Verifying a low power design

Asif Jafri

Verilab Inc.
Austin, USA

www.verilab.com

ABSTRACT

User expectations of mobile devices drive an endless race for improvements in both performance and battery life. This paper outlines the verification challenges created by some widely used low power design techniques, and shows how a digital verification methodology can be extended to existing testbenches, enabling low-power designs to be verified.
We describe our experience in using the Unified Power Format to define various power domains and isolation policies, and to control power states from the testbench.
We will also discuss how to structure your verification effort, starting by capturing key low-power features in a testplan, and using appropriate tools and methodology to satisfy your test-plan's requirements so that you can confidently claim that the design has been verified.
This paper will be useful to verification engineers who have begun a challenging low-power design verification project and who wish to apply best-practice techniques that have already shown their practical value.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

This paper discusses our experiences performing power aware verification on an SoC based around multiprocessor clusters. As shown in Figure1, Cluster 0 comprises of a multiprocessor system running at a higher clock frequency, while Cluster 1 is also a multiprocessor system running at a lower clock frequency.

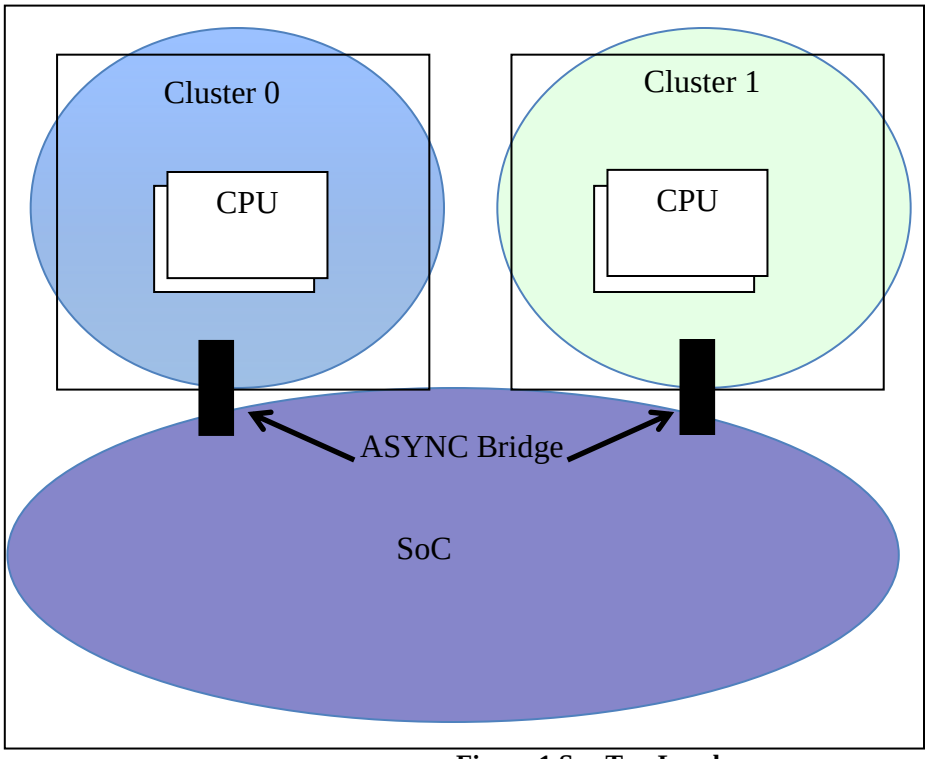

Figure 1 Soc Top Level

The cores had been verified extensively in both functional and power aware simulations before they were used in this system and therefore we will not be talking about their verification effort.

2. Feature list:

As seen in Figure 1, the SoC was divided into three power domains (Cluster 0, Cluster 1 and the rest of the SOC). The requirement was that Cluster 0 and 1 could be completely powered ON or OFF, while the SoC domain will remain powered up while either cluster is powered down and all control registers reside in the SoC domain. A specific sequence of events needs to be followed while powering down or powering up a core, which was provided by the core team.
Table 1 shows the various power domain states for all the cores.

The most interesting cases to test were when either cluster was powered down while the other cluster was used to wait for powerdown and then generate an event to the external power control-

ler. Once the power controller saw the event, it followed the necessary steps to program the registers in the SoC domain to start the powerup process. We decided this was best achieved by developing a power aware simulation strategy.

Table 1 Power Domain States

CLUSTER 0	CLUSTER 1	SOC
OFF	OFF	OFF
ON	ON	ON
OFF	ON	ON
ON	OFF	ON

The next important feature to verify was Frequency Scaling . This is a method to provide a variable amount of energy to a cluster by dynamically changing the operating frequency. The system could generate multiple clocks from the system clock which were then multiplexed to the two clusters. Controlling the input clock frequencies to the multiplexer and which clock to pass to the cluster is controlled by registers in the SoC domain. The simplest solution to verify this was to build a checker that measured the clock frequency. All possible combinations of frequencies were tested to achieve 100% coverage .

Finally a feature called big.LITTLE switching was to be tested. As mentioned before the two CPU clusters were running at different frequencies, so the idea behind big.LITTLE switching was that if the faster core was running a process which was not very compute intensive, this process could be switched to the slower cluster thereby saving power. This was a very difficult problem to verify in simulation as it would require running the whole switching software for multiple days to a week. This task was then verified using emulation and was shown to work as specified.

3. Building the Unified Power Format File (UPF)

A UPF file is defined by the IEEE 1801-2009 standard to describe the power intent of the chip. Three UPF files were provided to us by the implementation team as they are also used by them in the implementation flow. These UPF files were generated providing one UPF file for each cluster and a top level UPF file. The cluster level UPF files were hierarchically scoped into the top level UPF as opposed to using just one UPF file defining the full chip. A single UPF file is hard to read and to debug. Defining separate UPFs as strategy is a clean way to maintain the system level power intent strategy and I would advise doing it this way. In this section we will discuss briefly the various sections of a UPF file.

One analogy to explain the UPF file would be the power wiring in your house. There is a power supply which is connected to different sections (domains) of the house through switches using

wires. The next section briefly describes the various components of a UPF file, and we'll follow this analogy through as we describe the key features of UPF.

Creating supply ports

The Supply ports are like the main supply to the house that feeds into the circuit breaker. There can be more than one supply port defined for a design.

```
create_supply_port VDD_CLUSTER0
create_supply_port VSS_CLUSTER0
```

Creating power domains

Power domains are like the rooms in the house with all of their different components like lights, outlets to power TV's, stereos, etc.. Power domains are used to define a collection of design components that will be controlled by a power supply. In this example we needed to define three such power domains. The code below shows how to define a power domain.

```
create_power_domain TOP
create_power_domain CLUSTER0
create_power_domain CLUSTER1
```

Creating supply nets

The supply nets are like the wiring between the breakers in the house and the main power supply In the example below we are creating a net named net_VDD_CLUSTER0 and this will supply power to the CLUSTER0 power domain.

```
create_supply_net net_VDD_CLUSTER0 –domain CLUSTER0
```

Connecting supply nets to supply ports

The supply nets need to be connected to the supply ports as shown below.

```
connect_supply_net net_VDD_CLUSTER0 –ports {VDD_CLUSTER0}
```

Connecting the supply nets to a power domain

Finally the supply nets need to connect to a specific power domain.

```
set_domain_supply_net CLUSTER0 –primary_power_net
net_VDD_CLUSTE0 –primary_ground_net net_VSS_CLUSTER0
```

Creating power switches

The power switches are like the breakers that power the rooms with the lights and outlets..

```
create_power_switch main_sw \
     -domain CLUSTER0 \
     -input_supply_port {in VDD_CLUSTER0} \
     -output_supply_port {out VDD_O_CLUSTER0} \
     -control_port {inst_on inst_on} \
     -on_state {state2001 in {!list_on}}
```

Isolation

Once the cores are switched off, their outputs will not be driven any more which cause issues with X's propagating into other blocks. To fix this problem, isolation cells are added to the design using the commands below in the UPF file. Continuing with the analogy, we could think of doors in a room as Isolation, as they prevent darkness from getting into another room when closed

```
set_isolation CLUSTER0_SIG –domain CLUSTER0
```

Creating a hierarchichal scpope for UPF files

To define the hierarchical scope of UPF files you can add the following command to the top level UPF:

```
set_scope <PATH TO UPF FILE>
```

As we were loading the various UPF files into the top level UPF file the following command was used:

```
load_upf  $env(DESIGN_ROOT)/upf/CLUSTER0.upf  -scope <path to
cluster0 core>
```

The UPF file is a Tcl based file, so $env() picks up the predefined environment variable "DESIGN_ROOT".

Creating Level Shifters

The Level shifters are used to vary the voltage levels to the various devices in the house. There might be some devices that need as 120V supply while others run at 220V. We did not have to use level shifters in this project, as the requirement was only to check that the various domains could be powered up and down.

Figure 2 shows us all the components used in a UPF file and ties it to the basic analogy we have been walking through.

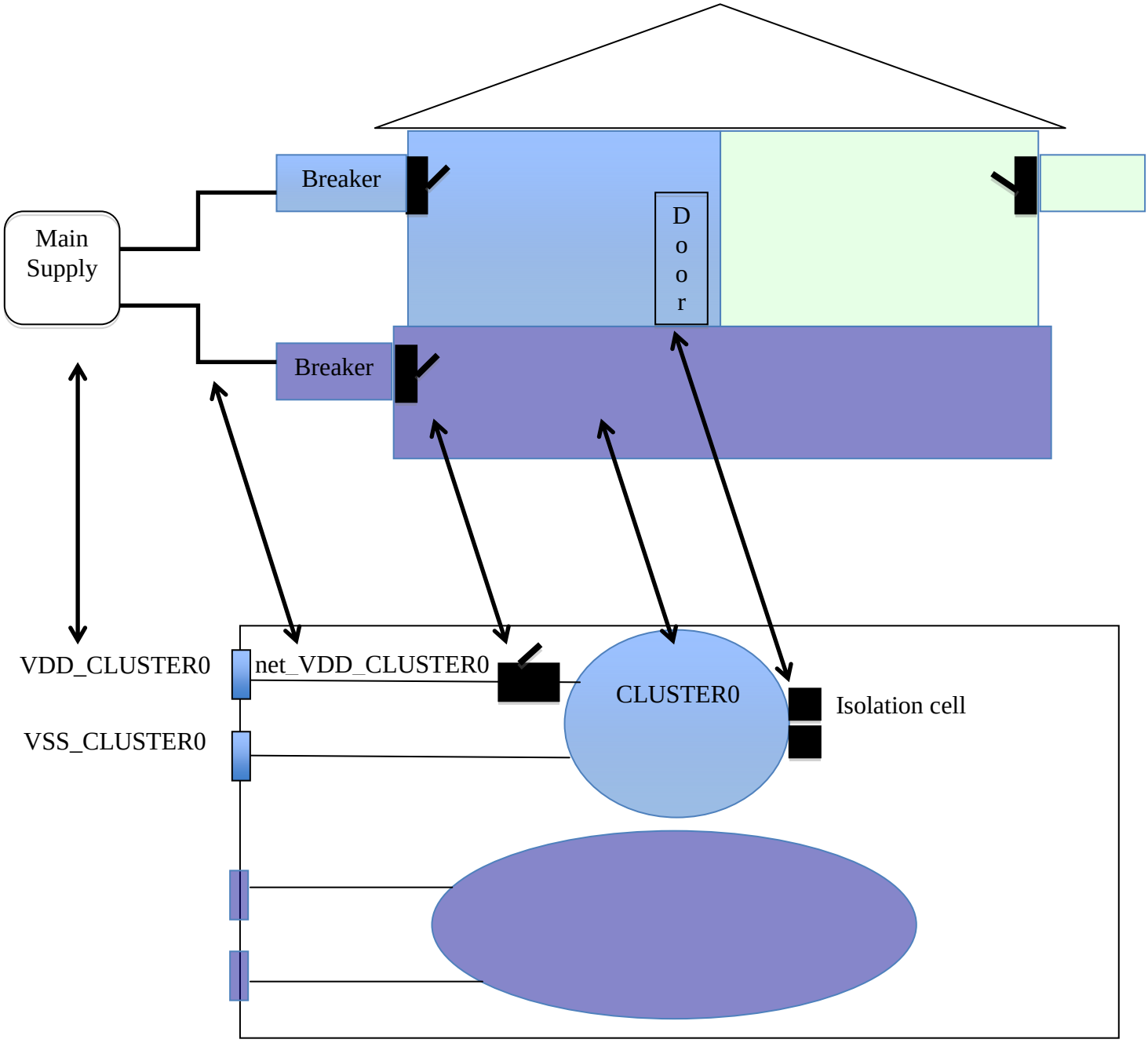

Figure 2 UPF Components

4. Power controller

The existing verification environment had tasks available to program the various system registers and were re-used to create power sequences. A new power controller had to be created in the testbench that monitored the states of the power controller register and the power state output

from the chip. This controller had a simple state machine to mimic the prescribed sequence to power up and power down a specific cluster. To make the controller power aware we needed to import a UPF package that defines some key tasks such as `supply_on` and `supply_off`. These tasks take the supply ports and voltages as arguments and are used to control voltage to the different domains.

```
import UPF::*;


initial begin
  supply_on ("VDD_CLUSTER0", 1.2);
end

state machine for power down sequence;
supply_off("VDD_CLUSTER0");
```

The power controller was the most troublesome part of the testbench to develop. The design was not following the sequence of events to power down the cluster as defined by the architect and most bugs were found here.

5. Tests and checks

To run power aware simulations the following switches need to be added to the VCS command line:

```
vcs –upf <top level upf file> -power_top <module scope for top
level upf> -lp_assert
```

Here are some of the objectives we were trying to achieve with the power aware tests:

1. Make sure the core context could be saved and restored reliably without corruption.
2. The isolation was done properly and it did not propagate X's to the SoC .

All tests written required a set of common steps to power down a cluster. While some of the steps were performed in the test, others were performed by the power controller as it had access to the system registers through an external interface and could monitor the status of the entire system. Having all the status registers was very beneficial as they acted as the synchronizing block between the instructions running in the core and the external power controller. They also showed us the exact state of the entire system.

Steps to power down the cluster:

- Save context.
- Clear and Invalidate caches (controlled by the test).
- Put the CPU's in standby mode in the test.
- Power down asynchronous bridge between SoC using the power controller.
- Power controller asserts reset to the cluster by writing to a control register.

- Power controller will isolate the cluster being powered down by writing to a control register.
- Finally, the power controller removed power.

To power up the steps were:

- Enable power.
- Supply the appropriate clocks to the cluster
- Remove isolation
- Remove reset
- And finally, the boot code can restore the context saved earlier.

Here is a list of some of the tests written

Basic Power up and power down test for each cluster

This test followed all the steps mentioned above. The checks performed were mainly monitoring all status registers in the SoC domain, which were covered with both code coverage and functional coverage. Most existing assertions were reused.

Power up and power down with context save and restore

This test was the same as the previous test but before the cluster was powered down, all the data, control, and status registers etc. were stored to the DDR memory in the SoC domain. Once the core was powered up again, the values were restored and checked. Code and functional coverage were used to make sure all features were covered.

Random Power up and down

In this test, random power down and power up sequences were created to exercise the control logic and see if any corner cases could be hit.

Checking: Extensive assertions were available to make sure all the signals coming out of the clusters followed protocol and no X propagation happened. All these assertions could be re-used in the SoC domain to make sure no signals were driven to X when a specific cluster was powered down. The assertions in the cluster being powered down were automatically disabled as their sampling clock is shut down during powerdown.
New assertions were written to make sure a clock in a powered down cluster was not running. There is a new switch "-lp_assert" that can be added to the VCS command line to enable assertions to check things like isolation and retention control sequences. These assertions help you with issues like "did the isolation happen before turning power off?".

```
Property check_power;
  @(Cluster0_clk) (cluster0_pwr_ctrl === 1'b1);
endproperty
```

The above assertion was used to monitor that the clock was switched off to cluster0 when power was switched off. Similar assertions were written to check reset states in the various domains.

6. Conclusions

Power aware simulations can be a very powerful tool to add to your verification methodology for low power designs. That being said, as shown in this paper we need to properly lay out the various features to be tested and then come up with a strategy to verify each feature with the tools available. Dynamic Voltage Frequency Scaling and big-LITTLE switching are used to reduce power consumption, but were best verified using SystemVerilog checkers in simulation and emulation respectively. Power aware verification was very useful for checking cluster power states. We were able to reuse most of the existing testbench components for the power aware simulations and add the power controller and UPF files on top.

7. References

[1] Michael Keating, David Flynn, Robert Aitken, Alan Gibbons, Kaijian Shi "Low Power Methodology Manual For System-on-Chip Design", Springer, www.springer.com

[2] "IEEE Standard for Design and Verification of Low Power Integrated Circuits" IEEE Computer Society, IEEE Std 1801-2009

[3] David J Lee, personal communication