UPF GENERIC REFERENCES: UNLEASHING THE FULL POTENTIAL

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

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

Abstract: Power Aware verification is one of the tough challenges in the semiconductor industry. One of the key things to verify is the different power elements placed in the design. Unified Power Format (UPF) provides constructs such as bind_checker and query_command to ease this process [1]. The bind checker construct would require the access of UPF control signals as well as design element's component. For example, writing a custom assertion for a retention cell using bind checker might require reference to restore pin and retention clock pin. Although the retention elements pertaining to the same retention strategy would have the same restore pin but their clock might differ. This is where UPF generic references are helpful because they provide a single reserved UPF keyword to refer to different clock in the design and help users write a generic bind checker. Another usage of generic references is to specify custom retention strategy and custom isolation/level-shifter cells. In this paper we describe various Generic References provided by UPF and how they can be used to write very concise and scalable UPF.

I. Introduction

A. Power Aware Verification

Power is an important dimension of complex modern chips. Designers use complex power aware techniques such as power gating, voltage scaling, and body biasing to save power and minimize heat dissipation. Such complex techniques need verification at early stages of the design cycle to minimize re-spin costs. This is why power aware verification of designs at the RTL level is now de facto in chip manufacturing. A common practice is to first verify the RTL for functional correctness without power consideration, and then move to power aware verification.

Power aware verification involves defining a power specification for a design. These specifications involve 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 specification further involves defining elements such as isolation, retention, level-shifter cells, supply nets, and ports. The power specification and the functional design are provided to power aware verification tools to check the power correctness of the design.

At present, UPF is the most widely used power specification format. UPF specifies ways to partition the design into various power domains, specify isolation cells at power boundary, specify retention cells for sequential logic, and many other power artifacts.

B. UPF Specification

The UPF specification, *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 that are necessary to describe the power management requirements for IPs or complete SoCs. Power intent specification in UPF is used throughout the design flow; however, it may get refined at various steps in the design cycle.

The following are some of the important concepts and terminologies used in the power intent specification:

- **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 are typically all in the same power state at a given time. These groups of instances are referred as the extent of a power domain.
- **Isolation cell** — An instance that passes logic values during normal mode of operation and clamps its output to some specified logic values 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 sequential elements or a memory such that memory values can be preserved during the power-down state of the primary supplies.

C. UPF Commands (needed to understand this paper)

- **set_isolation** — To specify the boundary ports on which an isolation cell needs to be applied. It also provides details about the isolation cell behavior such as clamp value and isolation enable signal.

  set_isolation ISO_1 –domain PD –elements {IN1 IN2 IN3} –isolation_signal {iso_en} –clamp_value 1'b1

- **map_isolation_cell** —To specify the custom isolation cell on the signals of the corresponding set_isolation strategy. Although this command has been deprecated in UPF 2.1 and replaced by the **use_interface_cell** command, the basic functioning remains the same.

  map_isolation_cell ISO_1 –domain PD –lib_model_name {iso_cell} –port "ISO iso_en" –ports "in IN1"

- **set_retention** —To specify the sequential elements to which retention needs to be applied. It also provides details about the retention cell behavior such as its save/restore behavior.

  set_retention RET –domain PD –save_signal {SAVE posedge} –restore_signal {RESTORE negedge}

- **bind_checker** — To bind the custom assertions, coverage models, or debug models to various UPF objects in their design, such as isolation cell, retention cell, supply sets, and power domain. For example:

  ```
  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}}
  ```

  In above sample code, the `bind_checker` UPF command instantiates 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 *-ports* option maps actual ports in the design to formal ports of the checker model.

- **query_commands** — The UPF 2.0 and 2.1 standards provide a great toolset of query commands (for example, `query_power_domain`, `query_isolation`, and `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, and supply ports.

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

  The return value of the `query_isolation` command can be used to get isolation strategy details such as isolation enable signal, its elements, and isolation power.

II. UPF GENERIC REFERENCES

UPF generic references were first introduced in UPF 2.0 (*IEEE_1801_2009*). The generic references provide reserved keywords that cannot be redefined and would accept values depending on the context of their use. The UPF-defined generic references are as follows:

- **UPF_GENERIC_CLOCK** — Refers to the clock of a sequential logic.
- **UPF_GENERIC_DATA** — Refers to the data pin of a sequential logic or isolation input pin.
- **UPF_GENERIC_ASYNC_LOAD** — Refers to the asynchronous pins of a sequential flop.
- **UPF_GENERIC_OUTPUT** — Refers to the output of a sequential logic or isolation cell.

To negate the signal value, the Verilog bit-wise negation operator ~ can be specified before these generic references.

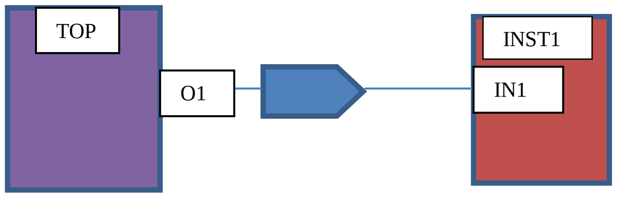
The following example of a Verilog synchronous flip-flop demonstrates these references:

```
always@(posedge reset, posedge clk)
begin
    if(reset)
            q <= 1'b0;
    else
            q <= d;
end
```

The various signals of the flop can be referred using UPF Generic References as follows:

UPF_GENERIC_CLOCK	clk
UPF_GENERIC_DATA	d
UPF_GENERIC_ASYNC_LOAD	reset
UPF_GENERIC_OUTPUT	q

Now let us take the example of an isolation cell inserted on an input port IN1 of instance INST1.



The various signals of the Isolation cell can be referred using Generic References as follows:

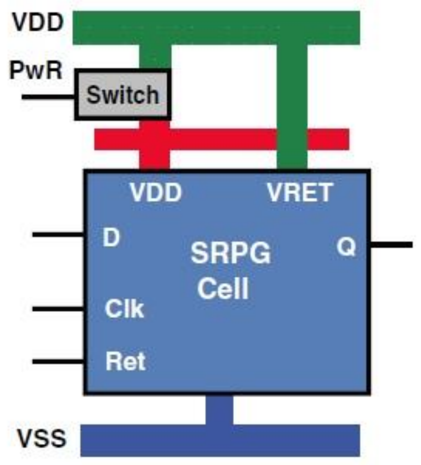
UPF_GENERIC_DATA	O1 ('actual' for the 'formal' port 'IN1')
U/PF_GENERIC_OUTPUT	IN1
UPF_GENERIC_CLOCK/UPF_GENERIC_ASYNC_LOAD	Not Applicable

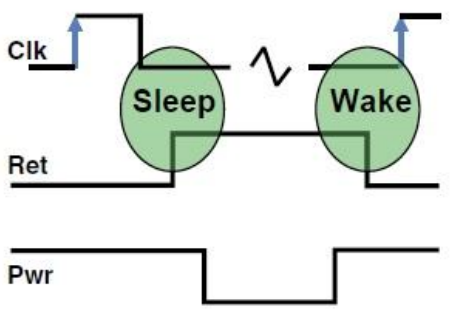
III. USAGE OF GENERIC REFERENCES IN THE UPF SPECIFICATION

Generic references can be used to specify retention strategies in a very concise and effective manner. They are also helpful in specifying the `map_isolation_cell`, the `map_level_shifter`, and the `use_interface_cell` commands for insertion of custom cells.

A. Retention Specification

Retention is an important aspect of low power designing. It provides the ability to retain the value of a state element in a power domain while switching off the primary power to that element. It also enables designers to use the retained value as the functional value of the state element upon power-up.





UPF provides the set_retention command to apply retention behavior on sequential elements such as latches or flip-flops. The command contains the following options to customize the retention behavior for different sequential elements:

- **–elements** — To select a particular set of sequential elements.
- **–save_signal/-restore_signal** — Logic signals to save and restore the register value.
- **–save_condition/-restore_condition/-retention_condition** — Boolean conditions that should be true for save/restore/retention operations.

The most common usage of set_retention is to apply retention strategy on all the sequential elements of a particular domain and provide save/restore signals to control retention behavior.

Single-control retention strategy:

*set_retention RET_STRATEGY –domain PD *

–save_signal {RET posedge} –restore_signal {RET negedge}

Dual-control retention strategy:

*set_retention RET_STRATEGY –domain PD *

–save_signal {SAVE posedge} –restore_signal {RESTORE posedge}

The above strategy would apply retention on all the state elements of power domain (*PD*) and all the register values are saved at the posedge of the '*SAVE*' signal and restored at the posedge of the '*RESTORE*' signal.

Nowadays, designers also want to restrict the Save/Restore operations on the values of clock/asynchronous load signals. So, they use the *–save_condition/-restore_condition* options with the *set_retention* command. For example, to restore only the register when the clock is not active, the following strategy can be used:

*set_retention RET –domain PD –save_signal {SAVE posedge} *

–restore_signal {RESTORE posedge} –restore_condition {!CLOCK}

Since restore_condition/retention_condition is different for different sequential elements (because their clock/asyn_load signals are different), it poses following problems:

- Users need to determine and group the clock/async_load of retention flops in the design. This is a daunting task.
- Even if users are able to extract the clk/async_load of their retention flops, they need to write numerous retention strategies, which will make their UPF file huge and error-prone.

- Custom assertions for retention cells, if written by users, are numerous and tough to maintain.

All of the above problems can be averted using UPF generic references. The following use case depicts how generic references can be helpful.

Use Case 1

A design contains two asynchronous flops, ‘Q1’ and ‘Q2,’ with different clocks. Both of them require to be retained if their respective clock is active low during restore. This is a common situation because designs usually have multiple gated clock signals so multiple retention strategies are required to model different retention conditions.

always @(posedge CLOCK1, posedge RESET)	set_retention RET1
begin	–domain PD
if(RESET)	–save_signal {SAVE posedge}
Q1 <= 1’b0;	–restore_signal {RESTORE posedge}
else	–restore_condition {!CLOCK1}
Q1 <= D;	–elements {Q1}
End	
always @(posedge CLOCK2, posedge RESET)	set_retention RET2
begin	–domain PD
if(RESET)	–save_signal {SAVE posedge}
Q2 <= 1’b0;	–restore_signal {RESTORE posedge}
else	–restore_condition {!CLOCK2}
Q2 <= D;	–elements {Q2}
End	

As the design grows, users have to identify and write as many retention strategies, which makes it a tedious and cumbersome process. One missing strategy can cause the complete design to fail. This is where UPF generic references come to rescue. UPF generic references allow users to write a single generic retention strategy for all the sequential elements.

always @(posedge CLOCK1, posedge RESET)	
begin	
if(RESET)	
Q1 <= 1’b0;	set_retention RET
else	–domain PD
Q1 <= D;	–save_signal {SAVE posedge}
End	–restore_signal {RESTORE posedge}
always @(posedge CLOCK2, posedge RESET)	–restore_condition {!UPF_GENERIC_CLOCK}
begin	–elements {Q1 Q2}
if(RESET)	
Q2 <= 1’b0;	
else	
Q2 <= D;	
End	

Only one set_retention strategy is required to cover all different types of CLOCK signals. EDA tools parse this UPF_GENERIC_CLOCK in restore_condition and replace it with the specific CLOCK of each sequential element for application of retention strategy. It is a more automated, robust, and easier flow for designers. Addition or deletion of sequential elements in a design does not require any change in the UPF. Similar generic references can be used for –save_condition and –restore_condition. Designers can use all types of UPF_GENERIC references according to their requirements. Additional enhancements are required in EDA tools to support this feature; this is covered in later sections.

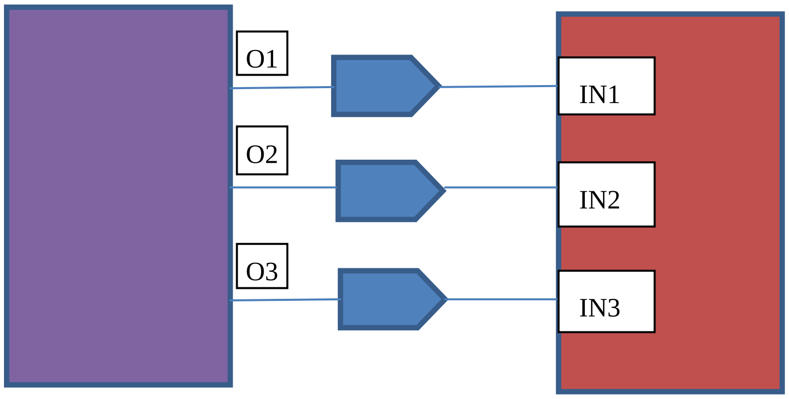
B. Isolation Specification (map_isolation_cell)

UPF provides the command *set_isolation* to describe the ports on which isolation is required. Commonly, users rely on tool-inserted standard isolation cells for RTL-level verification but sometimes, they want to verify their gate-level isolation model at the RTL stage itself. UPF provides commands such as *map_isolation_cell/use_interface_cell* to insert user's custom isolation model. The main difficulty in writing such commands is to provide port association. The following use case demonstrates how to use UPF generic references to ease this problem.

Use Case 2

A design contains three input ports, 'IN1,' 'IN2,' and 'IN3,' which require isolation. The user wants to verify the custom isolation cell rather than the tool standard cell for this purpose.

set_isolation ISO –domain PD –elements {IN1 IN2 IN3} –isolation_signal {iso_en}



The user's custom isolation cell has the following interface:

module SFX_ISO(input ISO, input I, output Z);

…

endmodule

The user needs to write three different map isolation cells as follows:

*map_isolation_cell ISO –domain PD –elements {IN1} –lib_model_name {SFX_ISO} *

*-ports "ISO iso_en" *

*-ports "I O1" *

-ports "Z IN1"

*map_isolation_cell ISO –domain PD –elements {IN2} –lib_model_name {SFX_ISO} *

*-ports "ISO iso_en" *

*-ports "I O2" *

-ports "Z IN2"

A similar command is required for the custom cell at IN3. The problems in inserting such custom isolation cells are as follows:

- Users need to find out the actual of the isolated port from their design for every such port. These actuals can be complex expressions (for example, a concat expression and a bit-select expression) and would be difficult to specify in *map_isolation_cell* command.
- Users need to write as many different *map_isolation_cell* commands as the number of isolated ports. Since complex designs have thousands of such isolation cells, this way of writing thousands of UPF commands is very error-prone and hard to maintain.
- If a designer has written an isolation strategy using the *-source/-sink* or the *-applies_to* option, it is very difficult to find the ports that will get isolated because of the strategy.

Using UPF generic references, the map_isolation_cell command can be written in a concise and simple form as follows:

*map_isolation_cell ISO –domain PD –lib_model_name {SFX_ISO} *

*-ports "ISO iso_en" *

*-ports "I   UPF_GENERIC_DATA" *

-ports "Z   UPF_GENERIC_OUTPUT"

There is no need to even specify the -elements option in map_isolation_cell, if users want to apply their custom isolation cell for every port specified in the isolation strategy.

IV. USAGE OF GENERIC REFERENCES IN UPF VERIFICATION

Low-power verification involves verifying various power elements such as isolation cells, retention cells, level shifters, repeaters cells, and power domain simstates. Verification itself involves various stages such as static verification and dynamic simulation. Dynamic simulation further involves verification based on simulation results, coverage and assertion. The scope of this paper is to demonstrate the use of UPF generic references in the verification of isolation and retention cells using custom coverage and assertions models in dynamic simulation.

A. Isolation Verification

EDA vendors provide automated static as well as dynamic checks to verify the correctness of isolation cells, but these are insufficient to verify the correct functionality of these isolation cells. Often, users write their custom assertions and coverage models using the *bind_checkers* command to verify the functionality [1]. The following example shows the use of generic references in bind checkers.

Use Case 3

In use case 2, the user wants to verify that the isolation cell acts like a buffer during non-isolation period. For this purpose, the user would like to write custom assertions in a module and bind it to design signals using bind checkers as follows:

User's checker module interface:

module iso_checker(input iso_in, output iso_out, input iso_en);

UPF bind checker commands:

```
bind_checker IN1_iso_checker –module iso_checker \
             -ports {{iso_in  O1}  \
                     {iso_out IN1}   \
                     {iso_en  iso_en}}

bind_checker IN2_iso_checker –module iso_checker \
             -ports {{iso_in  O2}  \
                     {iso_out IN2}   \
                     {iso_en  iso_en}}
```

A similar bind_checker needs to be written for IN3. The following shows how to write the same bind checkers in a simpler and scalable manner using UPF Generic references.

```
array set Iso_Strat[query_isolation ISO -domain PD -detailed]

bind_checker $iso_sig_iso_checker \
      -module iso_checker \
       -elements $Iso_Strat(elements)\
      -ports {{iso_in UPF_GENERIC_DATA } \
             {iso_out UPF_GENERIC_OUTPUT}   \
             {iso_en  iso_en}}
```

The above command binds the checker instance for each port in the *–elements* option and UPF_GENERIC_DATA and UPF_GENERIC_OUTPUT is replaced by the corresponding actual/formal of each port. Users do not have to worry about the identification of input/output of each isolated port as these are automatically addressed by the tools.

B. Retention Verification

After the application of retention strategies, the next step is verification of the retention cells. Often, users write their custom assertions and coverage models using the bind checkers command to verify the functionality [1].

Use Case 4

In use case 1, if user wants to verify retention behavior of sequential elements, he would write a custom checker module and bind it to design signals using bind_checker command. In checker module, he can write assertions or coverage model to verify the restore behavior when both restore signal and *restore_condition* is active. It would require multiple bind_checker commands as *restore_condition* is different for each element.

User's checker module interface:

```
module ret_checker(input restore_condition, input ret_ff, input restore_signal);
```

User's bind checker command:

```
bind_checker Q1_ret_checker –module ret_checker –bind_to inst1 \
          -ports {{restore_condition !CLOCK1 } \
                {ret_ff  Q1} \
               {restore_signal RESTORE}

bind_checker Q2_ret_checker –module ret_checker –bind_to inst1 \
          -ports {{restore_condition !CLOCK2 } \
                 {ret_ff  Q2} \
               {restore_signal RESTORE}
```

This requires writing as many bind_checker statements as the number of clocks/flops and this is very error-prone. UPF generic references provide an easy solution to handle this problem as follows:

```
array set RET_STRTGY [query_retention RET –domain PD –detailed]
set ELEMENTS  $RET_STRTGY(elements)
set RESTORE_COND  $RET_STRTGY(restore_condition)

bind_checker ret_checker_inst –module ret_checker –elements $ELEMENTS \
          -ports { {restore_condition $RESTORE_COND} \
                 {ret_ff   UPF_GENERIC_OUTPUT} \
                {restore_signal RESTORE}
```

In the above command, RESTORE_COND is !UPF_GENERIC_CLOCK. This command is applied to each element of the bind_checker command and UPF_GENERIC_CLOCK in the restore condition is replaced by the corresponding CLOCK of each sequential element. With the help of only one command, all the retention elements are verified.

C. Sequential Element Verification

The UPF generic references can be used for custom verification of each sequential element with the help of the bind_checker command.

Use case 5

If user wants to verify that clock should be off when async load is active, he can write following bind_checker commands.

Checker module :

```
    module checker { input clk, input async_load, input seq_elem}
            always @(posedge async_load)
            begin
              assert ( !clk ) else $display ("Error : CLK is not OFF, Element value is '%b' .",seq_elem);
            end
    endmodule
```

```
set ELEMENTS [query_retention RET –domain PD –detailed](elements)
bind_checker checker_inst –module checker –elements $ELEMENTS \
            -ports { {clk UPF_GENERIC_CLOCK} \
                    {async_load UPF_GENERIC_ASYNC_LOAD} \
                    {seq_elem UPF_GENERIC_OUTPUT} }
```

The above command would bind checker instance for each sequential element and the UPF generic references would be replaced by corresponding CLOCK/ASYNC_LOAD in port list.

V. EXTENSIONS REQUIRED IN THE UPF SPECIFICATION/EDA TOOLS

Although UPF standard defines all required UPF Generic References, it does not provide specific information on their usage. We propose following leniency and enhancements in UPF specification.

A. Retention Specification

The UPF command *set_retention* needs to be enhanced to accept UPF generic references and its simple expressions in the following options:

- save_condition
  set_retention RET –domain PD –save_condition {UPF_GENERIC_CLOCK} …

- restore_condition
  set_retention RET –domain PD –restore_condition {!UPF_GENERIC_CLOCK} …

- retention_condition
  set_retention RET –domain PD –retention_condition {UPF_GENERIC_ASYNC_LOAD && !UPF_GENERIC_CLOCK} …

B. *query_* Commands*

The query command *query_retention* needs to be extended to provide additional information that does not comply with UPF standard. For example, *query_retention* for *retention_condition* needs to be extended to provide the expressions in terms of Generic references as specified in retention strategy.

C. *Bind Checker*

To make the UPF command bind_checker useful with UPF Generic references, it needs the following enhancements:

1. Modify the *-ports* option to accept generic references as well as its simple expressions for port mapping:

```
bind_checker checker_inst –module checker –elements $ELEMENTS \
        -ports { {clock_signal (UPF_GENERIC_CLOCK && !UPF_GENERIC_ASYNC_LOAD)} …
```

The *-ports* option should also accept the output of *query_functions* in its port mapping. Note that such outputs are actually expressions of generic references:

```
set_retention RET –domain PD –save_condition {UPF_GENERIC_CLOCK && sig_save} \
bind_checker checker_inst –module checker –elements $ELEMENTS \
        -ports { {save_cond $ret_array(save_condition)} …
```

2. Relax the -elements option to accept the list of signals (sequential elements). Currently, UPF requires that it is the list of instances in which the checker module needs to be binded. Also, the checker instance name needs to be modified for different signals in its -elements list.

```
bind_checker checker_inst –module checker –elements {inst1/q1 inst1/q2} …
```

In the above case, ‘q1’ and ‘q2’ are signals within the instance ‘inst1’. The expected behavior of the above command should be to bind two instances, namely ‘checker_inst_q1’ and ‘checker_inst_q2’ in the scope ‘inst1’.

D. *EDA Tools*

EDA tools should be able to process generic references in the context of the -elements option of the bind checker command.

```
set_retention RET –domain PD –elements {q1 q2} … \
bind_checker checker_inst –module checker –elements $ELEMENTS \
        -ports { {clock_signal UPF_GENERIC_CLOCK} …
```

In the above example, the tool should be able to extract the respective clocks of ‘q1’ and ‘q2’ signals and bind the ports accordingly.

VI. Conclusion

The UPF generic references are a boon for Low Power Designing and Verification as these are easy-to-use and provide automation. The references facilitate the writing of a concise, scalable, and less error-prone UPF. Modeling a retention cell in a UPF in which the save/restore behavior is dependent on its clock is nearly impossible without the generic references. Similarly, using the custom isolation cell and checker modules would be a difficult task without the generic references. The UPF generic references also provide EDA tools with greater optimization opportunities because these are evaluated in their local context.

VII. References

1. Madhur Bhargava, Mentor Graphics, Durgesh Prasad, Mentor Graphics, “Low-Power Verification Methodology using UPF Query functions and Bind checkers,” DVCon Europe 2014.
2. IEEE Std. 1801™-2013 for Design and Verification of Low Power Integrated Circuits. IEEE Computer Society, 29 May 2013.