ZeBu® Power Aware Verification User Guide

Version V-2024.03-1, July 2024

SYNOPSYS®

Copyright and Proprietary Information Notice

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Feedback

Contents

Preface

This chapter has the following sections:

About This Book

The *ZeBu® Power Aware Verification User Guide* describes how to use Power Aware Verification in ZeBu environment, from the source files to runtime.

Audience

This guide is written for experienced ZeBu users who are familiar with the Unified Power Format (UPF), which is described in IEEE 1801-2009 standard. Also, the ZeBu users know how to compile and run a design with a C++ or zRci testbench.

Contents of This Book

The *ZeBu® Power Aware Verification User Guide* has the following chapters:

Chapter	Describes...
ZeBu Power Aware Emulation Flow	ZeBu UPF compilation flow.
UPF Commands Supported by ZeBu	List of UPF commands used for Power Aware verification.
Emulation Runtime for Power Aware Verification	ZeBu runtime requirements and methods used for Power Aware verification.
Runtime Flow for UPF	Runtime flows for Full UPF and Lightweight UPF.

Chapter	Describes...
Limitations of ZeBu Power Aware Verification	Limitations of ZeBu Power Aware verification.
Investigating Power Bugs with Power Aware Verification	Ways to investigate issues with Power Aware verification.
Troubleshooting Power Aware Verification	List of errors reported during Power Aware verification and the solution to resolve them.
Verdi Power Aware Debug	Describes how Verdi enables power aware debug for ZeBu.

Related Documentation

Document Name	Description
ZeBu User Guide	Provides detailed information on using ZeBu.
ZeBu Debug Guide	Provides information on tools you can use for debugging.
ZeBu Debug Methodology Guide	Provides debug methodologies that you can use for debugging.
ZeBu Unified Command-Line User Guide	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
ZeBu UTF Reference Guide	Describes Unified Tcl Format (UTF) commands used with ZeBu.
ZeBu Functional Coverage User Guide	Describes collecting functional coverage in emulation.
Simulation Acceleration User Guide	Provides information on how to use Simulation Acceleration to enable cosimulating SystemVerilog testbenches with the DUT
ZeBu Verdi Integration Guide	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
ZeBu Runtime Performance Analysis With zTune User Guide	Provides information about runtime emulation performance analysis with zTune.
ZeBu Custom DPI Based Transactors User Guide	Describes ZEMI-3 that enables writing transactors for functional testing of a design.
ZeBu LCA Features Guide	Provides a list of Limited Customer Availability (LCA) features available with ZeBu.
ZeBu Synthesis Verification User Guide	Provides a description of zFmCheck.

Document Name	Description
ZeBu Transactors Compilation Application Note	Provides detailed steps to instantiate and compile a ZeBu transactor.
ZeBu zManualPartitioner Application Note	Describes the zManualPartitioner feature for ZeBu. It is a graphical interface to manually partition a design.
ZeBu Hybrid Emulation Application Note	Provides an overview of the hybrid emulation solution and its components.

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	`OUT <= IN;`
Object names	`OUT`
Variables representing objects names	`<sig-name>`
Message	Active low signal name '<sig-name>' must end with _X.
Message location	`OUT` <= IN;
Reworked example with message removed	`OUT_X` <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry
{ } (Curly braces)	An entry that can be specified once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
... (Horizontal ellipsis)	Other options that you can specify

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

1

ZeBu Power Aware Emulation Flow

This section describes the ZeBu UPF compilation flow. See the following subsections:

UPF Support in Unified Compile

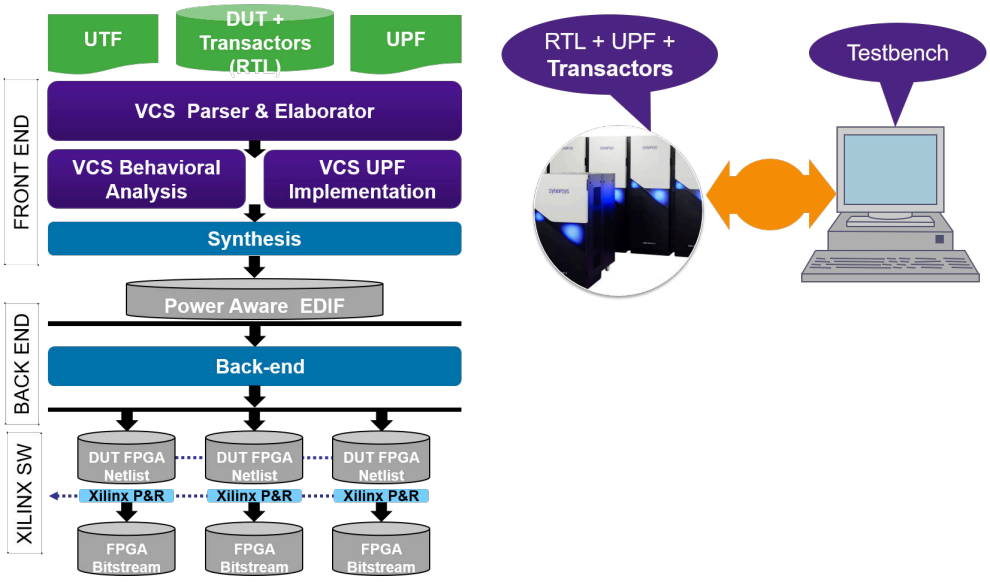
The UC flow supports the same UPF syntax, UPF command support, and error messaging as VCS. The same UPF file is used for both simulation and emulation. When compiling for ZeBu, VCS parses and elaborates both the design and UPF files.

The ZeBu front-end compiler generates the EDIF files, containing design and power intent information. The EDIF files and Core Definition Files are processed by **zTopBuild** in the back-end compilation. The Xilinx Place and Route software generates the final bitstream files that are downloaded into an FPGA.

The following figure displays the UPF compilation flow.

Feedback

Figure 1 UPF Flow in UC



Benefits

The ZeBu UPF compilation flow provides the following benefits:

- Same semantics and analysis like VCS
- A similar set of supported UPF constructs like VCS
- Early error (RTL/UPF) flagging by VCS
- Precedence rules and implementation matching Synopsys cross-tools like, VCS, DC, and ICC
- Integrated debug capability (ZeBu/VCS/Verdi)

ZeBu Front-End Compilation

In the front-end VCS parses and creates the data-model comprising both functional intent and power intent. After VCS execution the design is synthesized into EDIF.

Design Files

Power Aware Verification with ZeBu uses the same design source files as used for emulation without Power Aware.

Power Management Script (UPF)

The UPF script describes the topology of the power network. This script is an input to the compiler.

ZeBu supports the following UPF (Unified Power Format) versions:

- UPF 1.0 and UPF 2.0 (IEEE 1801-2009 standard)
- Limited set of commands from UPF 2.1 (IEEE 1801-2013 standard)

ZeBu Back-End Compilation Flow

After front-end compilation, the next steps in ZeBu emulation are back-end compilation and FPGA Place and Route. The EDIF generated by front-end compilation and Core Definition Files are processed by **zTopBuild**. The Xilinx Place and Route software generates the final bitstream files that are downloaded into the FPGAs.

ZeBu Power Aware Compilation

The UPF file is specified in the VCS command line (or VCS script) with the `-upf` option:

```
% vcs -upf <filename.upf> <vcs_options> <design_files>
```

You can optionally use `-power=<power options>` to enable additional power features.

Supported Power Options

Power options supported by ZeBu are listed in the following table.

Table 1 Supported Power Options

Power Options	Description
`-power_config <config file>`	Maps libraries such as: • `db_search_path = <db path>` • `db_link library = <.db files>` Passes configurations such as: • `set_design_attributes` • `set_power_black_box` • `set_power_domain_toggle_file`
`-power_top <Top module name>`	Captures module whose instance is design top. This power option can also be used to override the UPF command `set_design_top`.

Feedback

Table 1 Supported Power Options (Continued)

Power Options	Description
`-power=zebu_builtin_assertion`	Enables Low Power assertions
`-power=coverage`	Enables functional coverage of UPF objects
`-power=rtlpg`	Enables PG modeling in the RTL by allowing creation of supply port and supply net in the UPF for ports and nets which are also present in the RTL
`-power=scm_ret`	Enables retention instrumentation in Simon
`-power=scm_mem_ret`	Enables memory retention instrumentation This is hardware friendly instrumentation and does not consider each bit of memory.
`-power=hw_corrupt_boundary`	Selectively enables hardware based boundary gate corruption. Use this along with `power_config` option.
`-power=bmux_for_all_drivers`	Instrument boundary MUX on all the driver of the output port
`-power= disable_boundary_gates`	Stops boundary corruption
`-power=voltage_emulation`	Enables voltage emulation. Supply nets shows voltage values transition.
`-power=pst_emulation`	Enables PST emulation and capture `designState` signal indicating PST state of design.
`-power=optimized_voltage_emulation`	Optimizes voltage emulation by uniquely identifying the supplies
`-power=mergePowerDomain`	Merges equivalent power domains
`-power=ignore_ret_generic_clock`	Ignores the connectivity for generic retention clock and reset
`-power=implicit_bias_connection`	Connects the bias PG pins of a DB cell
`-power=propagateBlackBoxConstant`	Allows constant propagation for black box output ports through low power logic
`-power=upf_tokens`	Generates flat UPF file after VCS elaboration at `zcuiUC.work/vcs_splitter/mvsim_native_reports/tokens.upf`

Feedback

Table 1 Supported Power Options (Continued)

Power Options	Description
`-power=cov_pst`	Enables functional coverage for pst states and transitions
`-power=dumplpelab`	Captures design after PNM transformations Diagnostic option to capture design after PNM transformations
`-power=dumplpconnect`	Captures design after strategy transformation Diagnostic option to capture design after Strategy transformation. `Dumps file : lpconnect.ev`
`-power=ignore_iso`	Diagnoses compile-time performance by eliminating isolation Compile time performance diagnostics by eliminating isolation
`-power=ignore_retention`	Diagnoses compile-time performance by eliminating retention Compile time performance diagnostics by eliminating retention
`-power=ctcmdiag`	Diagnoses compile-time performance Compile time performance diagnostics

Note:

- In case of incorrect behavior of any UPF policy or incorrect instrumentation, recompile the design till VCS stage using `-power=dumplpconnect` and `-power=upf_tokens` and share the output files `lpconnect.ev` and `tokens.upf` generated from these power options respectively.
- In case of compile time degradation, recompile the design till VCS stage using `-power=ctcmdiag` and share the diagnostic reports.

Corruption in ZeBu Power Aware Emulation Flow

ZeBu models corruption of shutdown domain by randomizing its output ports and by scrambling the internal state elements.

Table 2 Randomization and Scrambling in ZeBu

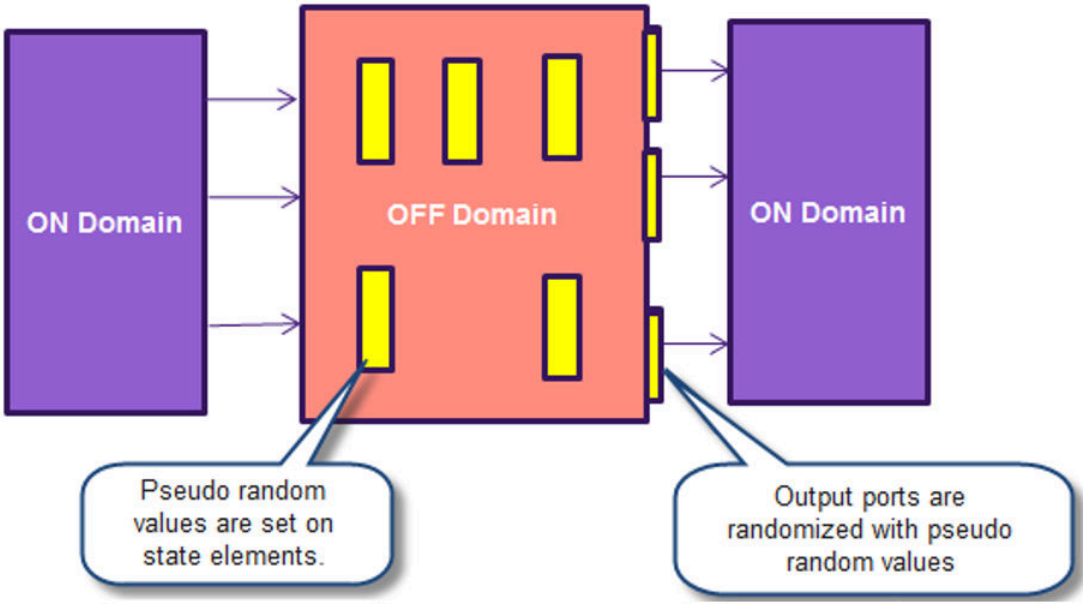
Randomization	Scrambling
Applies on outputs of state elements present at domain boundary ports	Applies on output of internal state elements: registers, latches and memories

Feedback

Table 2 Randomization and Scrambling in ZeBu (Continued)

Randomization	Scrambling
As long as power domain is `off`, output ports are continuously randomized with pseudo values on each clock edge. When isolation supply is `off`, output of isolation cells is continuously randomized	Output of state elements are scrambled only during power domain switching from `on` to `off` and `off` to `on`
Impacts hardware capacity by instrumenting boundary MUX for randomization	No impact on hardware capacity
No impact of wall time	Impacts wall time as it uses `zinject` mechanism which stops the clock to inject pseudo random values. It is proportional to domain switching activity at runtime and number of sequential elements in power domain.
Runtime API available to turn Randomization `on` or `off`	Runtime API available to turn scrambling `on` or `off` and separately control scrambling of registers and memories.

Figure 2 Corruption in ZeBu Power Aware Emulation Flow



Feedback

UPF File Example

```
set_design_top top

create_power_domain TOP -elements {} -include_scope
create_power_domain VCC0 -elements {child0 adder0}
create_power_domain VCC1 -elements {child1 adder1}
create_power_domain VCC2 -elements {adder2}

# VCC
create_supply_port VCC_TOP_port -domain TOP
create_supply_net  VCC_TOP_net  -domain TOP
connect_supply_net VCC_TOP_net  -ports VCC_TOP_port

# VSS
create_supply_port VSS -domain TOP
create_supply_net  VSS_net  -domain TOP
connect_supply_net VSS_net  -ports VSS

## Switch output
create_supply_net  VCC0_SW  -domain VCC0
create_supply_net  VCC1_SW  -domain VCC1
create_supply_net  VCC2_SW  -domain VCC2
######################################################
## Set Domain supplies
######################################################
set_domain_supply_net TOP \

  -primary_power_net VCC_TOP_net \
  -primary_ground_net VSS_net
## Declare that all power sets have a common ground.
create_supply_set VCC0.primary -function {ground TOP.primary.ground}
 -function { power VCC0_SW} -update
create_supply_set VCC1.primary -function {ground TOP.primary.ground}
 -function { power VCC1_SW} -update
create_supply_set VCC2.primary -function {ground TOP.primary.ground}
 -function { power VCC2_SW} -update
create_power_switch VCC0_SWITCH \
  -domain VCC0 \
  -input_supply_port {VCC_TOP_port VCC_TOP_net} \
  -output_supply_port {VCCU_SW VCC0_SW} \
  -control_port {ctrl_sig switch_ctrl_0_reg} \
  -on_state {VCCU_ON VCC_TOP_port {ctrl_sig} } \
  -off_state {VCCU_OFF {!ctrl_sig} }
create_power_switch VCC1_SWITCH \
  -domain VCC1 \
  -input_supply_port {VCC_TOP_port VCC_TOP_net} \
  -output_supply_port {VCCG_SW VCC1_SW} \
  -control_port {ctrl_sig switch_ctrl_1_reg} \
  -on_state {VCCG_ON VCC_TOP_port {ctrl_sig} } \
  -off_state {VCCG_OFF {!ctrl_sig} }
create_power_switch VCC2_SWITCH \
```

Feedback

```
  -domain VCC2 \
  -input_supply_port {VCC_TOP_port VCC_TOP_net} \
  -output_supply_port {VCCG_SW VCC2_SW} \
  -control_port {ctrl_sig switch_ctrl_2_reg} \

  -on_state {VCCG_ON VCC_TOP_port {ctrl_sig} } \
  -off_state {VCCG_OFF {!ctrl_sig} }
#-------------------------------------------------------------------
#                          set isolation strategies
#-------------------------------------------------------------------
name_format -isolation_prefix "ISO_PREFIX_"

set_isolation VCC0_isolation -domain VCC0 \
  -isolation_power_net VCC_TOP_net \
  -isolation_ground_net VSS_net \
  -applies_to outputs \
  -clamp_value 0

set_isolation_control VCC0_isolation -domain VCC0 \
  -isolation_signal switch_ctrl_0_reg \
  -isolation_sense low \
  -location self

set_isolation VCC1_isolation -domain VCC1 \
  -isolation_power_net VCC_TOP_net \
  -isolation_ground_net VSS_net \
  -clamp_value 1

set_isolation_control VCC1_isolation -domain VCC1 \
  -isolation_signal switch_ctrl_1_reg \
  -isolation_sense low \
  -location self

set_isolation VCC2_isolation -domain VCC2 \
  -isolation_power_net VCC_TOP_net \
  -isolation_ground_net VSS_net \
  -clamp_value Z
set_isolation_control VCC2_isolation -domain VCC2 \
  -isolation_signal switch_ctrl_2_reg \
  -isolation_sense low  \
  -location self
#-------------------------------------------------------------------
#                          set retention strategies
#-------------------------------------------------------------------
set_retention VCC0_retention \
  -domain VCC0 \
  -retention_power_net VCC_TOP_net \
  -retention_ground_net VSS_net \
  -save_signal       { save_sig_0_reg high } \
  -restore_signal    { restore_sig_0_reg high } \

set_retention VCC1_retention \
  -domain VCC1 \
```

```
  -retention_power_net VCC_TOP_net \
  -retention_ground_net VSS_net \
  -save_signal       { save_sig_1_reg high } \
  -restore_signal    { restore_sig_1_reg high } \
set_retention VCC2_retention \
  -domain VCC2 \
  -retention_power_net VCC_TOP_net \
  -retention_ground_net VSS_net \
  -save_signal       { save_sig_2_reg high } \
  -restore_signal    { restore_sig_2_reg high } \

#-------------------------------------------------------------------
#                          tools option set
#-------------------------------------------------------------------
set_design_attributes -attribute {SNPS_reinit TRUE}
```

Low Power Optimization and Reporting Commands

ZeBu back-end provides multiple low-power optimization. Enable or disable these commands depending on low power verification requirements. Specify this behavior using the **zTopBuild** advanced command `config_upf`:

```
ztopbuild -advanced_command {config_upf <options>}
```

Table 3 Low Power Optimization and Reporting Commands

config_upf Options	Description
`-enable_clk_cone_corruption`	Enables boundary MUXes in the clock cone.
`-share_randomizers`	`<yes|no>` Enables using one randomizer for 4 ports.
`-zc_upf_modify_seq_enable`	Modifies enable pin of sequential element to support Retention without clock gating in Lightweight UPF flow.

Lightweight UPF (LW UPF)

Lightweight UPF is a custom mode for power aware verification. While corruption is a critical part of the complete LP testing, user survey indicates 90%+ of emulation content could be verified “with no corruption”. Lightweight UPF provides a solution which has close to no hardware overhead (~1%) due to low power instrumentation compared to overall design. The typical UPF overhead for full UPF is 5-20%. Lightweight UPF helps verify retention and isolation functionality without corruption, with no capacity or performance impact.

Feedback

To enable Lightweight UPF, use the VCS elaboration option: `-power=upflite`

At runtime, no harm mode which is default and AON mode are supported. There is no need of a power up sequence, as it does not matter in Always-On mode.

Table 4 UPF Features Supported by ZeBU UPF and ZeBu LW-UPF

UPF Feature	ZeBu UPF	ZeBu LW-UPF
ZeBu UPF	✓	✓ Simplified isolation cell – no corruption support
Retention	✓	✓ Optimized Retention cell implementation No Scrambling
Corruption/Scrambling	✓ (Random /All 1 / All 0)	✗ No Support
Power Domain Switching	✓	✗ No Support
Runtime	Full Control with API No harm/AON/PAE mode Corruption/Scrambling Iso/retention enable/disable	Limited API support No harm and AON mode Iso/retention enable/disable Lighter Runtime Power DB

Automated FWC for critical UPF signals

This feature enables faster root cause analysis and debug by adding hierarchical power domain status signal, supplies and UPF control signals (isolation, retention and power switch controls) automatically to FWC for the whole design.

To enable this feature, add following in the Verilog file where the dumpvars are specified.

```
 initial [ begin : VSET_NAME ]
 (*upf* ) $dumpvars( 0 , <hw_top> );
[ end // VSET _NAME ]
```

2

UPF Commands Supported by ZeBu

UPF commands and options are parsed by VCS.

ZeBu supports UPF 1.0, UPF 2.0, UPF 2.1 and limited set of UPF 3.0 commands.

3

Emulation Runtime for Power Aware Verification

A design compiled with UPF can be emulated at runtime with the same test environment when no Power Aware Verification is required. See the following subsection for more information:

C++ Interface

The C++ API methods are provided in the `PowerMgt` class, which is described in the `$ZEBU_ROOT/include/PowerMgt.hh` header file. This API is included in the ZEBU namespace.

For easier implementation, `PowerMgt.hh` header file is automatically available when including the `libZebu.hh` header file.

The `PowerMgt` APIs must be called during emulation runtime in the following order:

1. Anytime during emulation, call `PowerMgt::isPowerManagementEnabled()` to find whether power aware verification is enabled during runtime.
2. To initialize power aware verification during runtime, call `PowerMgt::init`.
3. To enable power aware verification during runtime, call `PowerMgt::enable`.

Feedback

4. The following APIs must be called before calling `PowerMgt::enable`:
   - `PowerMgt::initializeRandomizer`
   - `PowerMgt::setForceMode`
   - `PowerMgt::setPollingSleepTime`
5. The following APIs must be called after calling `PowerMgt::enable`:
   - `PowerMgt::getPowerDomainState`
   - `PowerMgt::releasePowerDomain`
   - `PowerMgt::getLastTriggeredDomain`
   - `PowerMgt::getSupplyState`
   - `PowerMgt::enableIsolation`
   - `PowerMgt::enableIsolationStrategy`
   - `PowerMgt::isIsolationStrategyEnabled`
   - `PowerMgt::enableRetention`
   - `PowerMgt::enableRetentionStrategy`
   - `PowerMgt::isRetentionStrategyEnabled`
   - `PowerMgt::setDomainOnPreCallback`
   - `PowerMgt::setDomainOnPostCallback`
   - `PowerMgt::setDomainOffPreCallback`
   - `PowerMgt::setDomainOffPostCallback`
   - `PowerMgt::enableSRSN`
   - `PowerMgt::getListOfDomains`
   - `PowerMgt::getListOfIsolationStrategies`
   - `PowerMgt::getListOfRetentionStrategies`
   - `PowerMgt::supplyOn`
   - `PowerMgt::StartWriteBack/FlushWriteBack`

6. The following APIs must be called after *PowerMgt::init* and after or before `PowerMgt::enable`:
   - `PowerMgt::supplyOff`
   - `PowerMgt::setCorruptionState`
   - `PowerMgt::setScramblingState`
   - `PowerMgt::getPowerDomainName`
   - `PowerMgt::setMultiPowerDomainState`
   - `PowerMgt::setPowerDomainState`
   - `PowerMgt::setPowerDomainOn`
   - `PowerMgt::setPowerDomainOff`
   - `PowerMgt::disableIsolation`
   - `PowerMgt::disableIsolationStrategy`
   - `PowerMgt::disableRetention`
   - `PowerMgt::disableRetentionStrategy`
   - `PowerMgt::disableSRSN`

This API provides the following methods:

- To start emulation runtime with Power Aware Verification. See Starting Emulation Runtime with Power Aware Verification.
- To initialize the environment. See Initializing the Environment.
- To get information about the design. See getListOfDomains.
- To get information about retention. See Managing Retention Strategies.
- To control the power domains. See Controlling Power Domains.
- To declare user callbacks. See Declaring User Callbacks.
- To manage forces and injections. See Managing Forces and Injections.

For more information on featured example of a testbench for Power Aware Verification, see C++ Example for Power Aware Verification.

Feedback

Note:

All methods described in this section throw an exception if the pointer to the ZeBu board is incorrect. For any other failure, these methods return a Boolean value:

- `true` for a correct processing.
- `false` in case of error.

The following table lists the C++ API methods to control power aware verification. These methods are described in this section.

Table 5 C++ API to Control Power Aware Verification: PowerMgt Class

C++ Methods	Description
`Init`	Starts Power Aware Verification.
`Enable`	Enables Power Aware Verification.
`initializeRandomizer`	Initializes the generator that later applies values to registers, ports and memories in one or all power domains.
`isPowerManagementEnabled`	Checks if the design is compiled to support Power Aware Verification.
`getListOfDomains`	Returns a list of all power domains declared in the Power Management (UPF) Script.
`getPowerDomainState`	Returns the state of a power domain.
`getLastTriggeredDomain`	Returns the list of power domains that changed state (*ON→OFF* or *OFF→ON*).
`enableRetentionStrategy`	Enables a retention strategy.
`disableRetentionStrategy`	Disables a retention strategy.
`isRetentionStrategyEnabled`	Returns information about the retention strategy.
`getListOfRetentionStrategies`	Returns the list of available retention strategies
`setPowerDomainOn`	Switches a power domain *ON*.
`setPowerDomainOff`	Switches a power domain *OFF*.
`setPowerDomainState`	Switches a power domain to the given state.
`releasePowerDomain`	The domain is no longer controlled from the testbench, but by the design itself.

Feedback

Table 5 C++ API to Control Power Aware Verification: PowerMgt Class (Continued)

C++ Methods	Description
`setMultiPowerDomainState`	Switches multiple power domains to the given state.
`setPowerDomainOffCallback`	Designates a replacement function for the default behavior of the software when a power domain is switched *OFF*.
`setPowerDomainOnCallback`	Designates a replacement function for the default behavior of the software when a power domain is switched *ON*.
`setForceMode`	Defines the behavior of forces and injections regarding power domain states.

Note:

For legibility purposes, `<method_name>` is often used in this chapter in place of `PowerMgt::<method_name>`.

Starting Emulation Runtime with Power Aware Verification

Power Aware Verification must be started with the following methods in the following order:

1. `init(Board*);`
2. `enable(Board*);`

The `PowerMgt::init` method must be called after the `Board::open` and before the `Board::init` methods.

Note:

If your design is compiled with a Power Management script, but you do not want to enable power aware verification during runtime, the `init` and `enable` methods can be omitted.

For example:

```
Board* zebu = Board::open(workdir);
PowerMgt::init(zebu);
zebu->Board::init();
PowerMgt::enable(zebu);
```

Initializing the Environment

The environment is initialized to define the randomizer mode. This section describes the commands to initialize the environment. See the following commands:

- initializeRandomizer
- isPowerManagementEnabled
- getListOfDomains
- getPowerDomainState
- getLastTriggeredDomain

initializeRandomizer

This method initializes the random generator that applies values to registers, ports, and memories in shutdown power domains. Three different modes are available: pseudo-random values (to simulate X values), all `-0` values, or all `-1` values.

```
bool initializeRandomizer (Board *board, const string &mode, const
 unsigned int seedValue) throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object.
- `mode`: Type of randomization; the following values are supported:
  - `MODE_ZERO`: Forces all elements to value `0`.
  - `MODE_ONE`: Forces all elements to value `1`.
  - `MODE_RND`: Forces all elements to a pseudo-random value.
  - `seedValue`: Integer value to initialize the pseudo-random generator.

    Note:

    All `-0` and all `-1` values are only applicable to state elements (registers, latches, and memories). They do not apply to the power-domain's interface ports.

isPowerManagementEnabled

This method checks whether the design is compiled to support Power Aware Verification.

```
bool isPowerManagementEnabled (Board *board, unsigned int &enabled)
 throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object.
- `enabled`: Capability to run with power aware verification (reference to object).

---

getListOfDomains

This method returns a list of all power domains created by the Power Management Script.

```
bool getListOfDomains(Board *board, std::set<std::string> &domains)
```

where,

- `board`: Pointer to the `ZeBu::Board` object.
- `domains`: List of domains names (reference to board).

---

getPowerDomainState

This method provides the state of a power domain.

```
bool getPowerDomainState (Board *board, const std::string &domainname,
 unsigned int &state) throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object.
- `domainname`: Name of the power domain declared in the Power Management Script
- `state`: Pointer to the current state of `domainname`; `1` stands for `ON` and `0` stands for `OFF`.

---

getLastTriggeredDomain

This method retrieves the list of power domains for which the state changed (ON→OFF or OFF→ON).

```
bool getLastTriggeredDomain(Board *board, std::set<std::string>
 &triggerChangedDomain);
```

where,

- `board`: Pointer to the `ZeBu::Board` object.
- `triggerChangedDomain`: Names of domains whose power state changed (reference to object).

Managing Retention Strategies

At runtime, you can turn retention strategies on or off for analysis or performance tuning. This section describes the following methods to manage retention strategies:

- enableRetentionStrategy
- disableRetentionStrategy
- isRetentionStrategyEnabled
- getListOfRetentionStrategies

enableRetentionStrategy

This method enables the retention strategy.

```
bool enableRetentionStrategy (Board *board, const std::string
 &strategyName) throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object.
- `strategyName`: Name of the retention strategy.

disableRetentionStrategy

This method disables the retention strategy.

```
bool disableRetentionStrategy (Board *board, const std::string
 &strategyName) throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object.
- `strategyName`: Name of the retention strategy.

isRetentionStrategyEnabled

This method retrieves information about the retention strategy.

```
bool isRetentionStrategyEnabled (Board *board, const std::string
 &strategyName, bool &enabled) throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object.
- `strategyName`: Name of the retention strategy.
- `enabled`: Capability to see if a retention strategy is enabled.

getListOfRetentionStrategies

This method retrieves the list of available retention strategies.

```
bool getListOfRetentionStrategies (Board *board, std::set<std::string>
 &strategies) throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object.
- `strategies`: List of retention strategies sorted in alphabetical order.

Controlling Power Domains

The methods described in this section provide runtime control for turning power domains `ON` or `OFF`. This is another way of testing the low power behavior of the design.

- setPowerDomainOn
- setPowerDomainOff
- setPowerDomainState
- releasePowerDomain
- setMultiPowerDomainState

setPowerDomainOn

This method switches a power domain *ON* (the power domain is no longer controlled by the design).

```
bool setPowerDomainOn (Board *board, const std::string &domainname)
 throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object
- `domainname`: Name of the domain (reference to object)

setPowerDomainOff

This method switches a power domain `OFF` (the power domain is no longer controlled by the design).

```
bool setPowerDomainOff (Board *board, const std::string &domainname)
 throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object
- `domainname`: Name of the domain (reference to object)

setPowerDomainState

This method switches a power domain to the given state (the power domain is no longer controlled by design).

```
bool setPowerDomainState (Board *board, const std::string &domainname,
const unsigned int state) throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board object`
- `domainname`: Name of the domain (reference to object) `getListOfDomains()`
- `state`: Integer value that specifies the state (0 for `off`, any non-zero value for `on`)

Note:

`setPowerDomainState (board, domainname, 1)` is equivalent to `setPowerDomainOn(board, domainname).`

`setPowerDomainState (board, domainname, 0)` is equivalent to `setPowerDomainOff(board, domainname)`

releasePowerDomain

This method designates the domain to be controlled by design, and no longer by the testbench.

```
bool releasePowerDomain (Board *board, const std::string &domainname)
 throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object
- `domainname`: Name of the domain (reference to object)

setMultiPowerDomainState

This domain switches a list of multiple power domains to the given state (the power domains in the list are no longer controlled by design).

```
static bool setMultiPowerDomainState (Board *board,
     const std::set<std::string> &domains, const unsigned int state);
```

where,

- `board` : Pointer to the `ZeBu::Board` object
- `domains` : Set of domain names, for example populated by `getListOfDomains()`
- `state` : Integer value that specifies the state (`0` for `off`, `1` for `on`)

Declaring User Callbacks

These methods enable you to override the default behavior of a power domain when it changes state from `ON` to `OFF` or vice-versa.

- setPowerDomainOffCallback
- setPowerDomainOnCallback

setPowerDomainOffCallback

By default, ZeBu sets all registers/memories and all ports of a power domain to pseudo-random values when the domain is switched *OFF*.

This method is used to specify a function that overrides the default behavior when a power domain is switched `OFF`.

Note:

Pseudo-random values are applied to ports with a user-callback as well. The user-callback only applies to registers and memories.

```
bool setPowerDomainOffCallback (Board *board, const std::string
 &domainname, void (*callback)(void *), void *userData)
 throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object
- `domainname`: Name of the domain (reference to object)
- `callback`: Pointer to the callback function declared by the user
- `userData`: Pointer to the data structure transmitted to the user callback

The default behavior applies when the `setPowerDomainOffCallback()` method is not called by the testbench or when it is called with as null function, as `setPowerDomainOffCallback(board, NULL, NULL)`.

setPowerDomainOnCallback

By default, ZeBu sets all registers and memories of a power domain to pseudo-random values to prevent power-on to restart with a previous (and valid) state.

This method is used to specify a function that overrides the default behavior when a power domain is switched `ON`.

```
bool setPowerDomainOnCallback (Board *board, const std::string
 &domainname, void (*callback)(void *), void *userData)
 throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object
- `domainname`: Name of the domain (reference to object)
- `callback`: Pointer to the callback function declared by the user
- `userData`: Pointer to the data structure transmitted to the user callback

The default behavior applies when the `setPowerDomainOnCallback()` is not called by the testbench or when it is called with a null function as `setPowerDomainOnCallback(board, NULL, NULL)`.

Managing Forces and Injections

Forcing a signal means to set a user-defined value at runtime until explicitly released. Injecting a signal means to set a user-defined value at runtime, which is overwritten by

the design at the next write operation. The `setForceMode` method defines the behavior of forces and injections of power domain states:

- If set to `0` (default mode), the power domain state is considered when applying forces and injections.
- If set to `1`, forces and injections are applied without considering the power domain states.

```
bool setForceMode(Board *board, unsigned int mode) throw(std::exception);
```

where, `board` is the pointer to the `ZeBu::Board` object.

C++ Example for Power Aware Verification

The following example displays a C++ testbench for Power Aware verification after initialization of the ZeBu board:

```
using namespace ZEBU;

// Initialize generator of pseudo-random values
PowerMgt::initializeRandomizer(zebu, "MODE_RND", 42);

// Run the testbench with power methods activated
top_ccosim->run(cycle);

// force the signal top.regs.d2 to the value "2"
unsigned int value2 = 0;
Signal::Force(zebu, "top.regs.d2", &value2);
[...]
Signal::Release(zebu, "top.regs.d3");
top_ccosim->run(cycle);
// display the power domains whose state has changed
std::set<std::string> domains;


PowerMgt::getLastTriggeredDomain(_zebu, domains);

for (std::set<std::string>::const_iterator dit = domains.begin();
      dit != domains.end(); ++dit)
{
  const std::string& domain = *dit;
  std::cerr << "Domain " << domain << " has switched" << std::endl;
}
```

zRci interface

The following table describes the usage of the power management commands.

Feedback

Table 6 Power Management Commands

Syntax	Description
`powermgt -enable|-disable`	Enables or disables the power management feature.
`powermgt -corrupt -enable|-disable`	Enables or disables the corruption feature.
`powermgt -domain -enable|-disable|-release [<domain list>]`	Lists the power domains to act upon. If omitted, all domains are affected. • `-enable`: Forces domains to turn on. • `-disable`: Forces domains to turn off. • `-release`: Stops domains being forced. • `-state`: Checks the power status of domains.
`powermgt -domain -list|-last_fired`	• `-list`: Returns the list of power domains defined. • `-last_fired`: Prints the power domain activated last.
`powermgt -domain -name <signallist>`	Finds the domain name of each signal
`powermgt -domain -state [<domainlist>]`	`<domain-list>` is the list of power domains to act upon; if omitted, all domains are affected. • `-enable`: Forces domains to turn on. • `-disable`: Forces domains to turn off. • `-release`: Stops domains being forced. • `-state`: Checks the power status of domains.
`powermgt -force random|regular`	Checks if the forced signals must be randomized when the power domain is `off`
`powermgt -isolation -enable|-disable [<strategy-list>]`	`<strategy-list>` is the list of strategies to act upon; if omitted, all strategies are affected. • `-enable`: Activates strategies. • `-disable`: Deactivates strategies. • `-state`: Checks the activation status of strategies.
`powermgt -isolation -list`	Returns the list of defined strategies
`powermgt -isolation -state [<strategy-list>]`	`<strategy-list>` is the list of strategies to act upon; if omitted, all strategies are affected. • `-enable`: Activates strategies. • `-disable`: Deactivates strategies. • `-state`: Checks the activation status of strategies.

Feedback

Table 6 Power Management Commands (Continued)

Syntax	Description
`powermgt -randomizer -init -random|-fixed <value>`	Initializes the randomizer with the given values. The emulated domain signals are randomized on power off, as specified. • `-fixed <value>` must be 0 or 1. • `-random <value>` is the random number generator seed. Default is `-random 0`.
`powermgt -randomizer -run [<domain-list>]`	Runs the randomizer `<domain-list>` is the list of domain names to be randomized. If omitted, all domains are randomized.
`powermgt -retention -enable|-disable [<strategy-list>]`	`<strategy-list>` is the list of strategies to act upon; if omitted, all strategies are affected. • `-enable`: Activates strategies. • `-disable`: Deactivates strategies. • `-state`: Checks the activation status of strategies.
`powermgt -retention -list`	Returns the list of strategies defined in the design.
`powermgt -retention -state [<strategy-list>]`	`<strategy-list>` is the list of strategies to act upon; if omitted, all strategies are affected • `-enable`: Activates strategies. • `-disable`: Deactivates strategies. • `-state`: Checks the activation status of strategies.
`powermgt -scramble -enable|-disable [-all|-registers|-memory]`	Enables/disables power management scrambling for registers and/ or memories. If the option is not provided, it activates/deactivates scrambling for all (registers and memories).
`powermgt -srsn -enable|-disable`	Enables or disables the `set_related_supply_net` feature.
`powermgt -supply -state <pad-list>`	`<pad-list>` is the list of pad names whose supply state is checked.
`powermgt -supplyOff <pad-list>`	`<pad-list>` is the list of pad names to be disabled.
`powermgt -supplyOn <voltage> <padlist>`	Voltage value for pads `<pad-list>` is the list of pad names to be enabled.

Feedback

zRci Example for Power Aware Verification

The following example displays a zRci testbench for Power Aware Verification.

```
set seed $::env(SEED)
config zebu_work {../zcuiUC.work/zebu.work}
global DEFAULT_CLK; set DEFAULT_CLK ufe_top.myclk
#PowerManagement initialization is default in zRci
start_zebu db_FWC
powermgt -randomizer -init -fixed $seed
powermgt -scramble -disable
powermgt -domain -enable ufe_top/top/VCC0 ufe_top/top/VCC1
powermgt -domain -disable ufe_top/top/VCC2
powermgt -force regular
powermgt -enable
ZEBU_Signal_force ufe_top.switch_ctrl_1 1
run 10
powermgt -supplyOff ufe_top.top.addshift1.VCC6_SW
run 10
powermgt -supplyOn 1 ufe_top.top.addshift1.VCC6_SW
……
exit
```

4

Runtime Flow for UPF

This section describes the runtime flows for Full UPF and Lightweight UPF. With UPF, there are multiple modes of operation supported at runtime, as follows:

- **No Harm Mode**: In No Harm Mode, UPF is not enabled at runtime.
- **Always-On Mode**: All the power domains are forced to ON and Scrambling & Corruption are disabled using APIs at runtime. Only the isolation and retention are exercised.
- **Power Aware Enable Mode**: In Power Aware Enable Mode, all the UPF functionality can be fully exercised.

The following table displays the modes supported by Full UPF and Lightweight UPF:

Table 7 Modes Supported by Full UPF and Lightweight UPF

Full UPF	Lightweight UPF
• No Harm Mode • Always-On Mode • Power Aware Enable Mode	• No Harm Mode • Always-On Mode

Model compiled with UPF by default runs in No Harm Mode.

Enabling Runtime for Full UPF and Lightweight UPF

The following flow diagrams show the steps involved in enabling the runtime in different modes of Full UPF and Lightweight UPF.

Figure 3 Full UPF Using C++ Testbench

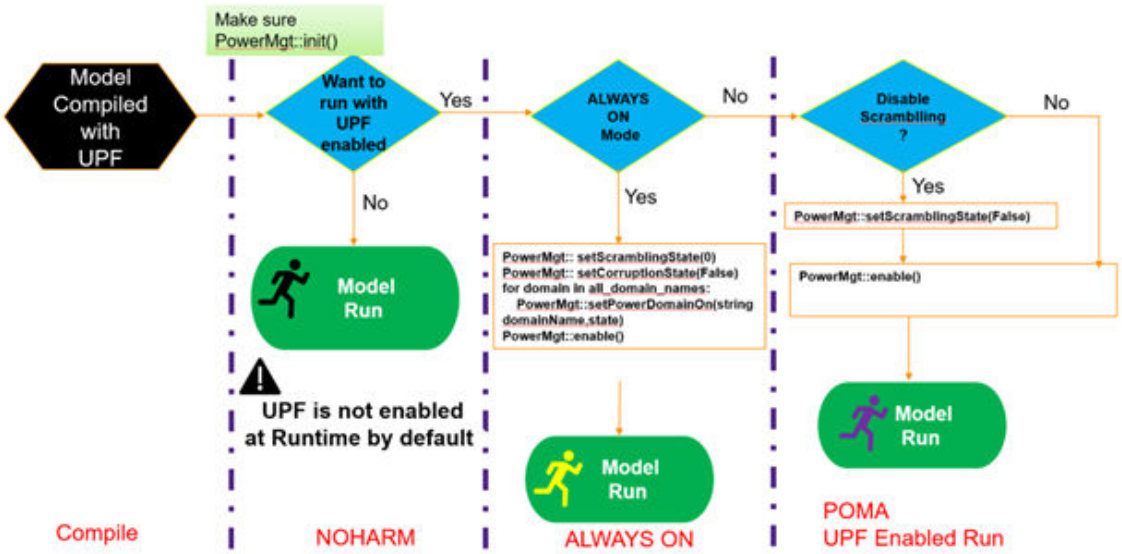


Figure 4 Full UPF Using zRci Testbench

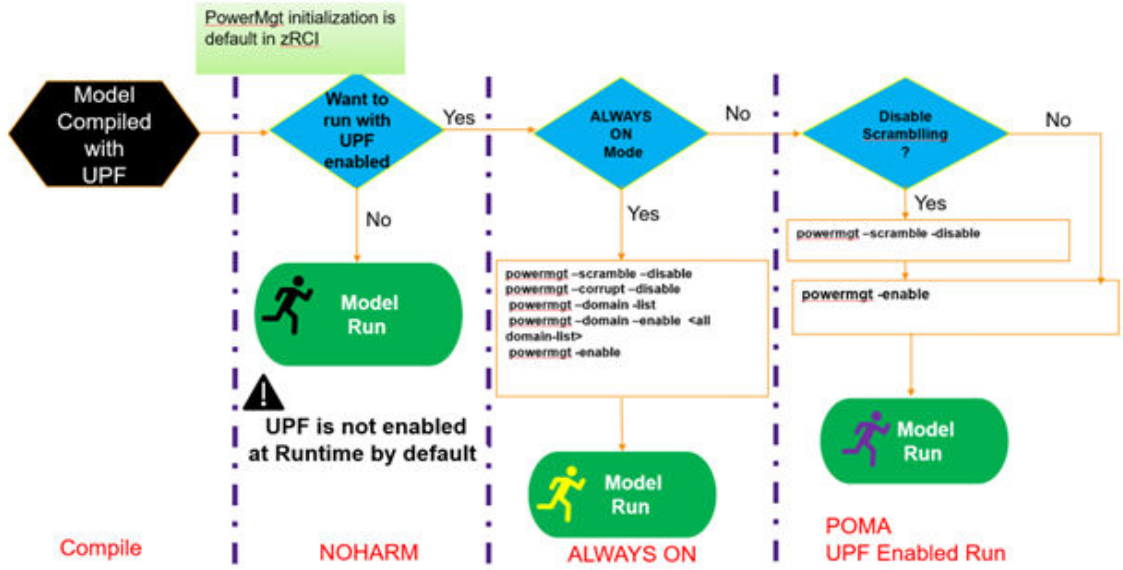
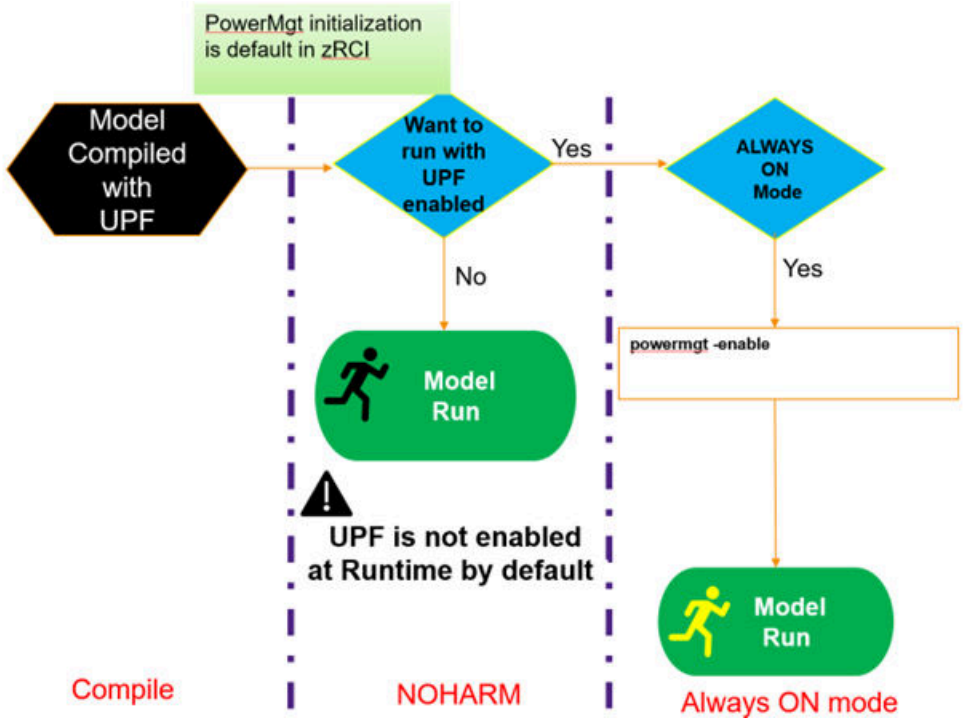


Figure 5 Lightweight UPF Using zRci Testbench



5

Limitations of ZeBu Power Aware Verification

The following features are not supported by the current version of ZeBu Power Aware Verification with:

- Voltage-level (value) shifting - only `ON`/`OFF` is supported.
- Randomization on registers in case of X on control signals of the retention.
- Randomization on isolation output in case of X on control signals of isolation.
- Definitions of tuples (triplets of isolation supply, isolation signal and sense and isolation clamp arguments in an isolation strategy).

6

Investigating Power Bugs with Power Aware Verification

ZeBu provides several means to investigate issues with Power Aware Verification typically found when the design controls the power domains specified in UPF. See the following subsections:

- Checking Isolation between Power Domains
- Examining the State of Power Domains

Checking Isolation between Power Domains

When an unexpected behavior is observed, the issue may be caused by isolation between power domains. In particular, the pseudo-random values applied to the output ports of an `OFF` power domain may cause unexpected values on other `ON` domains not properly isolated. For example, the isolation control is not enabled, or the isolation supply is `OFF`.

These unexpected values must be investigated upstream to locate the corresponding power domain that is switched `OFF`.

Once a particular power domain is identified as the root cause for the isolation problem, it can be manually switched `OFF` using the ZeBu API and then verify the inputs of the other `ON` domains.

The ZeBu C++ API for power aware verification offers specific methods to manually control the activation of power domains, see Controlling Power Domains.

Table 8 C++ and zRci Methods for Power Domains

C++ Method	zRci Method	Description
`setPowerDomainOn`	powermgt -domain -enable	Switches a power domain *ON* (the power domain is no longer controlled by the design).
`setPowerDomainOff`	powermgt -domain -disable	Switches a power domain *OFF* (the power domain is no longer controlled by the design).

Table 8 C++ and zRci Methods for Power Domains (Continued)

C++ Method	zRci Method	Description
`setPowerDomainState`	powermgt -domain -enable|-disable	Switches a power domain to the given state (the power domain is no longer controlled by the design).
`releasePowerDomain`	powermgt -domain -release	The domain is no longer controlled from the testbench but by the design itself.

Examining the State of Power Domains

The ZeBu C++ API for Power Aware Verification offers specific methods to check the properties and state of a power domain, see Controlling Power Domains.

Table 9 Methods to Check the State of Power Domains

C++ Method	zRci Method	Description
`isPowerManagementEnabled`	-	Checks whether the design has been compiled for Power Aware Verification.
`getListOfDomains`	`powermgt -domain -list`	Returns a list of all power domains declared in the UPF Script.
`getPowerDomainState`	`powermgt -domain -state`	Returns the (*ON*/*OFF*) state of a power domain.
`getLastTriggeredDomain`	`powermgt -domain -list_fired`	Returns the list of power domains whose state changed (*ON*→*OFF* or *OFF*→*ON*).

In addition, messages are reported in the runtime logs indicating when power domains change state (`ON`/`OFF`).

7

Troubleshooting Power Aware Verification

The following sections display the error messages reported in the log file of a Power Aware Verification testbench, and solutions to solve each problem.

Incorrect Order when Calling PowerMgt::init

If you do not call the `PowerMgt::init` method before the `PowerMgt::enable` method, the testbench fails with the following error message:

```
-- ZeBu : tb : ERROR : ZHW1027E : Call 'PowerMgt::init' before any call
 to 'PowerMgt::enable'
```

Failure when Calling any Power Aware Method

If you call any of the methods related to the Power Aware Verification feature (see C++ Interface) without compiling the design with this feature, the testbench fails with one of the following error messages:

```
-- ZeBu : tb : ERROR : ZHW1031E : 'init' call failed: Power Awareness
 feature not available
```

Or

```
-- ZeBu : tb : ERROR : ZHW1031E : 'initializeRandomizer' call failed:
 Power Awareness feature not available
```

It is mandatory that you compile your design with the Power Aware Verification feature before attempting to use any of its features.

Failure When Calling any Retention-Related Method

If you call any of the methods related to retention strategies without prior definition of the retention strategy in your UPF file, the testbench fails with one of the following error messages:

```
-- ZeBu: tb: ERROR : ZHW1146E : 'getStrategies' call failed. Power
 Retention is not available
```

Or

```
-- ZeBu: tb: ERROR : ZHW1143E : 'isStrategyEnabled' call failed. Power
 Retention is not available
```

Your UPF file must define the given retention strategies to call any retention-related method on them.

Debug Runtime Freeze

In case of finding a runtime freeze in Power Aware mode, try the following steps one-by-one :

1. Disable scrambling for all registers and memories using runtime API. If runtime doesn't freeze with it then try further disabling either registers or memories.
2. Disable corruption.
3. Set all Power Domains to `ON`.

8

Verdi Power Aware Debug

Verdi enables power aware debug for ZeBu.

Verdi Visualization for Power Debug

Verdi provides the comprehensive design views in:

- Power Aware Design Hierarchy Pane
  - Power domain hierarchy
  - Power domain On/Off information
  - Isolation/Retention/Power Switch information
- Complete schematic views
- Annotated Power intent

Verdi provides the power intent visualization through:

- Power Map
- Impacted signal report

Verdi provides the power aware debug through:

- Power Aware Temporal Flow View
- Power specific waveform debug
  - Isolation/Retention/Domain Off Masking
  - Trace X
- Tracing through schematic

Feedback

Figure 6 Hierarchical Power Domain Pane

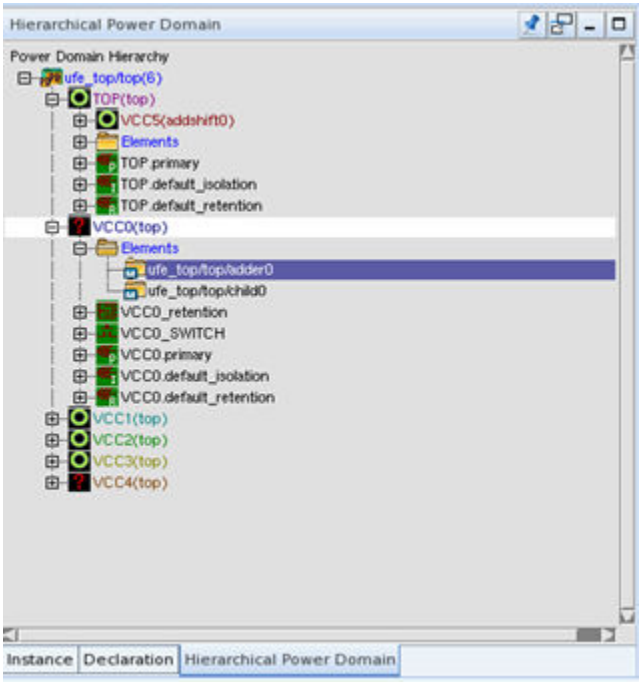
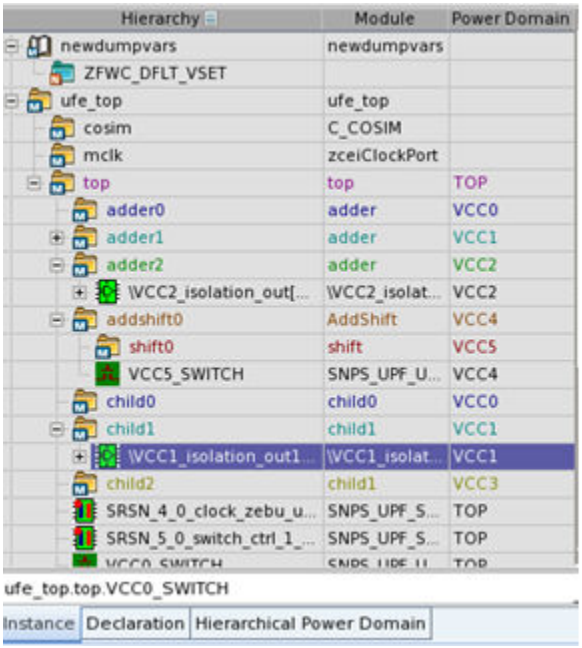
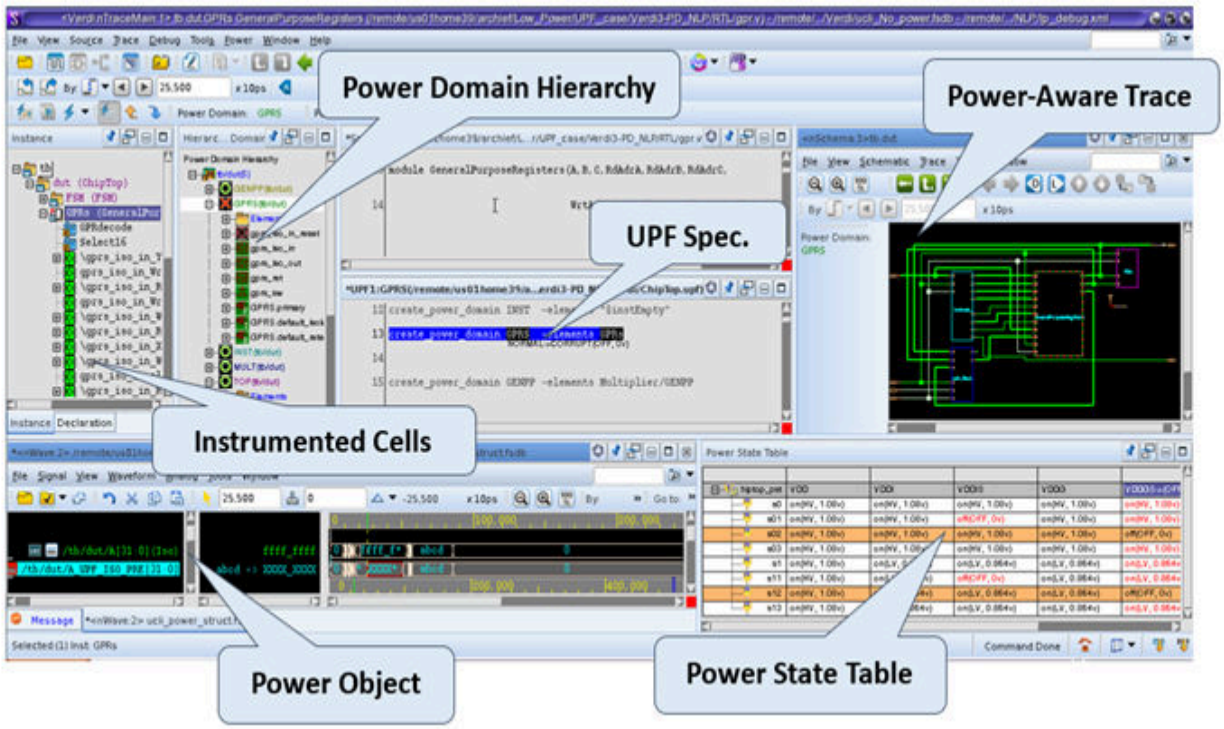


Figure 7 Instance pane showing Instrumented cell in design Hierarchy



Feedback

Figure 8 Power Aware Debug using Verdi

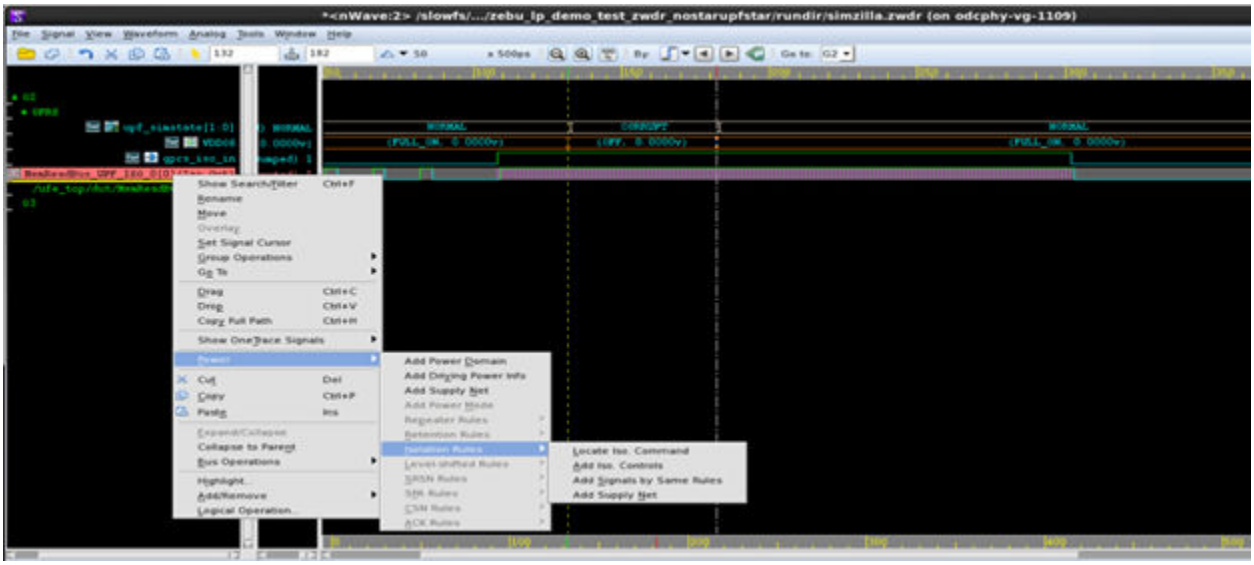


Power Aware Debug Through nWave Window

Any design signal can be checked for related instrumentation details as shown in Figure 6.

Feedback

Figure 9 Low Power instrumentation details on design Signals



nWave supports following kinds of Power Masking or Shading for debug:

- Power Off: The power-off range for HDL signals is masked
- Isolation: The isolation range of HDL signals is masked according to applied isolation condition.
- Retention: The retention range of HDL signals is masked according to applied retention condition.
- Driving Power Off : The driving power-off range for HDL signals is masked.

Feedback

Figure 10 Power masking on Waveform in nWave window