle as the full hierarchical path of the object referenced from the design top.
- Certain `query_*` commands need to be extended to provide additional information which is not as defined, per the UPF LRM. For example, `query_power_domain` needs to be extended to provide the primary supply (power/ground) of the queried power domain.
- The `query_* -elements` command needs to be extended to provide processed information. For example, it is not possible to extract list of isolated signals when using `-source/sink` option in `set_isolation` command.

2) `bind_checker` command

- Certain SystemVerilog assertions and covergroups use the object name or constant values to give intuitive messages. These names are passed on as parameters to the checker module. The `bind_checker` command allows port mapping, however it does not provide a `-parameters` option. Extension of "`bind_checker -parameters`" needs to be added.
- Support for expressions in `-ports` in `bind_checker`: In certain cases, the actual port of checker module can be an expression composed of power objects extracted from power architecture. For example, `save_condition/restore_condition` of `set_retention`.

D. *Advantages with this methodology*

- Because the *query_** and *find_objects* commands are dynamic in nature, any change in the UPF will automatically reflect on the output of these commands and will be fed to *bind_checker* for the correct placement of assertions. Hence this methodological usage for modeling of custom low-power assertion is immune to changes in the UPF.
- Since the methodology relies on a method to query and extract information from UPF, it has access to all power objects and design signals.
- It is highly programmable and easy to use.

V. CASE STUDIES

To better understand the methodology and demonstrate the usage, this paper includes a few case studies.

A. *Power Domain State and Transition Coverage*

Today's SOCs contain a large number of power domains and each of the power domains can operate in various power states. For the verification of such a design, one of the important properties is to ensure that "all the power domains cover all power states".

Consider the following example, in which a power domain can exist in various power states that are dependent on the primary power supply of that power domain. Below is the FSM diagram for the power state machine of a power domain:

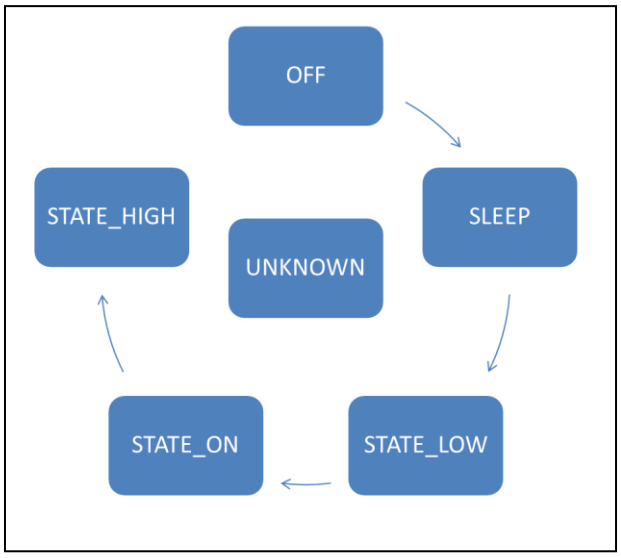

Figure2. FSM Diagram of power domain PD_LP

Covergroups are being modeled in the following checker module. In this example, the coverbin is being updated for every `STATE_LOW` reached by the power domain `PD_LP`.

Checker Module

```
module power_state_checker (input supply_net_type PSN);
parameter string PD_NAME = "";
parameter string pri_power_net_name  = "";
initial begin : COV_GRP_SAMPLE_BLOCK
    cov_grp = new;
    forever begin
       @(seq_state) ;
       if (seq_state === STATE_LOW) begin
          cov_grp.sample();
       end
    end
  end
………..
initial begin : PD_STATE_TRACKING_BLOCK
    seq_state = UNKNOWN;
    forever begin
       case (seq_state)
          UNKNOWN: begin
              if (get_supply_voltage(PSN) == 1.2) seq_state = STATE_LOW;
              else if(get_supply_voltage(PSN) == 1.8) seq_state =
STATE_HIGH;
          end
          STATE_LOW: if (get_supply_voltage(PSN) == 1.4) seq_state =
STATE_ON;
          STATE_ON: if (get_supply_voltage(PSN) == 1.8) seq_state =
STATE_HIGH;
          default: seq_state = UNKNOWN;
       endcase

       @(PSN.voltage)
        $display ("Power domain '%s' changed its voltage to '%d'", PD_NAME,
get_supply_voltage(PSN));
    end //forever
 end
……
```

The above checker module requires the handle of the primary power net of a power domain. This can be extracted from power architecture using the UPF command `query_power_domain`. We can then attach this power net to the formal port names in the checker module. After that we can instantiate the checker module into the design using the `bind_checker` command.

Tcl Proc
```
proc cov_power_states_of_power_domain {
set pd_list [query_power_domain *]
foreach PD $pd_list {
        array set pd_details [query_power_domain $PD -detailed]
        set pri_net $pd_details(primary_power_net)
        set pdName $pd_details(domain_name)
        bind_checker check_inst_$pdName \
                -module power_state_checker \
                -ports [list [list PSN $pri_net]] \
                -parameters [list [list PD_NAME $pdName]]
    }
}
```

B. Isolation Protocol Checking

In low-power designs, whenever the driving logic supply is switched off while the receiving logic supply is still on, an isolation cell is required. One of the things to be verified is that output port (op) is clamped to a golden expected value throughout the duration that isolation enable is asserted.

The above check can be expressed in the form of SVA which is written inside a checker module as follows:

Checker module:
```
module checker_isolation(input op, iso_en, clk) ;
    parameter int clamp_value      = 1 ;
    parameter isolated_signal_name = "" ;
    parameter iso_strategy_name    = "";
    always@(posedge clk)
      if(iso_en)
        assert (op == clamp_value) else $error("isolated signal '%s' for
isolation     strategy '%s' is not clamped(%b) correctly",
isolated_signal_name, iso_strategy_name, clamp_value);
endmodule
```

The above checker module requires the handle of isolated signals, isolation_enable, clk and parameter values. UPF query functions can be used to extract these handles from power architecture. These power handles are then passed as actual to the formal port names in the checker module. The last step is to attach the checker module to the design using the `bind_checker` command.

Tcl Proc:
```
proc chk_isolation_properties {
foreach domain [query_power_domain *] {
    foreach isolation [query_isolation * -domain $domain] {
        array set Iso_Strat[query_isolation * -domain $domain]
        foreach iso_sig $Iso_Strat(elements) {
          bind_checker chk_$Iso_Strat(isolation_name)_$domain(domain_name)\
            -module checker_isolation
            -ports [list \
                   [list op $iso_sig] \
                   [list iso_en $Iso_Strat(isolation_signal)]\
                   [list clk clk]\
                   ]\
             -parameters [list \
                         [list clamp_value $Iso_Strat(clamp_value)] ...]
        }
    }
}
```

VI. Drawbacks

The recommended methodology is very useful to achieve verification closure of low-power designs in a short span of time. However it may not be useful in all the scenarios because of following drawbacks:

- Since the UPF commands used for this methodology require few enhancements so it might not be portable to all tool vendors.
- Some of the highly complex low-power assertions might not be achievable using the recommended methodology because of lack of query functions/UPF commands to provide the assertion inputs. For example an assertion which requires the sink/source supply of an isolated port cannot be implemented using existing `query_*` function capabilities.
- There could be a potential simulation performance impact of bind checker assertions written using our methodology versus the tool generated assertions because tool generated assertions are highly optimized by the tool vendor.

VII. Conclusion

SystemVerilog assertions and Cover groups can be used to achieve the verification closure of low-power designs which otherwise can prove to be a very difficult task. Some of the EDA vendors provide a fixed set of low-power assertions and coverage for this purpose but there is still a need for custom low-power assertions and coverage items.

The UPF command `bind_checker` is a step towards such a goal but its standalone usage doesn't provide a strong way of writing custom assertions and cover groups. Hence we suggest a methodology using `bind_checkers`, `query commands` and `find_objects` to write some of the very powerful low power assertion which have considerable immunity from any change in the UPF or the design. We have also listed a few enhancements required in these UPF commands that enable the implementation of some very complex low-power assertions. Lastly we have demonstrated a few case studies to prove our methodology and its advantages. Although there are few minor drawbacks with the suggested strategy but the kind of flexibility it provides in writing assertions and covergroup would be a leap forward in low-power verification.

References

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

[2] Rudra Mukherjee, Amit Srivastava, Stephen Bailey: "Static and Formal Verification of Low Power Designs at RTL using UPF", DVCon 2008.