# UVM and UPF: an application of UPF Information Model

Amit Srivastava
asrivas@synopsys.com

Harsh Chilwal
harsh@synopsys.com

Srivatsa Vasudevan
srivats@synopsys.com

***Abstract*- Due to ever increasing complexity of today's SoC designs, sophisticated verification methodologies like UVM have gained popularity. However, more designs now use power management strategies to minimize power consumption which uses UPF to provide power intent in a side file. There hasn't been a proper integration of UVM with the UPF standard – leading to adhoc or proprietary extensions. In this paper, we will provide an application of the UPF Information Model APIs to enable UVM interact with low power designs having UPF. This will address the problem of UVM interoperability with UPF.**

## I. INTRODUCTION

UVM testbench methodology has gained a significant amount of popularity in recent years. With the growing complexity of designs, power management in contemporary designs has gained a great deal of prominence. Power management techniques have become a must-have due the limited amount of power available from a power source. Power management itself comes in a variety of ways depending on the design choices made. Popular techniques like clock gating, frequency scaling, multi-voltage designs are all in frequent use.

The IEEE 1801 LRM provides standard UPF packages for SystemVerilog, and VHDL testbenches to import the appropriate UPF packages to manipulate the supply pads of the design under verification. Users must import the package and make appropriate changes to their testbenches to make use of facilities available in the UPF package.

UVM testbenches needs to be adaptive even for low power designs. To achieve that the UVM testbench needs to keep track of the operating state of the design. The operating state of the design depends on the state of the power domains which model the power management architecture of the design. e.g. a transaction sent to the design which is powered off should take that into consideration and accordingly fail. To address this, verification engineers have either adopted proprietary extensions or model this on some control signals. The problem with proprietary extensions is that its not portable across tools and reliance on control signals can become inaccurate as the design evolves and goes to lower abstraction levels.

Moreover, UVM testbenches often need to manipulate the state of DUT and hence may affect the state of the power management architecture as well. In the absence of standard interface, this task becomes challenging and depends on proprietary extensions.

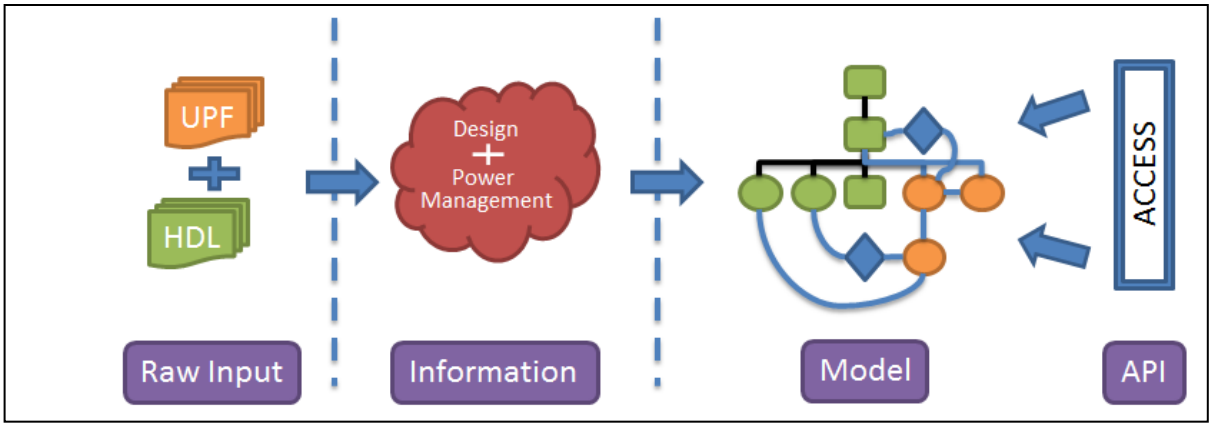
Since, UVM presents an Object Oriented verification environment which is not available with UPF. The interaction between the UVM classes and UPF Information Model becomes challenging. The UPF Information Model APIs are similar to VPI / VHPI which offers procedural APIs and not very convenient to use in an Object Oriented verification environment. The state mirroring support in the UPF Information Model is based on mirroring of static types which requires additional compilation steps to generate additional HDL code which interacts with UVM verification environment.

UVM environments help define covergroups and control which is lacking in UPF environment. Users rely on proprietary extensions to generate covergroups for the UPF objects which depends on additional flags to the compilation stage which generates models containing covergroups.

In this paper, we present a class library which is based on UPF Information model and enables seamless integration with the UVM environment.

## II. UPF INFORMATION MODEL

The IEEE 1801-2015 UPF [1] standard introduced the new UPF Information model (UPF IM). It is one of the most significant additions to the UPF standard, which enables access to the power management information. A number of interesting applications can be built using the APIs defined by the standard [2]. In this paper we use the UPF Information Model to build a class library which enables UVM integration.




The UPF information model captures the power-management information which is the result of application of UPF commands on the user design. It consists of a set of objects containing information and various relationships between them. The model contains information about UPF objects and user design in order to comprehensively capture the power intent in a standard form which can be queried via UPF queries and UPF HDL package functions.

The UPF standard defines APIs in three different languages, Tcl, SystemVerilog and VHDL. It provides small set of APIs which enable access to the UPF Information Model. The Tcl APIs support access to static structure of the information, whereas SystemVerilog and VHDL provide both static and dynamic access. They also allow limited write capability to enable building sophisticated testbench capabilities. The Standard defines a set of SystemVerilog functions which allow access to the UPF IM.

While the UPF standard describes the UPF Information Model and APIs in great detail, in this paper, we focus of few APIs which are relevant to the application we are developing.

*A. Getting handle of UPF Object*

The handle of any UPF object can be accessed by upf_get_handle_by_name API, which accepts a string pathname which points to the UPF object.

```
upfHandleT upf_get_handle_by_name(upfStringT pathname, upfHandleT relative_to = null);
```

The example below illustrates the usage of API:

*Example 1: Get handle to a power domain*

*SV code*

```
initial begin
upfHandleT pd = upf_get_handle_by_name("/top/dut_i/pd");
...
End
```

*B. Read Access*

The read access is achieved by the upf_query_object_properties API. The syntax is shown below:

```
upfHandleT upf_query_object_properties(upfHandleT object_handle,upfPropertyIdE attr);
```

This API accepts a handle to object in UPF IM and an enumerated value to point to property which is requested. Below example illustrates the usage of this API:

*Example 2: Get current value of logic net*

*SV code*

```
initial begin
upfHandleT lnet = upf_get_handle_by_name("/top/dut_i/logic_net");
upfHandleT curr_value = upf_query_object_properties(lnet,
UPF_CURRENT_VALUE);
if (upf_handle_in_class(curr_value, UPF_BOOLEAN)) begin
upfBooleanT val = upf_get_value_int(curr_value);
...
end
end
```

*Output*

```
Iso Name: iso
```

*C. Continuous Read Access*

The standard also provides an API to perform a continuous read access of the dynamic properties of the UPF objects. This is achieved via upf_create_object_mirror API which creates a mirror relationship from a source UPF object to any destination object in the testbench environment. The syntax is show below:

```
upfBooleanT upf_create_object_mirror(upfStringT src, upfStringT dst);
```

Below example illustrates the usage of this API:

*Example 3: Create a monitor of UPF supply*

*SV code*

```
module tb;
upfSupplyObjT vdd_monitor;
upfBooleanT status;
initial begin
status = upf_create_object_mirror("/top/dut_i/vdd", "vdd_monitor");
end
always @vdd_monitor begin
$display($time, " Supply %s changed\n",
upf_query_object_pathname(vdd_monitor.handle));
end
endmodule
```

*Output*

```
100 Supply /top/dut_i/vdd changed
200 Supply /top/dut_i/vdd changed
```

*D. Immediate Write Access*

The standard also defines a set of APIs to write values on the UPF objects. These are:

1. `upfBooleanT set_power_state(string object, string power_state);`
2. `upfBooleanT supply_on(string supply_name, real value = 1.0);`
3. `upfBooleanT supply_off(upfStringT supply_name);`

These APIs allow modifying the state of UPF objects like, Power Domain, Supply Set, Supply Net and Supply Port.

Below example illustrates the usages of these APIs:

*Example 1: Changing supply voltages*

*SV code*

```
upfSupplyObjT vdd_local;
upfSupplyTypeT vdd_value;
initial begin
status = supply_on("/top/dut_i/vdd", 1.2);
...
status = supply_off("/top/dut_i/vdd");
...
#10 set_power_state("/top/dut_i/PD", "domain_off");
...
```

## III. UVM INTERFACE REQUIREMENTS

UVM is a methodology based on Object Oriented concepts leveraging SystemVerilog classes. In a low power design, the UVM testbench needs to be aware of the power management architecture, more specifically the power state of the design. This is highly important as to minimize power consumption, designers use sophisticated power management techniques like power gating, multi-voltage, etc. These power management techniques have impact on regular functionality of the design and hence the testbench needs to be aware of the effect of these techniques. It is important that it also enables power aware verification, which includes verification of power management along with the regular functionality of the design.

Since, power management is captured in the side file in UPF format, the traditional UVM methodology is unable to access the power management information by just referring to the HDL files. Hence, it either ignores the power management information or must rely on proprietary tool extensions – which is not portable.

The UPF Information Model provides a complete access in terms of basic APIs which are basically SystemVerilog functions. Being part of the UPF standard it provides a portable access to power management information.

Although, UPF IM is able to provide access to power management information, it still is a procedural interface – similar to VPI (SystemVerilog) or VHPI (VHDL). Using it for make UVM testbenches power aware is not a trivial task and requires translating procedural interface to object oriented interface.

More specifically, the following capabilities are needed to enable interface with the UVM world.

### A. *Access Object Oriented Handles*

UVM testbenches should be able to access UPF Objects in the form of SystemVerilog class objects. While this may not be an absolute requirement it becomes convenient to build a complex testbench in an Object Oriented manner.

### B. *Callbacks*

UVM testbenches need to be able to define callbacks on the states of UPF Objects like Power Domain. The application should be able to trigger events whenever there is any change in the state which can be caught by the UVM testbench and appropriate action can be taken.

### C. *Modifying State*

Another important thing is ability to modify the state of UPF objects to a given state. It is needed to create more abstract testbenches in the UVM environment.

## IV. PROPOSED SOLUTION

In this paper, we address the UVM requirements by creating a class library which is an application of UPF Information Model. This library can be used by any simulation tool that support UPF Information model.

The library is divided into two parts UPF Classes and UPF Mirror Utilities. The UPF Classes define the SystemVerilog classes which will represent UPF Information Model Objects and how they interface with the UPF IM. The UPF Mirroring Utilities are a set of utilities which are needed to establish the link between the SystemVerilog classes and the UPF Information Model objects to overcome simulator and SystemVerilog limitations. We believe that adding these classes to the IEEE 1801 UPF standard will enable tools to implement mirroring capabilities natively in the simulation eliminating the need for UPF Mirroring Utilities.

### A. *UPF Classes*

We have defined SystemVerilog classes corresponding to UPF Objects defined in the Information Model. These classes are essentially a wrapper over UPF Object handles, but contain important events corresponding to dynamic properties of the UPF Objects. The UVM testbench can put callbacks on the events of the objects representing UPF Objects.

The sample code for some of UPF objects is shown below:

```
class upfPdSsObj extends upfBase;
int id;
event current_state_changed;
event current_simstate_changed;
event power_down;
event power_up;
upfSimstateE current_simstate;
process p1;
process p2;
covergroup cov_simstate ;
    cover_sim_state : coverpoint current_simstate;
endgroup

function new(upfHandleT handle);
    super.new(handle);
    populate_all_mirrors();
    this.id = pd_ss_mirror_map[upf_query_object_pathname(handle)];
    if (this.id == 0) begin
        `uvm_error("UPF_UVM",$sformatf("[ERROR] UPF_UVM: Failed mirror for:
%s", get_pathname()))
        return;
    end else begin
```

```
        `uvm_info("UPF_UVM", $sformatf("Successfully mirrored [%d]: %s",
this.id, get_pathname()),UVM_LOW)
    end
    fork
    begin
        p1 = process::self();
        forever @(pd_ss_mirror[id].current_state.handle) begin
            `uvm_info("UPF_UVM,",$sformatf("UPF_UVM: Current state changed:
%s", get_pathname()),UVM_LOW)
            -> current_state_changed;
        end
    end
    begin
        p2 = process::self();
        forever @(pd_ss_mirror[id].current_simstate) begin
            current_simstate = pd_ss_mirror[id].current_simstate;
            `uvm_info("UPF_UVM",$sformatf("[INFO ] Current simstate changed:
%s   to %s", get_pathname(), current_simstate.name()),UVM_LOW)
            -> current_simstate_changed;
            cov_simstate.sample();
            if (pd_ss_mirror[id].current_simstate == NORMAL) begin
                -> power_up;
            end else begin
                -> power_down;
            end
        end
    end
    join_none
    cov_simstate = new();
endfunction
function upfPowerState get_current_state();
    upfPowerState ps = new(get_object_property_handle(UPF_CURRENT_STATE));
    return ps;
endfunction
function upfSimstateE get_current_simstate();
    return pd_ss_mirror[id].current_simstate;
endfunction
function void release_handle();
    if (this.id != 0) begin
        p1.kill();
        p2.kill();
    end
endfunction
endclass
```

*B. UPF Mirror Utilities*

The library also defines certain utility functions which help establish mirror relationships between the objects of classes defined in the library and the objects in the UPF Information Model.

The code snippet for the utility is shown below:

```
function void create_pd_ss_mirror(upfHandleT ss_handle);
    upfBooleanT  bMirrorStatus;
    string       sMirrorNodeSelect;
    string       sMirrorIdx;
    sMirrorIdx.itoa(pd_ss_mirror_idx);
    sMirrorNodeSelect = $sformatf("%s[%s]", sPdSsMirrorNode, sMirrorIdx);
```

```
    $display ("UPF_UVM: Creating mirror: [%s => %s]",
upf_query_object_pathname(ss_handle), sMirrorNodeSelect);
    bMirrorStatus =
upf_create_object_mirror(upf_query_object_pathname(ss_handle),
sMirrorNodeSelect);
    if (bMirrorStatus == 1'b1) begin
        pd_ss_mirror_map[upf_query_object_pathname(ss_handle)] =
pd_ss_mirror_idx;
        $display("UPF_UVM: Success");
    end
    pd_ss_mirror_idx = pd_ss_mirror_idx + 1;
endfunction

function void populate_all_mirrors();
    upfHandleT design_handle;
    upfHandleT pd_iter;
    upfHandleT pd_handle;
    if (bMirrorPopulated) return;
    bMirrorPopulated = 1'b1;
    design_handle = upf_get_handle_by_name("#UPFDESIGN#");
    pd_iter = upf_query_object_properties(design_handle,UPF_POWER_DOMAINS);
    pd_handle = upf_iter_get_next(pd_iter);
    $display ("UPF_UVM: Populating all PD Supply Set handle mirrors");
    while (pd_handle != upf_null) begin
        $display ("UPF_UVM: PD: %s",
upf_get_value_str(upf_query_object_properties(pd_handle,UPF_NAME)));
        create_pd_ss_mirror(pd_handle);
        pd_handle = upf_iter_get_next(pd_iter);
    end
endfunction
```

*C. Low Power Coverage*

The new classes in the library also enable capturing coverage metrics related to UPF objects. We have added covergroups to capture coverage related to states of the power domain. These classes can also be extended to capture more coverage metrics like state transitions. Users can use the events triggered by these classes to capture more custom coverage metrics related to their environment.

The code to capture coverage for Power Domain is shown below:

```
covergroup cov_simstate ;
    cover_sim_state : coverpoint current_simstate;
endgroup
```

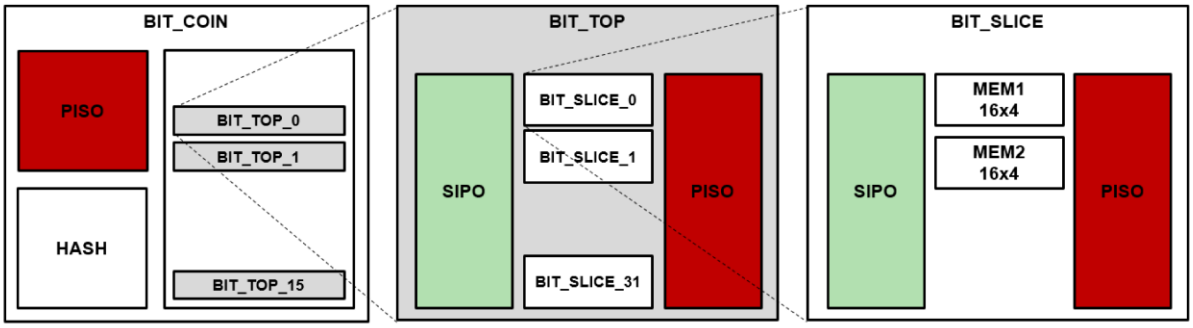
The sampling code is added in the below code:

```
    begin
        p2 = process::self();
        forever @(pd_ss_mirror[id].current_simstate) begin
            current_simstate = pd_ss_mirror[id].current_simstate;
            `uvm_info("UPF_UVM",$sformatf("[INFO ] Current simstate changed:
%s   to %s", get_pathname(), current_simstate.name()),UVM_LOW)
            -> current_simstate_changed;
            cov_simstate.sample();
            if (pd_ss_mirror[id].current_simstate == NORMAL) begin
                -> power_up;
            end else begin
                -> power_down;
            end
        end
    end
```

## V. CASE STUDY (BITCOIN DESIGN)

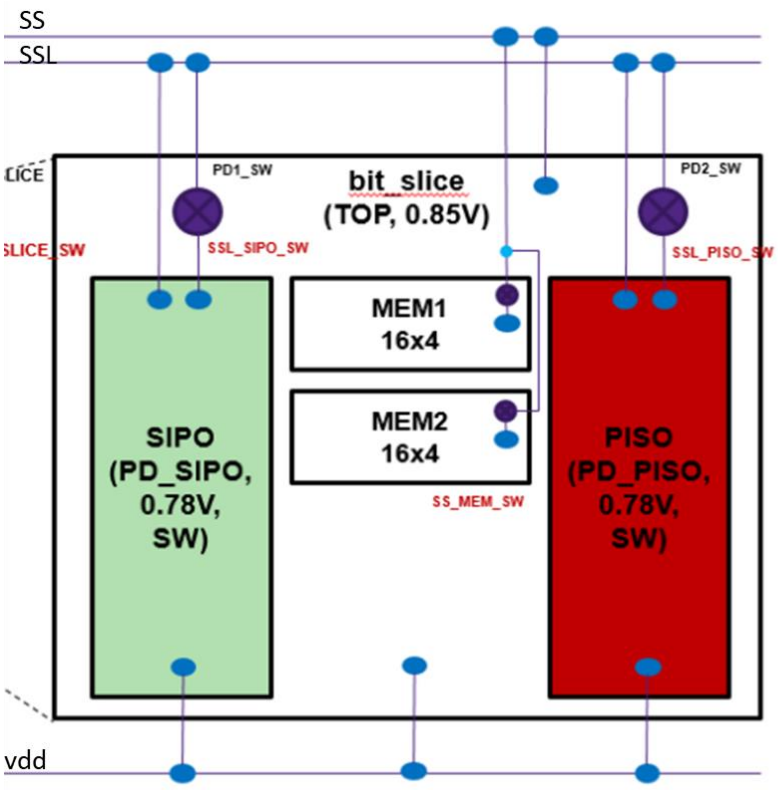
Bitcoin is a popular digital currency that has garnered a lot of attention lately. The compute servers that make up the Bitcoin Network comprise the most powerful network on the planet, and it's become so just in the past ten years. A Bitcoin Mining server is essentially a massive set of parallel hashing functions, whose sole purpose is to solve mathematical challenges as quickly as possible, to gain a reward of Bitcoins. Thus, mining is the method in which new Bitcoins are made available. These Miners in the early days were based on CPUs and later GPUs; but later FPGAs and now specialized ASICs have come into play. A massive collection of Mining Systems, also known as Mining Farms, are now dominant in the Bitcoin Network.

For the purposes of this paper, we focus on a bitcoin slice. "BIT_SLICE", which is comprised of a SIPO (Serial In-Parallel Out) block, two memories, and a PISO (Parallel In-Serial Out) block. The BIT_SLICE block is instantiated 32-times in "BIT_TOP", which also has SIPO and PISO blocks. And finally, BIT_TOP is instantiated 16 times in "BIT_COIN" which has a PISO and HASH block. The HASH used is a simplified version of the real hash, a SHA-256 (Secure Hash Algorithm, 256-bit) and such a design is enough for our purposes.




## VI. THE UPF AND POWER SCHEMA FOR THE DESIGN

For the purposes of low power integration also, we only focus on a simple testbench that has a single bit_slice along with the associated UPF for this design which serves to illustrate the capabilities in this paper.

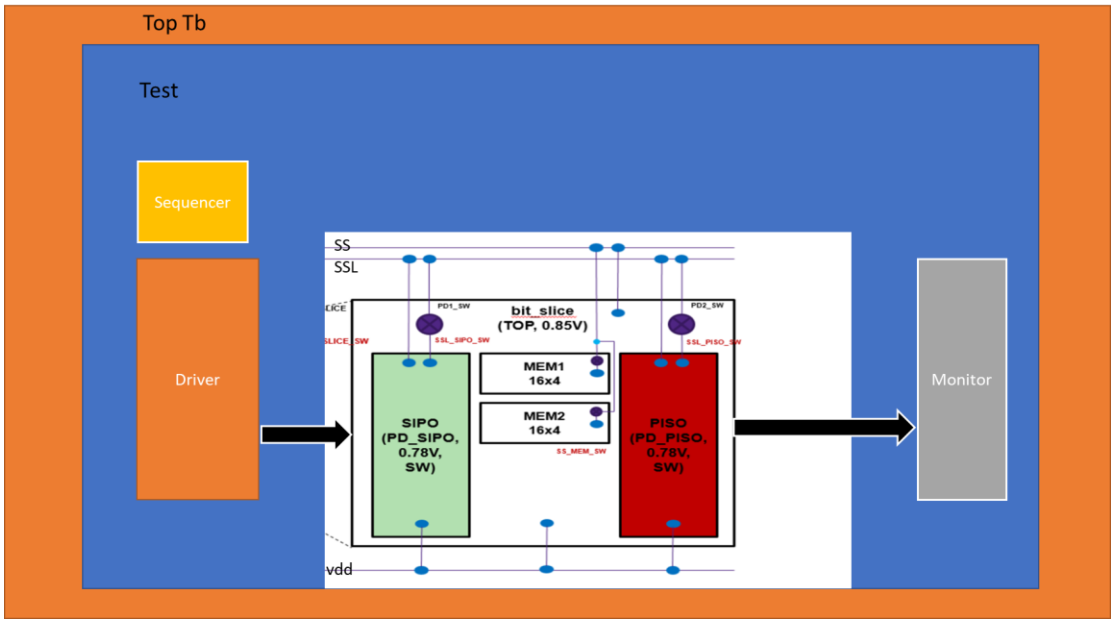



The current design shown in the figure is a multi voltage design with the SIPO and PISO blocks run at a lower voltage. The memories, and top level logic run at the higher voltage. The SIPO and PISO blocks at each level are switched and the memories in the bit_slice module have internally switched supplies. All the SIPO blocks are tied together, so all SIPOs at all levels are commonly switched. Same is true for the PISO blocks.

The UPF for the above design includes the creation of three power domains TOP, PD_SIPO, and PD_PISO in addition to a couple of power switches that are shown in the diagram. A number of power states are also present in the UPF. These power switches are accessible from I/O signals at the module port which is wired to the interface in the UVM testbench that can drive the low power modes.

## VII. UVM INTEGRATION AND RESULTS

The DUT is placed in a test harness along with a UVM testbench connected to it via SV interfaces for the driver and the monitor as shown in the block diagram below. The simulator has been provided with the UPF using the tool options in addition to the DUT and testbench files.




The only salient change in this testbench is the addition of the UPF to initialize the supplies as shown in the listing below.

```
//UPF Related
import UPF::*;
initial
begin
  supply_on ("SS.power", 0.85);
  supply_on ("SSL.power", 0.78);
  supply_on ("SS.ground", 0.0);
  supply_on ("SSL.ground", 0.0);
  supply_on ("SS.ground", 0.0);
  supply_on ("SS_MEM_SW.ground", 0.0);
  supply_on ("VSS", 0.0);
end
```

As in UVM testbenches, the monitor and driver interface signals are connected to the Bitslice design via virtual interfaces. The top level environment contains the instance of the driver and monitor and also an instance of the upfPowerDomain class from the package described above. This upfPowerDomain class is instanced in the environment and connected to the power domain in the build_phase() of the environment. As a result, the class now contains information whenever the power state changes.

```
// Environment additions.
import UPF::*;
import UPF_UVM::*;
class bitcoin_env extends uvm_env;
// monitor and driver instances
upfPowerDomain c_pd_piso;
```

In addition, a uvm_event named powerDomain_event is also instantiated in the environment. This event serves to notify components of power state changes.

```
uvm_event powerDomain_event;
```

This class is connected to the power domain via the upf function upf_get_handle_by_name under a macro during the build_phase.

```
c_pd_piso = `UPF_NEW("/dut/bit_secure_0/slice_0/PD_PISO");
```

During the run_phase, the state of the power state is maintained in the class handle c1. This is done by the following piece of code below which also prints out a message any time the state changes and triggers the powerDomain_event;

```
task run_phase(uvm_phase phase);
// ...
  fork begin
    forever @ (c_pd_piso.current_state_changed) begin
      upfPowerState ps = c_pd_piso.get_current_state();
      powerDomain_event.trigger();
      `uvm_info(get_full_name,$sformatf($time," UPF_UVM: %s changed to
%s",c1.get_pathname(),ps.get_name()), UVM_LOW)
      ps.release_handle();
    end
  end
join_none
// ...
```

The power domain handle is also placed in the config_db in the environment in the build_phase().

```
powerDomain_event = new("powerDomain_event");
uvm_config_db  #(uvm_event)::set(null, "", "powerDomainEvent",
powerDomain_event);
```

The monitor component is able to get the handle to this event from the config_db and use it to detect power state changes.

The DUT power switches are wired to the module DUT ports and are connected to the driver and monitor interfaces.

The testbench environment is designed to produce stimulus via sequences and the driver which then drives them into the DUT. The driver also drives the switches for low power operation. Specific sequences have been created to power up/power down portions of the bit_slice module.

When the sequences are run on the sequencer and driven on the DUT pins by the driver, the bitslice design cycles through its power states. As the power state changes, the class in the environment that monitors the power state detects the power state change and displays messages in the log file as shown below.


In addition, the monitor component having obtained access to the powerDomain_event can easily be notified of state changes. This is accomplished by the code below:

```
uvm_config_db #(uvm_event)::get(null, "", "powerDomainEvent",
powerDomain_event);
```

## VIII. FUTURE WORK

This paper proposes a method to import UPF Information Model Objects into a verification environment and demonstrates how such an integration may be accomplished to leverage existing testbenches with UPF and UVM.

The upf_uvm package and its current implementation suffers from a few performance problems simply because the UPF package as it is defined today provides a non-performant API into the simulator. Work is ongoing to expose the entire UPF API to the UVM class library and also to identify the non-performant areas.

The class that exposes the UPF information model in its current form is limited in functionality, we anticipate having to supplement the class with additional properties.

This paper does not discuss a power domain manager or other functionality that would be essential in an UVM environment that has multiple power domains. Work is ongoing to define an efficient power domain manager class that can communicate effectively with components in UVM testbenches.

## IX. CONCLUSION

In this paper, we address the problem of interoperability between UVM and UPF which has been missing for more than ten years since the standardization of UPF. The application defined in the paper will allow creation of re-usable UPF aware UVM verification environment and testbenches. While this paper only shows a small portion of the capabilities of the upf_uvm package along with a simple example of integration with the test design, a number of possibilities arise from this work. Essential questions like whether the power-controller working correctly with the UPF provided, isolation clamp values/retention signals/corruption and normal states can easily be answered by structuring extensions of the existing monitors which can obtain a local handle to the upfPowerDomain class and continuously monitor the power state without resorting to the development of a new testbench.

## REFERENCES


[1] IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems," in IEEE Std 1801-2015 (Revision of IEEE Std 1801-2013) , vol., no., pp.1-515, 25 March 2016.

[2] Awashesh Kumar, Madhur Bhargava, Pankaj Gairola, and Vinay K. Singh, “Low Power Apps: Shaping the Future of Low Power Verification,” DVCon 2018.