IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 1801™-2015
(Revision of
IEEE Std 1801-2013)

IEEE Std 1801™-2015
(Revision of
IEEE Std 1801-2013)

IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 5 December 2015

IEEE-SA Standards Board

Grateful acknowledgment is made to the following for permission to use source material:

Accellera Systems Initiative

Unified Power Format (UPF) Standard, Version 1.0

Cadence Design Systems, Inc.

Library Cell Modeling Guide Using CPF

Hierarchical Power Intent Modeling Guide Using CPF

Silicon Integration Initiative, Inc.

Si2 Common Power Format Specification, Version 2.1

Abstract: A method is provided for specifying power intent for an electronic design, for use in verification of the structure and behavior of the design in the context of a given power-management architecture, and for driving implementation of that power-management architecture. The method supports incremental refinement of power-intent specifications required for IP-based design flows.

Keywords: bottom-up implementation, buffers, energy-aware design, IEEE 1801™, interface specification, IP reuse, isolation, level-shifting, power domains, power intent, power modeling, power states, successive refinement, supply states, repeaters, retention, Unified Power Format (UPF)

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

PDF: ISBN 978-1-5044-0105-0 STDGT20513
Print: ISBN 978-1-5044-0106-7 STDPD20513

Participants

At the time this IEEE standard was completed, the P1801 Working Group had the following membership:

John Biggs, *Chair*
Erich Marschner, *Vice Chair*
Sushma Honnavara-Prasad, *Secretary*

Houssam Abbas
Paul Bailey
Guillaume Boillet
Conor Byrne
Louis Cardillo
Shir-Shen Chang
David Cheng
Cyril Chevalier
Ashley Crawford
John Decker
Stephan Diestelhorst
Shaun Durnan
Paul Floyd
Jerry Frenkil
Alan Gibbons
Josefina Hobbs
Anand Iyer
Fred Jen
Tim Jordan
Sylvian Kaiser
James Kehoe
Tim Kogel
Rick Koster
Shaji Kunjumohamed
Kaowen Liu
Debajani Majhi
Ilija Materic
Gene Matter
Jon McDonald
Don Mills
Kevin Nesmith
Lawrence Neukom
David Peterson
Shreedhar Ramachandra
Judith Richardson
Frederic Saint-Preux
Rich Scales
Guido Schlothane
Krishna Sekar
Desinghu Pundi Srinivasan
Amit Srivastava
James Su
Haruyuki Tago
Ajay Thiriveedhi
Venki Venkatesh
Vita Vishnyakov
Jon Worthington
Vojin Zivojnovic

The following members of the entity balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Accellera Organization, Inc.
Advanced Micro Devices (AMD)
ALDEC, Inc.
ARM, Ltd.
Broadcom Corporation
Cadence Design Systems, Inc.
Google
Intel Corporation
Japan Electronics and Information Technology Industries Association (JEITA)
Marvell Semiconductor, Inc.
MediaTek, Inc.
Mentor Graphics
Micron Technology, Inc.
Microsoft Corporation
NVIDIA Corporation
PMC-Sierra, Inc.
Qualcomm, Inc.
Silicon Integration Initiative, Inc.
STMicroelectronics
Synopsys, Inc.
Verific Design Automation, Inc.

Introduction

This introduction is not part of IEEE Std 1801™-2015, IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems.

The purpose of this standard is to provide portable, low-power design specifications that can be used with a variety of commercial products throughout an electronic system design, analysis, verification, and implementation flow.

When the electronic design automation (EDA) industry began creating standards for use in specifying, simulating, and implementing functional specifications of digital electronic circuits in the 1980s, the primary design constraint was the transistor area necessary to implement the required functionality in the prevailing process technology at that time. Power considerations were simple and easily assumed for the design as power consumption was not a major consideration and most chips operated on a single voltage for all functionality. Therefore, hardware description languages (HDLs) such as VHDL (IEC 61691-1-1/ IEEE Std 1076™[a])and SystemVerilog (IEEE Std 1800™[b]) provided a rich set of capabilities necessary for capturing the functional specification of electronic systems, but no capabilities for capturing the power architecture (how each element of the system is to be powered).

As the process technology for manufacturing electronic circuits continued to advance, power (as a design constraint) continually increased in importance. Even above the 90 nm process node size, dynamic power consumption became an important design constraint as the functional size of designs increased power consumption at the same time battery-operated mobile systems, such as cell phones and laptop computers, became a significant driver of the electronics industry. Techniques for reducing dynamic power consumption—the amount of power consumed to transition a node from a `0` to `1` state or vice versa—became commonplace. Although these techniques affected the design methodology, the changes were relatively easy to accommodate within the existing HDL-based design flow, as these techniques were primarily focused on managing the clocking for the design (more clock domains operating at different frequencies and gating of clocks when logic in a clock domain is not needed for the active operational mode). Multi-voltage power-management methods were also developed. These methods did not directly impact the functionality of the design, requiring only level-shifters between different voltage domains. Multi-voltage power domains could be verified in existing design flows with additional, straightforward extensions to the methodology.

With process technologies below 90 nm, static power consumption has become a prominent and, in many cases, dominant design constraint. Due to the physics of the smaller process nodes, power is leaked from transistors even when the circuitry is quiescent (no toggling of nodes from `0` to `1` or vice versa). New design techniques have been developed to manage static power consumption. Power gating or power shut-off turns off power for a set of logic elements. Back-bias techniques are used to raise the voltage threshold at which a transistor can change its state. While back bias slows the performance of the transistor, it greatly reduces leakage. These techniques are often combined with multi-voltages and require additional functionality: power-management controllers, isolation cells that logically and/or electrically isolate a shutdown power domain from "powered-up" domains, level-shifters that translate signal voltages from one domain to another, and retention registers to facilitate fast transition from a power-off state to a power-on state for a domain.

The Unified Power Format (UPF) was developed to enable modeling of these new power-management techniques and to facilitate automation of design, verification, and implementation tools that must account for power-management aspects of a design. The initial version of UPF, developed by the Accellera Systems Initiative, focused primarily on modeling power distribution and its effects on the behavior of a system. In

[a] The IEEE standards or products referred to in this clause are trademarks of The Institute of Electrical and Electronics Engineers, Inc.

[b] IEEE publications are available from The Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

May 2007 that initial version was donated to the IEEE, and in March 2009 a new version, IEEE Std 1801, was released. That update of UPF added many new features, including the concept of successive refinement, more abstract modeling of system-power states, and more abstract modeling of supply networks.

This document, the latest revision of IEEE Std 1801, makes available further enhancements to UPF, including enhanced concepts for modeling power states and transitions at all levels of aggregation, enhanced support for methodologies such as successive refinement and bottom-up implementation, and a detailed information model that serves as the basis for enhanced package UPF functions and query functions. This current version also provides support for component power modeling for system-level power analysis in virtual prototyping applications.

Contents

IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems

1. Overview

1.1 Scope

This standard defines the syntax and semantics of a format used to express power intent in energy-aware electronic system design. *Power intent* includes the concepts and information required for specification and validation, implementation and verification, and modeling and analysis of power-managed electronic systems. This standard also defines the relationship between the power intent captured in this format and design intent captured via other formats (e.g., standard hardware description languages and cell libraries).

1.2 Purpose

The standard enables portability of power intent across a variety of commercial products throughout an electronic system design, analysis, verification, and implementation flow.

1.3 Key characteristics of the Unified Power Format

The Unified Power Format (UPF) provides the ability for electronic systems to be designed with power as a key consideration early in the process. UPF accomplishes this by allowing the specification of what was traditionally physical implementation-based power information early in the design process—at the register transfer level (RTL) or earlier. Figure 1 shows UPF supporting the entire design flow. UPF provides a consistent format to specify power-design information that may not be easily specifiable in a hardware description language (HDL) or when it is undesirable to directly specify the power semantics in an HDL, as doing so would tie the logic specification directly to a constrained power implementation. UPF specifies a set of HDL attributes and HDL packages to facilitate the expression of power intent in HDL when appropriate (see Table 4 and 11.2). UPF also defines consistent semantics across verification and implementation to check that what is implemented is the same as what has been verified.

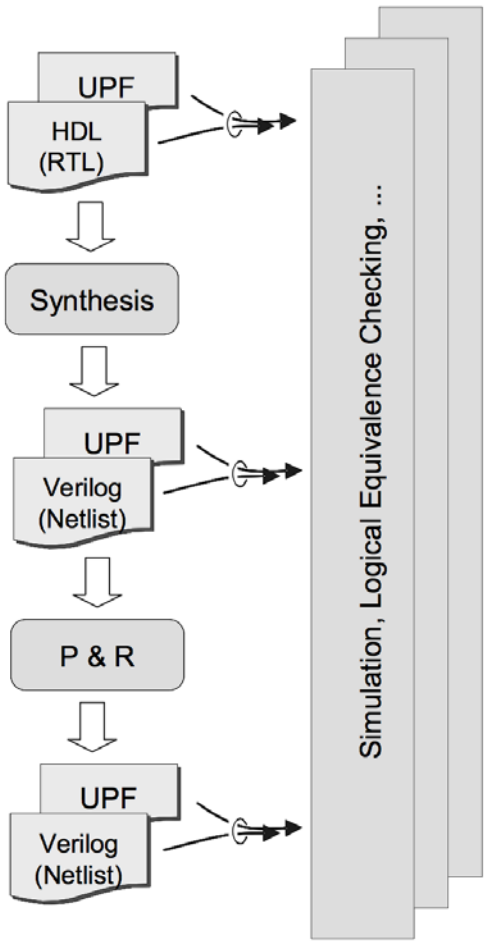


Figure 1—UPF tool flow

As indicated in Figure 1, UPF files are part of the design source and, when combined with the HDL, represent a complete design description: the HDL describing the logical intent and the UPF describing the power intent. Combined with the HDL, the UPF files are used to describe the intent of the designer. This collection of source files is the input to several tools, e.g., simulation tools, synthesis tools, and formal verification tools. UPF supports the successive refinement methodology (see 4.9) where power-intent information grows along the design flow to provide needed information for each design stage.

— Simulation tools can read the HDL/UPF design input files and perform RTL power-aware simulation. At this stage, the UPF might only contain abstract models such as power domains and supply sets without the need to create the power and ground network and implementation details.

— A user may further refine the UPF specification to add implementation-related information. This further-refined specification may then be processed by synthesis tools to produce a netlist and optionally update the UPF fileset accordingly.

— In those cases where design object names change, a UPF file with the new names is needed. A UPF-aware logical equivalence checker can read the full design and UPF filesets and perform the checks to ensure power-aware equivalence.

— Place and route tools read both the netlist and the UPF files and produce a physical netlist, potentially including an output UPF file.

UPF is a concise, power-intent specification capability. Power intent can be easily specified over many elements in the design. A UPF specification can be included with the other deliverables of intellectual property (IP) blocks and reused along with the other delivered IP. UPF supports various methodologies through carefully defined semantics, flexibility in specification, and, when needed, defined rational limitations that facilitate automation in verification and implementation.

1.4 Contents of this standard

The organization of the remainder of this standard is as follows:

— Clause 2 provides references to other applicable standards that are presumed or required for this standard.

— Clause 3 defines terms and acronyms used throughout the different specifications contained in this standard.

— Clause 4 describes the basic concepts underlying UPF.

— Clause 5 describes the language basics for UPF and its commands.

— Clause 6 details the syntax and semantics for each UPF power intent command.

— Clause 7 details the syntax and semantics for each UPF power-management cell command.

— Clause 8 defines a reference model for UPF command processing.

— Clause 9 defines simulation semantics for various UPF commands.

— Clause 10 defines the UPF information model.

— Clause 11 defines the UPF information model application programmable interface (API).

— Annex A lists potentially useful additional reference material.

— Annex B lists the predefined value conversion tables (VCTs) for use in power intent specifications.

— Annex C provides sample Tcl procs for retrieving power intent information.

— Annex D summarizes deprecated and legacy commands.

— Annex E provides an overview of UPF tool flows and use model with an illustrative example.

— Annex F provdes a summary of UPF power-management cell command semantics and Liberty mappings.

— Annex G provides examples of UPF power-management cell modeling.

— Annex H provides an overview of UPF use model for system-level IP power modeling.

— Annex I defines the Switching Activity Interchange Format (SAIF) for representing power-related activity in a design.

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEC 61691-1-1/IEEE Std 1076™, Behavioural languages—Part 1-1: VHDL Language Reference Manual.[10, 11, 12]

IEEE Std 1800™, IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.

ISO/IEC 19501:2005, Information technology—Open Distributed Processing—Unified Modeling Language (UML) Version 1.4.2.

3. Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary Online* [B1][13] should be consulted for terms not defined in this clause.[14] Certain terms in this standard reflect their corresponding definitions in IEEE Std 1800™ or IEC 61691-1-1/IEEE Std 1076™, or they are listed in Annex A.

3.1 Definitions

active component: A component that contains one or more input receivers and one or more output drivers whose values are functions of the inputs, but whose inputs and outputs are not directly connected; or any hardware description language (HDL) construct(s) that synthesize(s) to an active component.

active control signal: A control signal that is currently presenting the value (level) or transition (edge) that enables or triggers an active component to operate in a particular manner.

active power state: A power state whose logic expression—or, in certain cases, supply expression—evaluate to *True* at a given time.

activity: Any change in the value of a net, regardless of whether that change is propagated to an output.

analog port: A port that is part of a connection that delivers analog signals.

ancestor: Any **instance** between the current **scope** in the **logic hierarchy** and its **root scope**. When the current scope is a top-level module, it does not have any ancestors. *See also:* **descendant**.

anonymous object: An object that is not named in the context of Unified Power Format (UPF). Implementations may assign a legal name, but such names are not visible in the UPF context.

[10] ISO/IEC publications are available from the International Electrotechnical Commission (http://www.iec.ch/). IEC publications are also available in the United States from the American National Standards Institute (http://www.ansi.org/).
[11] IEEE publications are available from The Institute of Electrical and Electronics Engineers (http://standards.ieee.org/).
[12] The IEEE standards or products referred to in this clause are trademarks of The Institute of Electrical and Electronics Engineers, Inc.
[13] The numbers in brackets correspond to those of the bibliography in Annex A.
[14] *IEEE Standards Dictionary Online* subscription is available at:
http://www.ieee.org/portal/innovate/products/standard/standards_dictionary.html.

balloon latch: A retention element style in which a register's value is saved to a dedicated latch at power-down and the latch value is restored to the register at power-up.

boundary instance: An **instance** that has no parent or whose **parent** is in a different **power domain**.

child domain (of a **HighConn** port on the **lower boundary** of a **power domain**): The power domain whose **upper boundary** contains the corresponding **LowConn** of the **HighConn port**.

coarse grain switch: A power switch that is used to generate switched supply for a group of library cells. This is identified using the attribute `switch_cell_type: coarse_grain` in Liberty and design attribute **UPF_switch_cell_type coarse_grain** in Unified Power Format (UPF) (see Table 4).

component: A physical and logical construction that relates inputs to outputs.

composite domain: A power domain consisting of subordinate **power domains** called **subdomains**. All subdomains in a composite domain share the same primary supply set.

configuration UPF: A Unified Power Format (UPF) specification of the power-management configuration for a system.

connected: Attached together via a direct connection.

constraint UPF: A Unified Power Format (UPF) specification for an intellectual property (IP) block that defines constraints for any instance of this IP block that must be met by the power-management configuration of the system containing that instance.

correlated: A pair of **supply net**s or a pair of **supply set**s that are deemed to be at the same point in their voltage range when being considered for level-shifting. As such, when voltage levels are considered between them they should be mutually compared, e.g., minimum to minimum and maximum to maximum.

corruption semantics: The rules defining the behavior of logic in response to reduction or disconnection of power to that logic.

current scope: The design hierarchy location that serves as the immediate context for interpretation and execution of Unified Power Format (UPF) commands. Also, the **instance** specified by the **set_scope** command.

declared: Specified in the hardware description language (HDL) explicitly or implicitly via a Unified Power Format (UPF) command.

descendant: Any **instance** between the current scope in the **logic hierarchy** and its **leaf-level instance**s. *See also:* **ancestor**.

descendant subtree: A portion of a **logic hierarchy**, rooted at one **instance** in the hierarchy, and containing that **instance** and all of its **descendant**s.

design hierarchy: A hierarchical structure of nested definitions described in a hardware description language (HDL).

direct connection: A physical wire; or any hardware description language (HDL) construct(s) that synthesize(s) to a direct connection.

domain port: A **port** that is on the interface of a **power domain**.

driver: The source or drain of a transistor, if the drain or source is connected to a **power rail**; a complementary metal oxide semiconductor (CMOS) inverter that continually connects a node to power or ground; any **component** that sets the value of its output via a transistor or inverter; a constant assignment; any combinational logic including a buffer of any kind; any sequential logic; or any hardware description language (HDL) construct(s) that synthesize(s) to such combinational or sequential logic.

driver supply: For a **driver** that is a transistor, the supply connected to its source or drain; for a **driver** that is an inverter, the pair of supplies connected to the source/drain of the transistor pair comprising the inverter; or for an output of an **active component**, the related **supply set** of that output.

electrically equivalent: For **supply port**s/**net**s, connected (whether the connections are evident or not in the design) without any intervening switches, and therefore have the same value at all times from the perspective of any load; for **supply set**s/set handles, consisting of a set of electrically equivalent **supply net**s for each required function.

equivalent: A pair of **supply net**s, a pair of **supply set**s or a pair of **logic nets** that are considered to be interchangeable for certain purposes. *See also:* **electrically equivalent; functionally equivalent**.

erroneous: A usage that is likely to lead to an error in the design, but that tools may not be able to detect and report.

extent (of a domain): The set of instances that comprise a **power domain**.

fanout domain (of a given port to which a given strategy applies): The **power domain** containing any of the following: receiving logic for that **port**, or a **leaf-level cell** instance **HighConn** input **port** that is connected to the given **port**, or a design top module **LowConn** output **port** that is connected to the given port.

feedthrough: A direct connection between two **port**s on the **interface of a power domain**, where the connection involves two **port**s on the upper boundary, or two **port**s on the lower boundary, or one of each; also, a direction connection between two **port**s of the same **leaf-level instance**.

feedthrough port: A **port** on the **interface of a power domain** that is part of a **feedthrough** through that domain, or a **port** on the interface of a **leaf-level instance** that is part of a **feedthrough** through that **instance**.

fine grain switch: A power switch that is used to generate switched supply for a single library cell. This is typically used to describe embedded macro power switches. This is identified using the attribute `switch_cell_type: fine_grain` in Liberty and design attribute **UPF_switch_cell_type fine_grain** in Unified Power Format (UPF) (see Table 4).

functionally equivalent: Functioning identically from the perspective of any load, either as a result of being **electrically equivalent**, or due to independent but parallel circuitry.

generate block: In the hardware description language (HDL) code, this represents a level of design hierarchy, although a generate block is not itself an **instance**. After synthesis, generate blocks do not exist as an independent level of hierarchy. It is illegal to create any Unified Power Format (UPF) objects in a **scope** that corresponds to a generate block.

golden source: The design together with the constraint Unified Power Format (UPF) and the configuration UPF.

hard macro: A block that has been completely implemented and can be used as it is in other blocks. This can be modeled by an hardware description language (HDL) module for verification or as a library cell for implementation.

hierarchical name: A series of names separated by the **hierarchical separator character**, the final name of which is a legal hardware description language (HDL) name or Unified Power Format (UPF) name, and each preceding name is the name of an **instance** or **generate block** in which the following name is **declared**. *See also:* **hierarchical separator character**.

hierarchical separator character: A special character used in composing **hierarchical name**s. The hierarchical separator character is a slash (/).

HighConn: The side of a **port** connection that is higher in the **design hierarchy**; the actual signal associated with a formal **port** definition.

implementation UPF: The Unified Power Format (UPF) specification of how power distribution and control is to be implemented for a system.

inactive: A normally active component in a state in which it does not respond to **activity** on its inputs. Also, a control signal that is not currently presenting the value (level) or transition (edge) that enables or triggers an **active component** to operate in a particular manner.

instance: A particular occurrence of a SystemVerilog module (see IEEE Std 1800), very high speed integrated circuit (VHSIC) hardware description language (VHDL) entity (see IEC 61691-1-1/IEEE Std 1076), or library cell at a specific location within the design hierarchy.

interface of a power domain: The union of the upper boundary and the lower boundary of the **power domain**.

isolation: A technique used to provide defined behavior of a logic signal when its driving logic is not active.

isolation cell: An **instance** that passes logic values during normal mode operation and clamps its output to some specified logic value when a control signal is asserted.

leaf-level cell: An instance that has no descendants, or an instance that is a soft or hard macro.

leaf-level instance: *See:* **leaf-level cell**.

level-shifter: An **instance** that translates signal values from an input voltage swing to a different output voltage swing.

live slave: A retention element style in which the slave latch of a master-slave flip-flop (MSFF) is always on and therefore maintains the value of the MSFF during power-down.

logically equivalent: Logic **ports/nets** that are directly connected without any intervening logic and therefore have the same value at all times from the perspective of any sink.

logic hierarchy: An abstract view of a **design hierarchy** in which only those definitions representing **instance**s are included.

LowConn: The side of a **port** connection that is lower in the **design hierarchy**; the formal **port** definition.

lower boundary (of a power domain): The **HighConn** side of each **port** of each **boundary instance** in the **extent** of another **power domain** whose **parent** is in the **extent** of this domain, together with the **HighConn** side of each **port** of any macro cell instance in this **power domain**, for which the related supply set is neither identical to nor equivalent to the **primary supply set** of this domain.

map: Identify a specific **model** corresponding to an abstract behavior. An **instance** of the **model** can then be used to implement the specific behavior.

model: A SystemVerilog module, very high speed integrated circuit (VHSIC) hardware description language (VHDL) entity/architecture, or Liberty cell.

named power state: A **power state** defined using **add_power_state** for a **supply set**, **power domain**, **composite domain**, group, **model**, or **instance**, including the **power state**s ON and OFF that are predefined for **supply set**s and the **power state**s UNDEFINED and ERROR that are predefined for all objects that have **power state**s.

net: The individual **net segment**s that make up a collection of interconnections between a collection of **port**s. A **net** may be named or anonymous.

net segment: A direct connection within a single **instance**.

parent: The immediate **ancestor** of a given **instance** within the **logic hierarchy**.

parent domain (of a **LowConn port** on the **upper boundary** of a **power domain**): The **power domain** whose **lower boundary** contains the corresponding **HighConn** of the **LowConn** port.

passive component: A direct connection; a **component** that has neither a **receiver** nor a **driver**, whose output is connected to its input, and therefore its output is always the same as its input, e.g., a pass transistor; or any hardware description language (HDL) construct(s) that synthesize(s) to a **feedthrough** component.

pg_type: An attribute of a port that indicates its use in providing power to a cell.

port: A **connection** on the interface of a SystemVerilog module or very high speed integrated circuit (VHSIC) hardware description language (VHDL) entity. Also, a **port** on the **interface of a power domain**.

power domain: A collection of **instance**s that are treated as a group for power-management purposes. The **instance**s of a power domain typically, but do not always, share a **primary supply set**. A power domain can also have additional supplies, including **retention** and **isolation** supplies.

power rail: The physical implementation of a power **supply net**.

power state: A subset of the functional states of an object that have the same characteristics with respect to power supply (for a **supply set**) or power consumption (for a **power domain**, **composite domain**, group, **model**, or **instance**).

power state table (PST): A table that specifies the legal combinations of **supply states** for a set of supply objects (**supply ports**, **supply net**s, and/or **supply set functions**).

primary supply set: The **supply net** connections inferred for all **instance**s in the **power domain**, unless overridden.

receiver: The gate of a transistor; the input to an inverter; any **component** whose behavior is determined by an input signal; any combinational logic including a buffer of any kind; any sequential logic; or any hardware description language (HDL) construct(s) that synthesize(s) to such combinational or sequential logic.

receiver supply: For a **receiver** that is the gate of a transistor, the supply connected to that transistor's source or drain; for a **receiver** that is the input to an inverter, the pair of supplies connected to the source/drain of the transistor pair comprising the inverter; or for a **receiver** that is part of an **active**

component, the primary supply of the **power domain** to which that **receiver** belongs or, in some cases, the secondary supply of the **component** if it has a secondary supply.

regulator: An **instance** that takes a set of input **supply net**s and provides the source for a set of output **supply net**s. The output voltage is a function of the input voltages and the logical state of any control signals.

retention: Enhanced functionality associated with selected **sequential element**s or a memory such that memory values can be preserved during the power-down state of the primary supplies.

retention register: A register that extends the functionality of a **sequential element** with the ability to retain its memory value during the power-down state.

rooted name: The **hierarchical name**, relative to the **current scope**, of an object in the **logic hierarchy** or a Unified Power Format (UPF) object defined for a **scope** in the hierarchy.

root scope: The topmost scope in the **logic hierarchy**, which contains an implicit instance of each top-level module.

root supply driver: The origin of a supply, e.g., an on-system voltage regulator, bias generator modeled in hardware description language (HDL), or an off-chip supply source; also, any supply object that functions as a **root supply driver**, including a primary supply input to the design, a **leaf-level instance** supply output port, a power switch output port, and any supply object that is an input to a resolved **supply net**. *See also*: **supply source**.

self domain (of a **port** to which a given strategy applies): **The power domain** for which the strategy has been defined.

scope: A region in which names may be defined; such a region is either an hardware description language (HDL) model or **instance** in the **logic hierarchy** or a Unified Power Format–defined global context, **power state table**, **supply set**, **power domain**, **composite domain**, group, or **strategy**.

silicon UPF: *See:* **implementation UPF** (Unified Power Format).

simple name: An identifier that denotes an object declared in a given **scope** and is not a **hierarchical name**.

simstate: The level of operational capability supported by a given **power state** of a **supply set**.

sink: A **receiver**; the **HighConn** of an input port or inout port of an **instance**; or the **LowConn** of an output port or inout port of an **instance**.

soft macro: An **instance** that is represented by the original register transfer level (RTL) and Unified Power Format (UPF) from which its implementation is (or will be) derived. Additionally, **ancestor** power intent objects are not available for use within the **scope** of the **instance**.

source: A **driver**; the **LowConn** of an input port or inout port of an **instance**; or the **HighConn** of an output port or inout port of an **instance**.

state element: A sequential element such as a flip-flop, latch, or memory element. Also, a conditionally stored value in register transfer level (RTL) code from which a sequential element would be inferred.

strategy: A rule that specifies where and how to apply **isolation**, level-shifting, state retention, and buffering in the implementation of power intent.

subdomain: A member of the set of domains comprising a composite **power domain**.

supply function: An abstraction of a **supply net** in a **supply set**, the name of which identifies the purpose of the corresponding **net** in the **supply set**.

supply net: An hardware description language (HDL) representation of a **power rail**.

supply port: A connection point for **supply net**s.

supply set: A collection of **supply function**s that in aggregate provide a complete power source.

supply source: A **supply port** that propagates but does not originate a supply value.

supply subnet: A set of **electrically equivalent supply port**s, **supply net**s, and/or **supply set** functions.

switch: An **instance** that conditionally connects one or more input **supply net**s to a single output **supply net** according to the logical state of one or more control inputs.

top-level instance: An implicit **instance** corresponding to a top-level module.

upper boundary (of a power domain): The **LowConn** side of each **port** of each **boundary instance** in the **extent** of this **power domain**.

3.2 Acronyms and abbreviations

CMOS	complementary metal oxide semiconductor
DFT	Design for Test
EDA	electronic design automation
HDL	hardware description language
IP	intellectual property
MSFF	master-slave flip-flop
NMOS	N-channel metal oxide semiconductor
PG	power/ground
PMOS	P-channel metal oxide semiconductor
PST	power state table
ROM	Read-only memory
RTL	register transfer level
SAIF	Switching Activity Interchange Format
SoC	System on Chip

Tcl	Tool Command Language
UPF	Unified Power Format
VCT	value conversion table
VHDL	very high speed integrated circuit (VHSIC) hardware description language
VHSIC	Very High Speed Integrated Circuit

4. Concepts

4.1 Introduction

Clause 4 provides an overview of concepts involved in defining power intent using Unified Power Format (UPF). These concepts include those related to the representation of the design structure and functionality in one or more hardware description languages (HDLs), as well as those related to power-management structures and functionality defined for and/or added to the design to model intended power-management capabilities.

The structure and functionality of a design is specified using HDLs such as Verilog, SystemVerilog, or very high speed integrated circuit (VHSIC) hardware description language (VHDL). Each HDL has specific terminology and concepts that are unique to that language, but all HDLs share some common concepts and capabilities. A typical design may be expressed in one or more HDLs.

UPF is defined in terms of a generalized abstraction of an HDL-based design hierarchy. This abstraction enables the UPF definition to apply to a design expressed in any of the three HDLs previously mentioned, or in any combination thereof, while at the same time minimizing the complexity of the UPF definition. Clause 4 presents the abstract model and maps it to specific HDL concepts.

UPF is intended to apply to a design as its representation changes from an abstract functional model to a concrete physical model, during which process the power intent expressed in UPF becomes realized as part of the implementation. Because of this, the abstract logic hierarchy that is the basis of the UPF definition shall be understood in terms of both functional specification and physical implementation.

4.2 Design structure

4.2.1 Transistors

At the lowest level, UPF focuses on controlling power (or more precisely, voltage and current) delivered to transistors. These are usually assumed to be digital complementary metal oxide semiconductor (CMOS) transistors, but they could be analog devices as well, or implemented in other technologies. The gate connection of a transistor is a receiver; the source of the signal provided to a gate (in CMOS, typically the output of a `P/N` transistor pair) is a driver.

4.2.2 Standard cells

Transistors are seldom modeled individually in an HDL description; typically, collections of transistors are represented by standard cells that have been developed as part of a particular technology library, which is usually expressed in the Liberty library format (see [B4]). Such cells typically have a primary supply (power and ground) and can also have a secondary supply for related behavior (e.g., state retention).

4.2.3 Hard macros

A library can also contain hard macros, which provide predefined physical implementations for much larger and more complex functions. A hard macro can have multiple supplies.

4.3 Design representation

4.3.1 Models

Library elements have corresponding behavioral models for use in simulation. These models may or may not include power and ground pins for their supplies. Standard cell models are usually written as Verilog modules and use constructs such as Verilog built-in primitives or user-defined primitives (UDPs) to express the relatively simple behavior of a standard cell. They can also be written as VHDL design entities (entity/architecture pairs) using package `VITAL`, which provides Verilog-like primitive modeling capabilities. Hard macro models can be written in either language, using more complex behavioral constructs such as Verilog initial blocks and always blocks, or VHDL processes and concurrent statements.

4.3.2 Netlist

A netlist is a collection of unique instances of standard cells and hard macros, interconnected by nets (Verilog) or signals (VHDL). Such instances are considered to be leaf-level instances, because their models are not constructed from an interconnection of subordinate instances, but instead are built using behavioral or functional HDL statements. A netlist can also include hierarchical instances, i.e., instances of a model that is itself defined as a netlist.

A power/ground (PG) netlist is a netlist containing cell and/or hard macro instances that include power and ground pins and a representation of the power and ground supply routing for those instances. A non-PG netlist is one that does not include any representation of the power supply network.

4.3.3 Behavioral models

Behavioral models that are written using the register transfer level (RTL) synthesis subset of Verilog or VHDL are synthesizable models which can be read by an RTL synthesis tool and mapped to a functionally equivalent netlist. Synthesis involves identifying or inferring the state elements needed to implement the specified behavior and implementing the combinational logic interconnecting those elements and the model’s ports.

For many synthesizable HDL constructs, synthesis creates combinational or sequential logic elements that are ultimately defined in terms of transistors, which in turn define drivers and receivers. In particular, any synthesizable statement that involves conditional computation or conditional updating of an output will most likely create logic. In contrast, unconditional assignment statements and port associations typically result in interconnect, not logic; for such HDL constructs, no drivers or receivers are created. In particular, ports do not create drivers; it is the logic driving a port that creates a driver for the port and for the net associated with the port.

4.3.4 HDL scopes

An HDL model defines one or more scopes. A scope is a region of HDL text within which names can be defined. Such names are typically visible (i.e., can be referenced) within the scope in which they are defined and, in certain cases, in other scopes (e.g., nested scopes). A Verilog model usually defines a single scope for the whole model. A VHDL model often defines multiple scopes; one for the whole model, plus other nested scopes for process statements and block statements. `generate` statements in either HDL are also considered to be nested scopes within the model's top-level scope.

4.3.5 Design hierarchy

A design hierarchy is constructed by defining one model in terms of interconnected instances of other models. Each instance represents a subtree of the hierarchy; the boundary between this subtree and its parent instance is defined by the interface of the model that has been instantiated to create the subtree. The interface consists of the model's ports, together with the nets associated with those ports for the instance that created this subtree. In Verilog, a port is defined as having two sides: a *HighConn* and a *LowConn*. The LowConn represents the port declaration in the model; the HighConn represents an instance of that port associated with an instance of the model, and therefore indirectly the net attached to that port instance. In VHDL, a somewhat different distinction is made between a formal port of a model and the actual signal associated with that port for a given instance of the model. In the context of UPF, regardless of what HDL is involved, the term *LowConn* means the (formal) port declaration in the model definition, and the term *HighConn* means the port of an instance of a model and by extension the net or signal connected to that port.

An HDL model that is not instantiated in any other instance is a top model, or simply `top`. A given design hierarchy usually contains a single top, but it may contain multiple tops in certain cases (e.g., if the design and the testbench in a simulation are modeled separately—neither instantiates the other). Each top is considered to be implicitly instantiated within the *root scope*. In Verilog, the root scope is `$root`; in VHDL, the root scope is the *root declarative region*. The instance name of such an implicit instance is the same as the model name.

4.3.6 Logic hierarchy

UPF assumes a somewhat more abstract model of the design hierarchy. This abstract model is called the *logic hierarchy*. As usual, the topmost scope is still the root scope and modules that are not instantiated elsewhere are the top modules (and instances) of the hierarchy. However, in the logic hierarchy, each scope corresponds to a whole instance; internal scopes presented in the design hierarchy are not modeled. In particular, HDL `generate` statements, which are considered to be internal scopes in the respective language definitions, are assumed to be collapsed into the parent module scope in the logic hierarchy.

UPF generally allows references to the names of objects defined anywhere in the subtree descending from a given instance when the *current scope* is set to that instance. Such references are called *rooted names*, meaning they are hierarchical names relative to the current scope. If the design hierarchy contains `generate` statements that have been collapsed in the logic hierarchy, then the hierarchical name of an object in the logic hierarchy may include simple names that encode the collapsed scope names.

UPF also uses the logic hierarchy as a framework for locating the power-management objects used to represent power-management concepts, e.g., power domains and power state tables (PSTs). Each such object is effectively declared in a specific scope of the logic hierarchy, and the name of the scope can be used as the prefix of the name of the object. Furthermore, certain UPF-defined objects act as scopes themselves, in which other subordinate named objects can be defined. In such cases, the name of the UPF-defined parent object can be used as the prefix of the name of any of its subordinate objects.

The logic hierarchy can be viewed as a purely conceptual structure that is independent of the eventual physical implementation. Alternatively, the logic hierarchy can be viewed as an indication of the floor plan to be used in the physical implementation. Either view can be used, but it is best to adopt one view or the other for a given design, because the choice can affect how the power intent is expressed in UPF.

4.3.7 Hierarchy navigation

In UPF, commands are executed in the context of a scope within the logic hierarchy. The **set_scope** command (see 6.51) is used to navigate within the hierarchy and to set the current scope within which commands are executed.

Consistent with SystemVerilog `$root`, the root of the logic hierarchy is the scope in which the top modules are implicitly instantiated. Other locations within the logic hierarchy are referred to as the *design top instance*, which has a corresponding *design top module*, and the current scope.

The design top instance and design top module are typically paired: the design top instance (represented by a hierarchical name relative to the root scope) is an instance in the hierarchy representing a design for which power intent has been defined, and the design top module is the module for which the UPF file expressing this power intent has been written. The association between the UPF file and the design top module is specified in the UPF file using **set_design_top** (see 6.41); this UPF file is then typically applied to each instance of that module in a larger system.

The current scope is an instance that is, or is a descendant of, the design top instance (represented by a relative pathname from the design top instance).

The **set_scope** command (see 6.51) changes the current scope locally within the subtree depending on the current design top instance/module. Since the design top instance is typically an instance of the design top module, they both have the same hierarchical substructure; therefore, **set_scope** can be written relative to the module, but still work correctly when applied to an instance. The **set_scope** command is only allowed to change scope within this subtree. It cannot change the scope to a scope above the design top instance or to a scope that is, or is below, a leaf-level instance.

The design top instance and design top module are initially set by the tool, possibly with direction from the user. They are implicitly changed when **load_upf** is invoked with the **-scope** argument or when **apply_power_model** is invoked to apply a power model to a given instance.

4.3.8 Ports and nets

Ports define connection points between adjacent levels of hierarchy. In HDL, ports are defined as part of the interface of a module and therefore exist for each instance of the module. Nets define interconnections between a collection of ports. In HDL, nets are defined within a module and therefore exist within each instance of the module.

A port has two sides. The top side is the HighConn side, which is visible to the parent of the instance whose interface contains the port. The bottom side is the LowConn side, which is visible internal to the instance whose interface contains the port.

When a net in the current scope is connected to a port on a child instance, the connection is made to the HighConn side of the port. When a net in the current scope is connected to a port defined on the interface of the instance that is the current scope, the connection is made to the LowConn side of the port.

A port can be referenced wherever a net is required. Such a reference refers to the LowConn side of the port. A port can be thought of as being implicitly connected to an implicit net created with the same name and in the same scope as the LowConn side of the port.

4.3.9 Connecting nets to ports

In an HDL description, ports are typically required to pass nets from one level of hierarchy to another. In UPF, a net in the current scope can be connected to the LowConn of any port declared in the same scope or to the HighConn of any port within its descendant subtree. If the port is not declared in the same scope as the net, additional ports, nets, and port/net associations may be created to establish the connection from the net to the port. Such implicitly created ports and nets shall have the same simple name as the net being connected unless that name conflicts with the name of an existing port or net; in which case, to avoid a name conflict, the tool shall create a name that is unique for that scope.

NOTE—Nets are propagated as necessary through the descendant subtree and may be renamed to avoid name collision; therefore, the same simple name in different scopes may refer to nets that are independent and unconnected. [15]

Implicitly created ports and nets should not be referenced directly by UPF commands, since the names of such ports and nets may not be the same as the original net name. These implicitly created ports and nets are merely a method of implementing a UPF connection in terms of valid HDL connections, when the UPF-specified power intent is represented in HDL form.

4.3.10 Representing SystemC design for power analysis

IEEE Std 1801™-2015 supports power analysis in system-level design; see Annex H for more details. In addition to the existing HDL support (Verilog, SystemVerilog, VHDL), IEEE Std 1801-2015 system-level IP power models can also be applied to design descriptions that are written in SystemC.

Existing UPF commands like **set_scope**, **create_power_domain, apply_power_mode**, etc. can be applied to a SystemC design in the same way as existing HDL support.

A design description in SystemC is treated in the same way as other HDLs (scopes, design hierarchy, etc.) and SystemC generally follows the same rules of other HDLs.

4.4 Power architecture

4.4.1 Introduction

A UPF power intent specification defines the power architecture to be used in managing power distribution within a given design. The power architecture defines how the design is to be partitioned into regions that have independent power supplies, and how the interfaces between, and interaction among, those regions will be managed and mediated.

[15] Notes in text, tables, and figures of a standard are given for information only and do not contain requirements needed to implement this standard.

4.4.2 Power domains

A power domain is a collection of instances that are typically powered in the same way. In the physical implementation, the instances of a power domain are typically placed together and powered by the same power rails. In the logic hierarchy, the instances of a power domain are typically part of the same subtree of the hierarchy, or of sibling subtrees with a common ancestor, and powered by the same supply nets.

A power domain is defined within a scope (or instance) in the logic hierarchy. The definition of the power domain identifies the uppermost instances of the domain: those that define the upper boundary of the domain. For any given instance included in the power domain, a child instance of the given instance is transitively included in the power domain, unless that child instance is explicitly excluded from this power domain or is explicitly included in the definition of another power domain.

More formally, a boundary instance of a given power domain is any instance that has no parent (it is an implicit instance of a top-level module) or whose parent is in the extent of a different power domain. It is possible for one boundary instance of a power domain to be an ancestor of another boundary instance of the same power domain. This occurs when one instance is in the extent of a given power domain and both an ancestor and a descendant of that instance are in the extent of a second power domain. In this case, both the ancestor and the descendant may be boundary instances of the second domain. A domain with such a structure is referred to as a *donut* power domain.

The upper boundary of a power domain consists of the LowConn side of each port on each boundary instance in the domain. The lower boundary of a domain consists of the HighConn side of each port on each child instance that is in some other power domain or is a port of a macro cell instance that is powered differently from the rest of the domain. Both boundaries include any logic ports added to the design for power management. The interface of a power domain consists of the upper boundary and the lower boundary.

The instance in the logic hierarchy in which a power domain is defined is called the *scope* of the power domain. The set of instances that belong to a power domain are said to be the extent of that power domain. This distinction is important: while a given instance can be the scope of multiple power domains, it can be in the extent of one and only one power domain. As a consequence of these definitions, all instances within the extent of a domain are necessarily within the scope of the domain or its descendants.

A power domain can be either contiguous or non-contiguous. In the physical implementation, a contiguous power domain is one in which all instances are placed together; a non-contiguous power domain is one in which instances in the domain are placed in two or more disjoint locations. A power domain is contiguous within the logic hierarchy if it contains a single boundary instance; it is non-contiguous within the logic hierarchy if it includes multiple boundary instances.

For a non-contiguous power domain, a connection from an instance in the extent of the power domain to some other instance in the extent of the domain may need to be routed through another power domain.

Power domains that share a primary supply set can be composed together to form a larger power domain such that operations performed on this larger power domain apply transitively to each subdomain. In this way, unnecessary power domains may be aggregated together and handled as one for simplicity.

After UPF-specified power intent has been completely applied, it shall be an error if any instance is not included in a power domain.

4.4.3 Drivers, receivers, sources, and sinks

A logic signal in the design originates at an active component (the driver) and terminates at another active component (the receiver). Along the way it may pass through ports and nets. The driver and any port it passes through on the way to a receiver is considered a source; the receiver and any port it passes through on the way from the driver is considered a sink. For example, a buffer defines both a source and a sink: the buffer's output port is a source; the buffer's input port is a sink.

A signal traversing a power domain may or may not be driven within the power domain. A port is neither a driver nor a receiver; it merely propagates a signal across a hierarchy boundary. If a port on the interface of a power domain is connected directly to another port on the interface of the same power domain, without going through an active component, the connection between those two ports has neither a driver nor a receiver in that domain. In this case, the connection is a feedthrough path through that domain.

HDL assignment statements may include delays, which either represent inertial delay (resulting from transistor switching) or transport delay (resulting from propagation along a wire). However, synthesis tools typically ignore such delays; therefore, the inclusion of such a delay, whether inertial or transport, does not by itself imply that an active component will be inferred from the assignment. For this reason, delays are not considered to create drivers or receivers.

A connection may be thought to exist in a given domain, if a user so chooses, but since a connection is by definition a passive component, it has no driver in the domain in which it exists and therefore is not affected or corrupted by the power state of the domain in which it exists.

4.4.4 Isolation and level-shifting

Two power domains interact if one contains logic that is the driver of a net and the other contains logic that is a receiver of the same net. When both power domains are powered up, the receiving logic should always see the driving logic's output as an unambiguous `1` or `0` value, except for a very short time when the value is in transition. The structure of CMOS logic typically means that minimal current flow will occur when the input value to a gate is a `1` or `0`. However, if the driving logic is powered down, the input to the receiving logic can float between `1` or `0`. This can cause significant current to flow through the receiving logic, which can damage the circuit. An undriven input can also cause functional problems if it floats to an unintended logic value.

To avoid this problem, isolation cells are inserted at the boundary of a power domain such that the receiving logic always sees an unambiguous `1` or `0` value. Isolation may be inserted for an input or for an output of the power domain. An isolation cell operates in two modes: normal mode, in which it acts like a buffer, and isolation mode, in which it clamps its output to a defined value. An isolation enable signal determines the operational mode of an isolation cell at any given time.

Two interacting power domains may also be operating with different voltage ranges. In this case, a logic `1` value might be represented in the driving domain using a voltage that would not be seen as an unambiguous 1 in the receiving domain. Level-shifters are inserted at a domain boundary to translate from a lower to a higher voltage, and sometimes from a higher to a lower voltage as well. The translation means that the logic value sent by the driving logic in one domain is correctly received by the receiving logic in the other domain.

Isolation and level-shifting are often implemented in combination, so one standard cell implements both functions. UPF includes support for such "combo" cells.

Isolation and level-shifter strategies specify that isolation and level-shifter cells are to be inserted in specified locations. However, there are some cases where implementation tools may choose not to insert such cells, or to optimize redundant insertion of such cells. For example, isolation/level-shifters on floating ports that appear to have no drivers, or have constant drivers, may be removed or transformed, provided the resulting behavior is unchanged. To prevent implementation tools from applying such optimizations, isolation and level-shifting strategies can instead specify that the respective cells are to be inserted regardless of optimization possibilities.

4.4.5 State retention

State retention is the ability to retain the value of a state element in a power domain while switching off the primary power to that element, and being able to use the retained value as the functional value of the state element upon power-up. State retention can enable a power domain to return to operational mode more quickly after a power-down/power-up sequence and it can be used to maintain state values that cannot be easily recomputed on power-up. State retention can be implemented using retention memories or retention registers. Retention registers are sequential elements (latches or flip-flops) that have state retention capability.

For a retention register, the following terms apply:

— *Register value* is the data held in the storage element of the register. In functional mode, this value gets updated on the rising/falling edge of clock or gets set or cleared by set/reset signals, respectively.

— *Retained value* is the data in the retention element of retention register. The retention element is powered by the retention supply.

— *Output value* is the value on the output of the register.

Depending on how the retained value is stored and retrieved, there are at least two flavors of retention registers, as follows:

a) *Balloon-style retention*: In a balloon-style retention register, the retained value is held in an additional latch, often called the *balloon latch*. In this case, the balloon element is not in the functional data-path of the register.

b) *Master/slave-alive retention*: In a master/slave-alive retention register, the retained value is held in the master or slave latch. In this case, the retention element is in the functional data-path of the register.

A balloon-style retention register typically has additional controls to transfer data from a storage element to the balloon latch, also called the *save step*, and transfer data from the balloon latch to the storage element, also called the *restore step*. The ports to control the save/restore pins of the balloon-style retention register need to be available in the design to describe and implement this style of registers.

A master/slave-alive retention register typically does not have additional save/restore controls as the storage element is the same as the retention element. Additional control(s) on the register may park the register into a quiescent state and protect some of the internal circuitry during power-down state, and thus the retention state is maintained. The restore in such registers typically happens upon power-up, again owing to the storage element being the same as the retention element. Thus, this style of registers may not specify save/restore signals, but may specify a retention condition that could take the register in and out of retention.

4.5 Power distribution

4.5.1 Overview

The electric current transported by a supply net originates at a root supply driver, which can be an on-chip voltage regulator, an embedded power switch, a bias generator, or an off-chip supply source. A power switch output and a resolved supply net are both also considered to be root supply drivers for semantic consistency.

Each supply subnet (see 3.1) has an associated root supply driver. If the supply subnet includes a primary supply input, the root supply driver is an implicit driver representing an off-chip supply source. If the supply subnet includes a macro cell output supply port, the root supply driver is an implicit driver representing an embedded power switch or supply regulator. If the supply subnet includes a switch output port, the root supply driver is the power switch output. If the supply subnet includes one or more resolved supply nets, the root supply driver is the output of the common resolution function shared by those resolved supply nets.

Initially, the root supply driver drives the supply subnet with the value `{OFF, unspecified}`. The package UPF functions `supply_on` and `supply_off` may be called to change the driving value of the root supply driver that drives a given supply subnet. These functions may be applied to any supply object in the supply subnet, provided that distribution of the supply value (or the result of a resolution of this supply value and other supply values) to loads of the supply network does not require violating the directionality of any port in the supply subnet.

A supply net can have one or more supply sources, depending upon its resolution type. During UPF processing, if the number of sources connected to a supply net do not conform to the requirements of its resolution type, an error shall be reported. At any given time during simulation, if the sources of a supply net do not conform to the requirements of its resolution type, the resolved value of the supply net at that time is set to `{UNDETERMINED, unspecified}`.

A power switch can have one or more input supply ports and one output supply port. Each input supply port can have one or more state definitions. At any given time during simulation, if the state definitions of a given input supply port are contradictory, or if multiple incompatible inputs are enabled at the same time, or if any input supply port is in an error state, the resolved value of the output supply port at that time is set to `{UNDETERMINED, unspecified}`.

The semantics defined in this standard, such as the supply net resolution functions, presume an idealized supply network with no voltage drop; the semantics for supply network resolution with modeled-voltage drop are outside the scope of this standard.

4.5.2 Supply network elements

Supply network objects (supply ports, supply nets, and switches) are created within the logic hierarchy to provide connection points for a root supply and to propagate the value of a root supply throughout a portion of the design. Supply network objects are created independent of power-domain definitions. This allows sharing of common components of the supply distribution network across multiple power domains.

4.5.3 Supply ports and nets

Supply ports provide a connection point for supply nets where they cross a hierarchy boundary. Supply nets can be used to create a connection between two supply ports or from a supply port to an instance within a power domain.

Supply ports and nets can be created in UPF or in the HDL design. If created in the HDL, the port or net shall be of the supply net type defined in the appropriate package UPF (see 11.2). Supply ports shall also be inferred from Liberty using the `pg_pin` attribute (see Annex F). In UPF, supply ports on power-management cells may be specified using the appropriate power/ground options on the define commands (see Clause 7) or specified through Liberty. Supply ports may be specified on hard-macros (4.9.2.4) using **create_supply_port**.

4.5.3.1 Supply switches

Supply switches conditionally propagate the value on an input supply port to an output supply port, depending upon the value of a control signal. A supply net is either connected to one or more power switches or supply ports, which function as root supply drivers.

4.5.3.2 Supply sets

A supply set represents a collection of supply nets that provide a complete power source for one or more instances. Each supply set defines six standard functions: **power**, **ground**, **pwell**, **nwell**, **deeppwell**, and **deepnwell**. Each function represents a potential supply net connection to a corresponding portion of a transistor. Each function of a given supply set can be associated with a particular supply net that implements the function.

A global supply set is one that is defined in a given scope and associates supply nets with its functions. One or more local supply sets, called *supply set handles*, can be defined for a power domain, a power switch, an isolation strategy (see 6.44), a level-shifting strategy (see 6.45), or a retention strategy (see 6.49). A supply set can be associated with a supply set handle as a whole; the functions of a supply set handle can be broken out and connected to ports of instances. This association creates a connection between the supply nets represented by corresponding functions of the supply set and supply set handle.

A supply set function is equivalent to a supply net and may be used anywhere a supply net is allowed. The supply set function represents the supply net that is or will be associated with that function of the supply set. The supply set function reference is a symbolic name for the supply net it represents.

A reference to a supply net by its symbolic name is an indirect reference.

NOTE—A supply net may be associated with a function of more than one supply set. The function that a given supply net performs in one supply set is unrelated to the function it may perform in any other supply set.

4.5.4 Supply network construction

4.5.4.1 Introduction

Supply ports and nets are interconnected to create a supply network. Certain definitions and restrictions constrain how these interconnections are made.

4.5.4.2 Supply sources and loads

Supply ports define supply sources and supply loads, as follows:

— The LowConn of an input or inout port is a supply source. The HighConn of an output or inout port is a supply source (including a switch output).

— The LowConn of an output or inout port is a load. The HighConn of an input or inout port is a load (including a switch input).

A port that is neither a top-level port nor a leaf-level port is an internal (hierarchical) port.

4.5.4.3 Supply port/net connections

Connections are made from nets to ports:

a) from a net to (the LowConn of) a port declared in the same scope; or

b) from a net to (the HighConn of) a port declared in a lower scope; or

c) from a net to a pin of a leaf cell.

The LowConn of a port may be used as an implicit net and connected to another port.

Only one net connection can be made to the LowConn of a port. Likewise, only one net connection can be made to the HighConn of a port. A source can be connected to a net that is in turn connected to multiple loads.

4.5.4.4 Supply net resolution

A supply net may be unresolved or resolved, as follows:

— An unresolved supply net shall have only one supply source connection.

— A resolved supply net can have multiple supply source connections. The resolution type may restrict how many supply sources can be on at the same time.

A supply net can have any number of load connections.

4.5.4.5 Supply net/supply set connections

Related supply nets can be grouped into a supply set, with each supply net in the group providing one or more functions of the supply set. The supply net corresponding to a given function of a supply set can be specified when the supply set is created or updated (see 6.26). One supply set may be associated with another supply set (see 6.10); this implicitly connects corresponding functions together and therefore it also implicitly connects the supply nets associated with corresponding functions and any instance ports to which those functions are connected.

4.5.4.6 Supply set function connections

4.5.4.6.1 Overview

Supply functions of a supply set, and the supply nets they represent, can be connected to instances in one of the following ways: explicitly, automatically, or implicitly. Connections are made downward, from ports or nets in the current scope to ports of descendant instances that are in the extent of the domain.

4.5.4.6.2 Explicit and automatic connections

An explicit connection connects a given particular supply set function directly to a specified supply port. See also 6.14 and 6.15.

An automatic connection connects each supply set function to ports of selected instances, based on the *pg_type* of each port, as indicated by the **UPF_pg_type** attribute (see 6.47) or the Liberty pg_type attribute.

For automatic connections, the default connection semantics for each function of a supply set are as follows:

a) **power** is connected by default to ports having the *pg_type* `primary_power`.
b) **ground** is connected by default to ports having the *pg_type* `primary_ground`.
c) **pwell** is connected by default to ports having the *pg_type* `pwell`.
d) **nwell** is connected by default to ports having the *pg_type* `nwell`.
e) **deeppwell** is connected by default to ports having the *pg_type* `deeppwell`.
f) **deepnwell** is connected by default to ports having the *pg_type* `deepnwell`.

4.5.4.6.3 Implicit connections

An implicit connection connects the required functions of a supply set to cell instances that do not have explicit supply ports. Such connections may involve implicit creation of ports and nets, as described in 4.3.9.

Implicit supply set connections are made in each of the following cases:

a) Primary supply set

   The functions of a domain's primary supply set are implicitly connected to any instance in the extent of the domain if the instance has no supply ports defined on its interface.

b) Retention supply set

   The functions of a retention strategy's supply set are implicitly connected to the state element that implements retention functionality (e.g., a balloon latch, shadow register, or live slave latch) for any register in the domain to which the strategy applies.

c) Isolation supply set

   The functions of a supply set for an isolation strategy are implicitly connected to the corresponding isolation cell implied by the application of the strategy.

d) Level-shifter supply sets

   The functions of a supply set for a level-shifting strategy are implicitly connected as appropriate to the input, output, or internal supply pins of any level-shifter implied by the application of the strategy.

After UPF-specified power intent has been completely applied, it shall be an error if any instance in the design does not have a supply set function or supply net connected to each of its supply ports, including any implicit power and ground ports.

4.5.4.7 Supply set required functions

Although a supply set represents a collection of six standard supply functions, not all functions are required in every context:

— `power` and `ground` are typically required in all cases.

— `nwell`, `pwell`, `deepnwell`, and `deeppwell` are only required occasionally.

The required functions of a given supply set are determined from its usage and include the following:

a) Any function used to define a power state of the supply set,

b) Any function used for automatic connection of the supply set based on `pg_type`, and

c) Any required function of a supply set handle with which the supply set is associated.

For implementation, a supply net shall be associated with each required function of a supply set. For verification, however, some aspects of the power intent can be verified before associating supply nets with the required functions. A supply set that does not have supply nets associated with each of its required functions is incompletely specified. For any required function of a supply set that is not associated with a supply net, an implicit supply net is created and associated with the function.

4.5.5 Supply equivalence

4.5.5.1 Overview

Various aspects of power management are determined in part by the identity of, and relationships between, supply nets and supply sets. For example, selection of ports to which isolation or level-shifting strategies can be defined based on the identities of the driver and receiver supplies of the sources and sinks connected to a port. Similarly, composition of power domains is possible provided the supplies of the subdomains involved meet certain constraints. In some situations, identical supply nets or supply sets are required; other situations will only require supply nets or supply sets that are equivalent.

There are two kinds of supply equivalence: electrical equivalence and functional equivalence.

Electrical equivalence can affect:

— The number of sources of a supply network, and therefore

— Whether resolution is required for that supply network

Electrical equivalence implies functional equivalence, but not vice versa.

Functional equivalence can affect any of the following:

— Insertion of isolation cells, level-shifter cells, and repeater cells

— Determination of power-domain lower boundaries

— Legality of power-domain composition

— Validity of driver and receiver supply attributes

Electrical equivalence is primarily related to supply ports and nets. Functional equivalence is primarily related to supply sets.

4.5.5.2 Supply port/net equivalence

Electrical equivalence is determined by connection, as follows:

a) A port `P` is electrically equivalent to itself.

b) A net `N` is electrically equivalent to itself.

c) If an unresolved net `N` and a port `P` are connected, then `N` and `P` are electrically equivalent.

d) If a resolved net `N` and a port `P` are connected, and `P` is an inout port or is a load of `N`, then `N` and `P` are electrically equivalent.

e) If `A` and `B` are electrically equivalent, and `B` and `C` are electrically equivalent, then `A` and `C` are electrically equivalent.

f) If `A` and `B` are connected via a supply set function (see 4.5.4.5), then `A` and `B` are electrically equivalent.

NOTE—By definition, a port that is a source of a resolved supply net is never equivalent to that resolved supply net, because the value provided by the port to the resolved supply net is not necessarily the same as the resolved value of the supply net.

g) Electrical equivalence can also be declared, as follows:

— If `A` and `B` are declared electrically equivalent, then `A` and `B` are electrically equivalent.

Electrical equivalence implies the two equivalent objects are electrically connected somewhere. If the connection is not evident in the design (e.g., if it is inside a hard macro whose internals are not visible or if it is a connection that is required outside the design), then declaration of electrical equivalence can be used instead of the explicit connection.

h) Functional equivalence is determined by connection or declaration, as follows:

— If `A` and `B` are electrically equivalent, then `A` and `B` are functionally equivalent.

— If `A` and `B` are declared functionally equivalent, then `A` and `B` are functionally equivalent.

An input and the output of a switch are never electrically equivalent; it shall be an error if they are directly connected or declared electrically equivalent. Similarly, the outputs of two different switches are typically not electrically equivalent, unless they are both driving the same resolved net. However, the outputs of two different switches that each drive an unresolved net can still be functionally equivalent if the input supplies of both switches are equivalent, the control inputs of both switches are logically equivalent, and the two switches have the same set of state definitions.

4.5.5.3 Supply set equivalence

A supply set handle is also a supply set.

A supply set function and its associated supply net are electrically equivalent; thus, for purposes of supply net equivalence, a supply set function acts like a supply net.

Corresponding functions of two supply sets are electrically equivalent if:

— their associated supply nets are electrically equivalent, or

— the two supply sets are directly associated with one another.

Corresponding functions of two supply sets are functionally equivalent if:

— they are electrically equivalent, or

— they have been declared as functionally equivalent.

Two supply sets are (functionally) equivalent if:

— they both have the same required functions, and the nets associated with corresponding functions are equivalent; or

— they are associated with each other directly or indirectly via one or more **associate_supply_set** commands (see 6.10); or

— they are each associated directly or indirectly via **associate_supply_set** (see 6.10) with two other supply sets, which are equivalent.

Two supply sets are also (functionally) equivalent if they have been declared equivalent; in this case, it shall be an error if they do not have the same required functions.

As a consequence of this:

a) two anonymous supply sets built from equivalent PG functions are equivalent;

b) two supply sets that are functionally equivalent can be used interchangeably;

c) a supply set and any supply set handle it is associated with are always equivalent.

4.5.6 Supply subnets

Supply ports, supply nets, and supply set functions that are electrically equivalent (see 4.5.5) make up a *supply subnet*. A supply subnet that contains no resolved supply nets has a single root supply driver (see 4.5) whose value determines the values of all supply objects in the supply subnet. A supply subnet that contains one or more resolved supply nets can have multiple root supply drivers.

A supply network consists of one or more supply subnets. Two supply subnets are indirectly connected when one contains a supply object that is an input to a resolved supply net and the resolved supply net is in the other subnet, or when one contains a supply object that is an input to a power switch and the output of the power switch is in the other subnet.

The definitions of root supply driver, electrical equivalence, and supply subnet mean that if more than one resolved supply net is present, all equivalent resolved supply nets are part of the same subnet, and all have the same resolution function. This allows all sources of all resolved supply nets to be resolved by one instance of the resolution function. The resolved value is then distributed to any unresolved supply objects in the supply subnet and to any loads of the supply subnet. See 9.2.3 for the simulation semantics of supply networks.

4.5.7 Supply variation

Supply ports, supply nets, and supply set functions take on values that consist of a state and a voltage. Named port states (see 6.4) and named power states (see 6.5) can be defined to represent the nominal voltages that a supply object may carry. These nominal voltage values are used also for determining whether level-shifting is required (see 6.45).

In an implementation, the actual voltage of an object may vary around the nominal values. There are several sources of such variation. One source of variation is the accuracy of the supply. Supply variation can be modeled in UPF using the **set_variation** command (see 6.53). Supply variation is applied to nominal voltages to derive variation ranges for those voltages. Other sources of variation are beyond the scope of this standard.

Nominal voltages are not intended to represent library characterization points and should not be used as such. Implementation tools need to take supply variation and other factors into account when determining what library elements to use.

4.5.8 Supply correlation

Supply variation ranges (see 4.5.7) are used when determining whether level-shifting is needed. How such variation ranges are used depends upon whether the supplies are correlated or not.

When the driving and receiving supplies are not correlated, they vary independently. In this case, level-shifting analysis considers the possibility that the two supplies are simultaneously at opposite ends of their respective variation ranges, and therefore compares minimum versus maximum voltage and maximum versus minimum voltage.

When the driving and receiving supplies are correlated, they are assumed to vary consistently, such that if one supply voltage is at its minimum (respectively maximum) value, then the other supply voltage is also at its minimum (respectively maximum) value. In this case, level-shifting analysis only considers the voltage difference when the two supplies are at the same end of their respective ranges, and therefore only compares minimum versus minimum voltage and maximum versus maximum voltage.

Correlation of supplies is transitive: if supplies A and B are correlated, and supplies B and C are correlated, then supplies A and C are also correlated.

4.6 Power management

4.6.1 Introduction

While a power supply network is a static structure, the power delivered via the power supply network can vary over time. Supply sources can provide different voltages; power switches can turn their outputs off or on and can selectively connect different inputs to the output. As a result, the power available to instances in the extent of a power domain will vary, and at any given time, each power domain's supplies will be in one of many possible states. To manage these various states, and in particular to manage the interactions between power domains that are in different states, power management is required.

Power management enables a system to operate correctly in a given functional mode with the minimum power consumption. Adding power management to a design involves analyzing the design to determine which power supplies provide power to each logic element, and if the driver and receiver are in different power domains, inserting power-management cells as required to ensure that neither logical nor electrical problems result if the two power domains are in different power states.

4.6.2 Related supplies

An active component consists of logic elements that receive inputs and drive outputs. The power supplies connected to an active component provide power for this logic. The supply nets that provide power for the logic that receives or drives a given input or output, respectively, are called the *related supplies* of that input or output. Related supplies typically include power and ground supplies and may also include bias supplies.

At the library cell level, related supplies may be identified for each input or output pin of a cell. Each related supply is a supply pin on that cell; the pin typically has a `pg_type` attribute indicating what supply function it provides (primary power, primary ground, etc.). For a cell that has one set of supply connections, all inputs and outputs would have the same set of related supplies. For a cell that has multiple supply connections, such as a cell with a backup power supply, different pins can have different sets of related supplies. This is particularly true of certain power-management cells, such as a level-shifter, which usually has different related supplies for the input and output.

Related supply nets are often considered in a group, as an implicit supply set. An implicit supply set made up of the supply pins of a cell that are the related supplies of a given input or output is by definition equivalent to any supply set that has been connected to those supply pins.

4.6.3 Driver and receiver supplies

Each output of an active component is typically connected to the input of some other active component in the design. The net connecting the two has a driver on one end (the logic driving the output port) and a receiver on the other end (the logic receiving the input). The driving logic is powered by a supply set called the *driver supply*; the receiving logic is powered by a supply set called the *receiver supply*.

The driver supply and the receiver supply can be the same supply set, e.g., if both components are in the same power domain; or the driver supply and the receiver supply can be different supply sets, e.g., if the two components are in different power domains. The driver supply and the receiver supply can also be different, but nonetheless equivalent, e.g., if they are connected externally or if they are generated by supply networks that ensure they always have the same values.

In some cases, the logic driving or receiving a given port is not evident. In particular, the logic inside a macro instance may not be represented in a way that can be used by a given tool. Similarly, the logic that drives primary inputs of the design and receives primary outputs of the design is typically not represented as part of the design. In such cases, it is convenient to be able to associate the driver supply or receiver supply of the missing logic with the port that is connected to that logic. UPF defines attributes that can be used to associate this information with ports of a model.

4.6.4 Logic sources and sinks

Logic ports can be a source, a sink, or both, as follows:

- The LowConn of an input or inout logic port whose HighConn is connected to an external driver is a source.
- The HighConn of an output or inout logic port whose LowConn is connected to an internal driver is a source.
- The LowConn of an output or inout logic port whose HighConn is connected to an external receiver is a sink.
- The HighConn of an input or inout logic port whose LowConn is connected to an internal receiver is a sink.

For a logic port that is connected to a driver, the supply of the connected driver is also the driver supply of the port. A primary input port is assumed to have an external driver and therefore is a source; such a port has a default driver supply if it does not have an explicitly defined **UPF_driver_supply** attribute. An internal port that is not connected to a driver is not a source, and therefore, does not have a driver supply in the design. To model this in verification, an anonymous default driver is created for such an undriven port. This driver always drives the otherwise undriven port in a manner that results in a corrupted value on the port.

For a logic port that is connected to one or more receivers, the supplies of the connected receivers are all receiver supplies of the port. A primary output port is assumed to have an external receiver and therefore is a sink; such a port has a default receiver supply if it does not have an explicitly defined **UPF_receiver_supply** attribute. An internal port that is not connected to a receiver is not a sink, and therefore, does not have any receiver supplies.

The following paragraphs define the power management of HDL literals (e.g., 1′b1)

A literal value (e.g., SystemVerilog 1′b1 or VHDL '1') can be connected to an input port of a macro instance either directly or indirectly. The following three cases can be distinguished:

— A literal is directly associated with an input port of a macro.

— A literal is assigned to a wire (in SystemVerilog) or signal (in VHDL) that is then possibly propagated through various port associations and finally associated with an input port of a macro instance, and the initial literal assignment is done within the extent of the power domain in which the macro is instantiated.

— A literal is associated with a formal input port in an instantiation of a module, and that input port is connected to a wire (SystemVerilog) or signal (VHDL) that is then possibly propagated through various additional port associations and finally associated with an input port of a macro instance, and the initial literal port association is done within the extent of the power domain in which the macro is instantiated.

A literal value shall be implemented with a tie cell. A tie cell can be either:

— A primary rail cell, the output of which is supplied by the primary supply of the domain in which the tie cell is located, or

— An Always-On cell, the output which is supplied by a backup supply (i.e., different from the primary supply of the domain in which the tie cell is located).

A literal connected to a macro instance input port with the attribute **UPF_literal_supply** (see 5.6) is modeled as a tie cell instantiated in the domain in which the literal is referenced (which domain also contains the macro instance) and supplied by the specified literal supply.

If the technology involved supports appropriate tie cells and if the specified literal supply is available in terms of supply availability rules (see 6.20), the implementation shall be a tie cell in the domain. It shall be an error if the technology involved does not support appropriate tie cells (e.g., no support of Always-On tie cells whereas the specified literal supply is different from the primary supply of the domain in which the tie cell is located) or if the specified literal supply is not available. If the technology involved does not support any tie cells, the implementation shall be a connection to the appropriate rail of the specified literal supply.

A literal connected to a macro instance input port that does not have the attribute **UPF_literal_supply** is modeled as a tie cell instantiated in the domain in which the literal is referenced (which domain also contains the macro instance) and supplied by the receiving supply of the macro port, defined by the predefined attribute **UPF_receiver_supply,** or by predefined attributes **UPF_related_power_port**, **UPF_related_ground_port**, and **UPF_related_bias_ports** (see 5.6).

If the technology involved supports appropriate tie cells and if the receiving supply is available in terms of supply availability rules (see 6.20), the implementation shall be a tie cell in the domain. It shall be an error if the technology involved does not support appropriate tie cells (e.g., no support of Always-On tie cells whereas the receiving supply is different from the primary supply of the domain in which the tie cell is located) or if the receiving supply is not available. If the technology involved does not support any tie cells, the implementation shall be a connection to the appropriate rail of the receiving supply.

For any other cases, a literal is modeled as a tie cell instantiated in the domain in which the literal is referenced and supplied by the primary supply of this domain.

If the technology involved supports tie cells, the implementation shall be a primary rail tie cell in that domain. If the technology involved does not support any tie cells, the implementation shall be a connection to the appropriate rail of the primary supply.

4.6.5 Power-management requirements

Power management is required to mediate the changing power states of power domains in the system and the interactions between power domains that are in different states at various times. There are four specific areas addressed by power management, as follows:

- If a power domain is powered down in certain situations, its state registers may need to have their values saved before power-down and restored after subsequent power-up, either to maintain persistent data or to enable faster power-up.
- If the distance between driver and receiver is long (the capacitive load is high), buffers (repeaters) may be required to strengthen the signal along the way, or to ensure that it stabilizes within the required time.
- If a receiver is powered on, but its driver is not, an isolation cell is required between driver and receiver to drive the receiver with a known value despite the fact that the ultimate driver is powered off.
- If the driver and receiver supplies (or isolation and receiver supplies, or driver and isolation supplies, etc.) are operating at different voltage levels, a level-shifter is required between them to translate between voltage levels.

UPF provides commands for specifying where power-management structures should be added to a design to address each of these areas.

4.6.6 Power-management strategies

Addition of power-management cells to a design is driven by rules or strategies. UPF provides commands for specifying retention strategies (see 6.49), repeater strategies (see 6.48), isolation strategies (see 6.44), and level-shifting strategies (see 6.45). Each of these strategies can be defined in various ways to apply to specific design features or more generally to classes of features. Precedence rules (see 5.7) define how multiple strategies for the same feature are to be interpreted. In general, more specific strategies take precedence over more general strategies.

Retention strategies apply to specific state variables in a given power domain or to all state variables in a domain. A retention strategy also defines the power supplies, the control signals and their interpretation, and certain behavioral characteristics of the retention registers to be used for the state variables to which it applies.

Repeater, isolation, and level-shifting strategies apply to ports of a power domain. The ports to which one of these strategies applies can be defined by name or can be selected by filters. Source and sink filters select ports based on the driver supply and receiver supply, respectively, of each port. The filters typically match equivalent supplies unless an exact match is specified. Ports can also be selected by direction. Each of these strategies also specifies the relevant power supplies and control signals and their interpretation to be used for any power-management cells added by the strategy.

4.6.7 Power-management implementation

Implementation of power-management strategies involves adding power-management cells—retention registers, repeaters (buffers), isolation cells, and level-shifter cells—to the design. Each added cell may add new driving and receiving logic and as a result may change the driver and receiver supplies of a given port, which could potentially affect the application of other strategies based on source and sink filters. To ensure the interaction of multiple strategies is well defined, strategies are applied according to the following rules:

a) Strategies are implemented in the following order: retention strategies, followed by repeater strategies, followed by isolation strategies, followed by level-shifter strategies.

b) A retention strategy may affect the driving supply of the retention cell output. If so, the new driving supply of the retention cell is visible to, and affects the result of, a source filter of any subsequently applied strategy.

c) A repeater strategy causes insertion of a buffer, which has a receiver and a driver; this insertion therefore affects both the receiving supply of ports driving the repeater input and the driving supply of ports receiving the repeater output. The new driving supply and receiver supply are visible to, and affect the result of, source and sink filters, respectively, of any subsequently applied strategy.

d) An isolation strategy may cause insertion of an isolation cell, which has a receiver and a driver; therefore if such insertion occurs, it affects both the receiving supply of ports driving the isolation cell input and the driving supply of ports receiving the isolation cell output. However, the new driving supply and receiver supply are not visible to, and do not affect the result of, source and sink filters, respectively, of any subsequently applied isolation or level-shifting strategies.

e) A level-shifting strategy may cause insertion of a level-shifting cell, which has a receiver and a driver; therefore if such insertion occurs, it affects both the receiving supply of ports driving the level-shifting cell input and the driving supply of ports receiving the level-shifting cell output. However, the new driving supply and receiver supply are not visible to, and do not affect the result of, source and sink filters, respectively, of any subsequently applied level-shifting strategy.

Repeater, isolation, and level-shifting strategies apply to all ports on the interface of a power domain, both those on the upper boundary of the domain and those on the lower boundary of a domain. As a result, a port on the boundary between two domains—the upper boundary of one, and the lower boundary of the other—may have multiple strategies of a given type defined for it, one from each of the two domains. In such a case, both strategies may cause addition of power-management cells.

4.6.8 Power-management cells

Power-management cells that have a single set of supply connections are also referred to as single-rail power-management cells. Single-rail cells typically share the same supply as the domain primary, but can also be physically placed in a different supply region. A typical example of a single-rail cell is an isolation cell placed in the destination domain.

Power-management cells that have two sets of supply connections are also referred to as dual-rail power-management cells. Dual-rail cells have a secondary or backup supply that enable them to be placed in the primary domain but still have secondary supply connectivity to other supplies. Typical examples of dual-rail power-management cells are dual-rail buffers and inverters, isolation cells placed in switched source domain, retention flops, power switches, level-shifters, etc.

Power-management cells with more than two sets of supply connections are also referred to as multi-rail power-management cells. Multiple supplies are more common in macros than standard cells. Typical examples of this case are multi-rail level-shifters (input supply, domain supply, output supply), power multiplexors, etc.

4.6.9 Power control logic

Most power-management cells require control signals to coordinate their activity. In particular, isolation cells require enable signals, retention cells may require save and restore signals or related control inputs, and power switches (see 4.5.3.1) require switch control signals. Logic ports and nets that implement these control signals may be present already in the HDL design or they may be added via UPF commands.

Control logic ports and nets defined in UPF are created within the logic hierarchy independent of power-domain definitions. This allows the power control network to be created and distributed across power domains.

A control signal is logically equivalent to itself. Two different control signals are logically equivalent to each other if one is directly connected to the other, if they are both directly connected to a common logic source (see 4.6.4), or if their respective logic sources are equivalent.

4.7 Supply states and power states

4.7.1 Overview

Power is required for the operation of a system. Supply ports, supply nets, and supply set functions propagate power from root supply sources to the active components of a system. Supply switches affect the propagation of power from supply sources to supply consumers. Active components consume power as required by a given operating mode.

An object that propagates power can be in various *supply states*. A supply state of an object that propagates power represents the power provided by the supply source(s) of that object and therefore the power it can propagate to power-consuming objects to which it is connected, assuming an ideal power source that can handle an infinite load.

An object that consumes power can be in various *power states*. A power state of an object that consumes power represents an operating mode of that object and therefore the power required by the object in that operating mode.

Power states can also be defined for collections of objects. Such power states name combinations of power states and/or supply states of other objects and may impose constraints on such combinations.

4.7.2 Supply states

The supply states of supply ports, supply nets, and supply set functions are represented by type **supply_net_type**, defined in package UPF (see 11.2). This type models electrical values as a combination of two values: a state value and a voltage value, which together constitute the supply state of a supply port, supply net, or supply set function.

— The state value is one of OFF, UNDETERMINED, PARTIAL_ON, or FULL_ON. The state value represents the ability of the object's root supply source(s) to provide power.

— The voltage value is internally represented as an integer number of microvolts, measured relative to a single common reference ground that is assumed to apply to the entire design. The voltage value is relevant only for the PARTIAL_ON and FULL_ON state values; it is undefined for the OFF and UNDETERMINED state values.

— The state value is not affected by or determined by the voltage value.

Supply states of a supply port that is a supply source propagate to any supply net or supply set function to which that port is electrically equivalent.

Supply states of a supply port are defined as named *port states*. Port states may be referenced in a power state table to specify legal combinations of port states that may exist.

Supply switches also have named states, which correspond to control expressions that determine which input(s) of a switch affect the switch output supply state (see 4.5.3.1). The named states of a supply switch therefore determine whether and how the switch propagates the supply state(s) of its input supply port(s) to its output supply port.

4.7.3 Power states

Power states are defined for supply sets, power domains, composite domains, groups, modules, and instances.

By default, any object for which power states can be defined has a predefined power state UNDEFINED. This power state initially represents the undifferentiated set of all possible functional states of that object. Specific states within that set of all possible power states of the object can be defined. Defining a power state creates a *named power state* that represents a subset of the functional states of the object. Defining a named power state removes that subset of functional states from the set of functional states represented by the UNDEFINED power state.

A given power state is *active* when certain conditions occur. When a power state of an object is active, that power state characterizes the operating mode of the object at that time. More than one power state of a given object may be active at the same time, subject to certain restrictions.

A power state of an object is characterized by its *defining expression*. The defining expression for a named power state is the logic expression specified in its definition (see 6.5). The defining expression of the UNDEFINED power state is effectively the condition that no other power state of this object is active.

A named power state is a *definite power state* if its defining expression consists of a single term or conjunction of terms, such that each term is one of the following:

a) a Boolean expression over signals in the design, or
b) a term of the form "<object>==<state>", where
    1) <object> is the name of an object for which power states are defined and
    2) <state> is the name of a definite power state of <object>.

In the latter case, the term evaluates to True when <state> of <object> is active. A definite power state is active when its defining expression evaluates to True.

A named power state is a *deferred power state* if it has no defining expression. This can occur when the exact definition will involve implementation details that are not yet known, such as which supply rail will be switched, or what control signals will determine the state. A deferred power state is considered to be a definite power state whose defining expression will be provided at a later time. A deferred power state is active when the state has been assigned to an object by the `set_power_state` function, or when certain conditions occur for predefined power states (see 4.7.4).

A named power state that is neither a definite power state nor a deferred power state is an *indefinite power state*. An indefinite power state is active when its defining expression evaluates to True.

A named power state R of an object O is a *refinement* of another named power state S of the same object O if both R and S are definite states and the defining expression of R includes exactly one term of the form O==S. A named power state S of an object O is an *abstraction* of another named power state T of the same object O if T is a refinement of S. The refinement and abstraction relations are both transitive; if R is a refinement of S, and S is a refinement of T, then R is a refinement of T, and T is an abstraction of R.

If one power state is an abstraction or refinement of another power state, then the two power states are *related by refinement*. For a given set of power states that are related by refinement, the *most refined power state* is the unique state that is not an abstraction of any other state in the set. A power state S of an object is a *fundamental power state* if it is a power state that is not a refinement of any other power state of that object.

By definition, if power state R is a refinement of power state S, and R is active, then S is also active. Also by definition, no named power state is related by refinement to the **UNDEFINED** power state.

Fundamental power states of a given object shall be *mutually exclusive*. It shall be an error if two fundamental power states of the same object are both active at the same time. Similarly, two different refinements of the same power state shall be mutually exclusive. It shall be an error if two different refinements of the same power state are both active at the same time. The predefined power state ERROR represents the error condition in which two states that should be mutually exclusive are both active at the same time.

A power state can be either *legal* or *illegal*. A legal power state represents a state of an object that is intended or expected to occur in normal operation of the system. An illegal power state represents a state of an object that is not intended or expected to occur in normal operation. By default, a named power state is legal unless its definition specifies that it is illegal or it is a refinement of an illegal power state. Therefore a legal state may be an abstraction of an illegal state, but an illegal state cannot be an abstraction of a legal state. Equivalently, a legal state may have an illegal refinement, but an illegal state cannot have a legal refinement.

A power state that is legal for a given model may be marked as illegal for a given instance of that model. Any refinement of such a power state is also illegal for that instance of the model.

The *current power state* of an object is determined as follows:

- if exactly one named power state of the object is active,
  - then that state is the current power state; else
- if all active states of the object are definite states that are related by refinement,
  - then the most refined power state is the current power state; else
  - the predefined ERROR state is the current power state.

The set of power states for a given object may be marked as *complete*, which indicates that all fundamental states of the object have been defined as named power states. If the set of power states for an object is complete, then it shall be an error for the **UNDEFINED** power state to be the current power state of that object. It is also an error if a new fundamental power state is defined after the power states are marked complete.

NOTE 1—By definition, a fundamental power state of an object is active whenever any refinement of that power state is active.

NOTE 2—Three distinct error conditions can occur related to active or current power states:

1) An illegal power state of an object is active.

2) Two different fundamental states of the same object or two different refinements of the same power state are active at the same time, and therefore the current power state of the object is the **ERROR** state.

3) The current power state of an object is the **UNDEFINED** state, but the set of power states for that object is specified as complete.

NOTE 3—Predefined power states **ERROR** and **UNDEFINED** represent situations in which the set of power state definitions for an object is inconsistent or incomplete. Illegal power states represent user-defined error conditions or states that should not occur in a given context.

4.7.4 Predefined power states

The predefined power states **UNDEFINED** and **ERROR** are defined for every supply set, power domain, composite domain, group, model, and instance. For a supply set, predefined power state **UNDEFINED** is defined with no simstate, and predefined power state **ERROR** is defined with simstate **CORRUPT**.

For a supply set, the power states **ON** and **OFF** are predefined as deferred power states. Power state **ON** is defined with simstate **NORMAL**; power state **OFF** is defined with simstate **CORRUPT**. (See 4.8 for a description of simstates.) The definitions of these states may be updated to specify a logic expression, a supply expression, or legality of the state (see 6.5).

Power state **ON** of a supply set is active when its defining expression is present and evaluates to True, or else when all of its required supply functions are FULL_ON, or when the supply set's power state has been successfully set to the **ON** state either directly or indirectly by the `set_power_state` function (see 9.3.1).

Power state **OFF** of a supply set is active when its defining expression is present and evaluates to True, or else when no other named power state defined for the supply set is active, or when the supply set's power state has been successfully set to the **OFF** state either directly or indirectly by the `set_power_state` function.

4.7.5 Objects with power states

Power states can be defined for various kinds of objects. These include supply sets, power domains, composite domains, groups, models, and instances. The definition of a power state for a given object can depend upon power states of other objects, with certain restrictions.

A supply set represents a collection of supply set functions that will eventually be provided by the supply distribution network. Power states of a supply set are defined in terms of the supply states of the supply set functions. Such power states identify various levels of power that can be made available via the supply set for consumption by design elements, the legality of each of these levels, and if the supply set is the primary supply of a domain, the simulation behavior for elements in that domain associated with each supply level.

A power domain represents a collection of instances that are powered with the same primary supply and that may share other auxiliary supplies such as isolation and retention supplies. Power states of a power domain can be defined in terms of power states of the domain's available supply sets and related control inputs. Such power states represent various operational modes of the power domain, each of which requires a particular set of power states of its supplies as well as specific control conditions.

A composite domain is a collection of subdomains, each of which shares the same primary supply. The shared primary supply implies that any combination of power states of those subdomains must involve primary supply power states defined with the same set of supply set function values, therefore the creation of a composite domain implicitly restricts the combinations of power states of its subdomains. In addition, power states can be defined on the composite domain in terms of the power states of the subdomains and related control inputs. Such power states identify specific combinations of subdomain power states and potentially restrict the set of legal combinations of subdomain power states.

A group represents a collection of power states defined in terms of power states of other objects within a given scope and its descendant subtree. Power states defined for a group identify and potentially restrict the combinations of power states of the other objects mentioned in their definitions. Multiple independent groups of power states can be defined within a scope.

An HDL module may be used to model an independent design component that can be instantiated in a larger context. In particular, a module may represent a hard macro, an IP block, a subsystem, or the entire system. Power states defined for an HDL module identify and potentially restrict the combinations of power states of objects defined within the HDL module and its descendant subtree. Such power states often represent abstract power states of the whole module that can be used as part of the power interface of the module. Exactly one set of power states may be defined for a given HDL module. Any instance of the module inherits these power states.

Examples

A power domain PD1 may have a power state RUNNING. This power state would require domain PD1's primary supply set to be in a power state in which all supply nets of the primary supply set are on and the current delivered by the power circuit is sufficient to support normal operation. Similarly, a SLEEP power state for domain PD1 would probably require the primary supply set to be in power state in which sufficient voltage and current is provided to maintain the state of registers, although not necessarily enough to support normal operation. A SHUTDOWN power state would typically require the primary supply set to be in the OFF state, and might also require retention and isolation supplies of the domain to be in the ON state.

The state of logic elements may be a relevant aspect to the specification of a domain's power state. For example, for a power domain PD2, its power state might be as follows:

a) UP when:
   1) The logic signal that turns on the domain primary supply switch is asserted.
   2) The logic signal(s) enabling isolation are deasserted.

b) DOWN when:
   1) The logic signal that turns on the domain primary supply switch is deasserted.
   2) If the isolation or retention supplies are switched, the control signals for those supplies are asserted (the power switch is on).
   3) Clock gating enable signals for the domain are deasserted.
   4) Isolation enables for the domain are asserted.
   5) Retention control signals for the domain are asserted.

A domain's power state may also be dependent on the clock period or similar signal interval constraint. For example, a domain in an operational bias mode may need to scale its clock frequency to a slower level to match the slower switching performance supported by the state of the primary supply set. This can be reflected in a bias power state for the domain's primary supply set power state, in which the logic expression includes a constraint on the clock period or duty cycle interval. A domain power state can then be defined that requires its primary supply set to be in that bias power state (see 6.5).

Power states of one domain can be defined in terms of power states of other domains. For example, assume the domain CORE_PD is defined on the root scope of a processor design. In this case, the logic expressions of power states of CORE_PD can reference lower-level power domains such as CACHE_PD, ALU_PD, and FP_PD. Thus, an example power state of FULL_OP for CORE_PD might require that its primary supply set is ON and that the CACHE_PD, ALU_PD, and FP_PD are all in the fully operational mode defined for each one. In contrast, a NON_FP_OP mode for the CORE_PD may be defined similarly, except that it might require domain FP_PD to be in a SLEEP mode.

Power states may also be defined for composite objects such as composite domains, groups, and modules. This enables specification of power states for IP blocks, subsystems, and entire systems.

NOTE—Specification of a given power state for a component does not imply that the power state will necessarily be used in a given system. A legal power state of a given component may be illegal for a given instance of that component. Similarly, an OFF state for a supply set does not imply that the supply set shall actually be a switched supply. The OFF state merely defines the simstate behavior (CORRUPT) in the event that, in a particular implementation, the supply is indeed switched off.

4.7.6 Power states as constraints

For any given object, the legal fundamental power states of that object (together with predefined power state UNDEFINED, if the power state definition for that object is not marked as complete) represent all possible legal power states of that object. For any given pair of objects, the set of the legal possible combinations of power states of the pair consists of the complete cross product of the respective sets of legal possible power states of the individual objects.

Defining a named power state identifies a particular power state so that it can be referenced by name. A named power state can be identified as an illegal state. A set of named power states for a given object can be specified as complete. In both cases, such a specification reduces the set of legal power states of the object to a subset of all possible power states of that object. In this manner, power state definitions that identify illegal power states or identify a set of power states as complete constrain the set of all possible power states of an object.

Power states can be defined in terms of other power states. In particular, power states of one object can be defined in terms of power states of other objects. This creates a hierarchy of power state dependencies. For example:

— power states of a supply set are defined in terms of the supply states of its supply set functions;
— power states of a domain are defined in terms of the power states of its available supply sets;
— power states of a composite domain are defined in terms of power states of its subdomains.

Power states can also be defined for groups (see 6.21) and for modules; such power states can be defined in terms of the power states of any object defined in or below the same scope as the group or instance. Power states of a module are inherited by each instance of the module.

In each of the above cases, named power states defined as illegal, or undefined power states made illegal by specifying that the set of power states is complete, further constrain the combinations of power states of the subordinate objects referenced in the defining expression of the power state.

A more specific power state definition can override a more generic legal power state definition to make that power state illegal in a specific context. For example, a legal power state defined for all instances of a given module can be updated to make that power state illegal for a particular instance of that module, or a legal power state of a domain that is a subdomain of a composite domain can be updated to make that power state illegal in the context of the composite domain. Similarly, the power states for an object in a given module can be updated to identify the set as complete for a particular instance of that module.

The set of legal power states and power state combinations that result when all such constraints are applied is the set of power states and combinations of power states that are expected to be reachable when the system is implemented. The set of reachable power states and combinations of power states imposes a constraint on the implementation of the supply distribution network for the specified power intent. Any implementation of the system shall be able to provide the necessary supply values and combinations to activate each of the reachable power states and combinations of power states.

4.7.7 Power states and power dissipation

Power domains represent a set of elements that consume power from the same primary supply and possibly from related auxiliary supplies. HDL modules in turn consist of one or more power domains. For both domains and modules, the power states of such objects represent various operational modes of the hardware elements involved, and this in turn implies various levels of power dissipation.

Power states defined for power domains or for HDL modules can be augmented with a characterization of the power dissipated by the domain or module instance in each state. This power dissipation specification represents both static power dissipation (leakage) and dynamic power dissipation for each distinct supply set that provides power to the object.

4.7.8 Power state control

For more abstract models, power state definitions should focus on determination of power state based on the actual states of objects rather than on the control mechanisms involved in producing those states. Power state control will ultimately involve detailed hardware protocols as well as software drivers, the full complexity of which cannot be captured in simple combinational power state definitions.

As a consequence, power state definitions for more abstract models should be expressed in terms of the end effect rather than in terms of control signals that cause the end effect. This approach avoids over-constraining the eventual power-management implementation. For example, the state of a given component can be determined by the state of its subcomponents and/or supplies, independent of any control signals that cause those subcomponent states or supply states to occur.

For more concrete models, such as RTL models that are ready for implementation, power state definitions may refer to control signals used in the implementation, such as isolation, retention, and power switch control signals. Even in this case, care should be taken not to over constrain the implementation. For example, configuration UPF need not specify any implementation details, and therefore it may be inappropriate to refer to power switch control signals in the definition of power states in configuration UPF.

4.7.9 Power state changes

The definition of a power state of an object is evaluated whenever an event occurs on one of the objects referenced in the defining expression of that power state. Evaluation of the power state definition determines whether a given power state is active. If a change occurs in the set of active power states for an object, the current power state is updated accordingly.

An event occurs on an object referenced in the defining expression of a power state as follows:

— For a control signal, when the value of the signal changes
— For a term of the form <object>==<state> or <object>!=<state>, when the referenced state of the referenced object becomes active or becomes inactive
— For an interval function, whenever an event occurs on the signal in the interval function

Power state change events propagate up the hierarchy of power state dependencies. For a given object, a change in the set of active power states of that object is an event that triggers re-evaluation of any power state definitions (of the same object or any other object) whose defining expression depends upon the state of the given object.

4.7.10 Power state transitions

A power state transition for a given object starts in one state and ends in another state. As the current state of an object changes, a sequence of actual state transitions occur, from one current power state to the next current power state.

Power state transitions can be defined as named state transitions with the **add_state_transition** command (see 6.7). Named state transitions may be defined as either legal or illegal. Named state transitions occur when the set of active power states changes. This ensures that power state refinement does not hide an occurrence of an illegal transition. It also allows for transitions within a set of states related by refinement, as well as between fundamental states.

A named state transition that is defined as legal is one in which a transition from a certain *from state* to a certain *to state* is allowed. Such a transition may pass through certain intermediate states that occur between the *from state* and the *to state*. A more general transition may be accompanied by more specific transitions that result if the *from state* and/or *to state* are further refined.

A named state transition that is defined as illegal is one in which a direct transition from a *from state* to a *to state* is not allowed. In effect, a transition between the *from state* and the *to state* requires the occurrence of intermediate states. Such a direct transition remains illegal even if the *from state* and/or *to state*s are refined and therefore more specific transitions may occur along with this more general transition.

There is no precedence among transitions; if two transitions start and end at the same time, then both occur in parallel. However, if two transitions start at the same time, and one ends before the other, only the transition that ended is considered to occur; the transition that is still in flight—which, by definition, cannot be an illegal transition—is ignored.

4.8 Simstates

Simstates specify the simulation behavior semantics for a power state. A simstate specifies the level of operational capability supported by a supply set state. The simstate specification provides digital-simulation tools with sufficient information for approximating the power-related behavior of logic connected to the supply set with sufficient accuracy.

Simstates are associated with power states of supply sets and supply set handles. A simstate defines how instances powered by the supply set or supply set handle react to a given power state. In particular, simstates can be associated with power states of the primary supply of a power domain, to define how instances in the power domain that are implicitly connected to that primary supply will behave under various power states of the primary supply.

UPF defines several simstates that can be associated with supply set or supply set handle power states. The simstates defined in UPF are an abstraction suitable for digital simulation. The following simstates are defined (from highest to lowest precedence):

a) **CORRUPT**—The supply set is either off (one or more supply nets in the set are switched off, terminating the flow of current) or at such a low-voltage level that it cannot support switching and the retention of the state of logic nets. It cannot guarantee to maintain even in the absence of activity in the instances powered by the supply.

b) **CORRUPT_ON_ACTIVITY**—The power characteristics of the supply set are sufficient for logic nets to retain their state as long as there is no activity within the elements connected to the supply, but they are insufficient to support activity.

c) **CORRUPT_ON_CHANGE**—The power characteristics of the supply set are sufficient for logic nets to retain their state as long as there is no change in the outputs of the elements connected to that supply.

d) **CORRUPT_STATE_ON_ACTIVITY**—The power characteristics of the supply set are sufficient to support normal operation of combinational logic, but they are insufficient to support activity inside state elements, whether that activity would result in any state change or not.

e) **CORRUPT_STATE_ON_CHANGE**—The power characteristics of the supply set are sufficient to support normal operation of combinational logic, and they are sufficient to support activity inside state elements, but they are insufficient to support a change of state for state elements.

f) **NORMAL**—The power characteristics of the supply set are sufficient to support full and complete operational (switching) capabilities with characterized timing.

The predefined power states for a supply set have corresponding simstates. The simstate for power state **ON** is **NORMAL**. The simstate for power state **OFF** is **CORRUPT**. The simstate for power state **ERROR** is **CORRUPT**. There is no simstate defined for power state **UNDEFINED**.

Simstate simulation semantics for a supply set are applied to instances implicitly connected to a supply set unless simstate behavior has been disabled (see 6.52).

NOTE 1—When greater accuracy is desired or required, a mixed signal or full-analog simulation can be used. Since analog simulations already incorporate power, this format provides no additional semantics for analog verification.

Simulation results reflect the implemented hardware results only to the extent that the UPF simstate specification for a given power state of a supply set is correctly specified. For example, if verification is performed with simulation of a supply set in a power state specified as having a **CORRUPT_ON_ACTIVITY** simstate, but the implementation is more accurately classified as **CORRUPT_STATE_ON_CHANGE**, the simulation results will differ.

NOTE 2—In this example, the inaccuracy in simstate specification is conservative relative to the implemented hardware behavior. However, in other situations, inaccurate specifications can be optimistic, resulting in errors in the implemented hardware that simulation failed to expose.

4.9 Power intent specification

4.9.1 Successive refinement

Design and implementation of a power-managed system using UPF proceeds in stages. During the design phase, a UPF-based specification of the power intent may be developed incrementally, first at the IP block level, and later at the system level. During implementation, UPF commands are added to drive implementation details, and a series of implementation steps map the design and the UPF commands into the final implementation (see Figure 2).

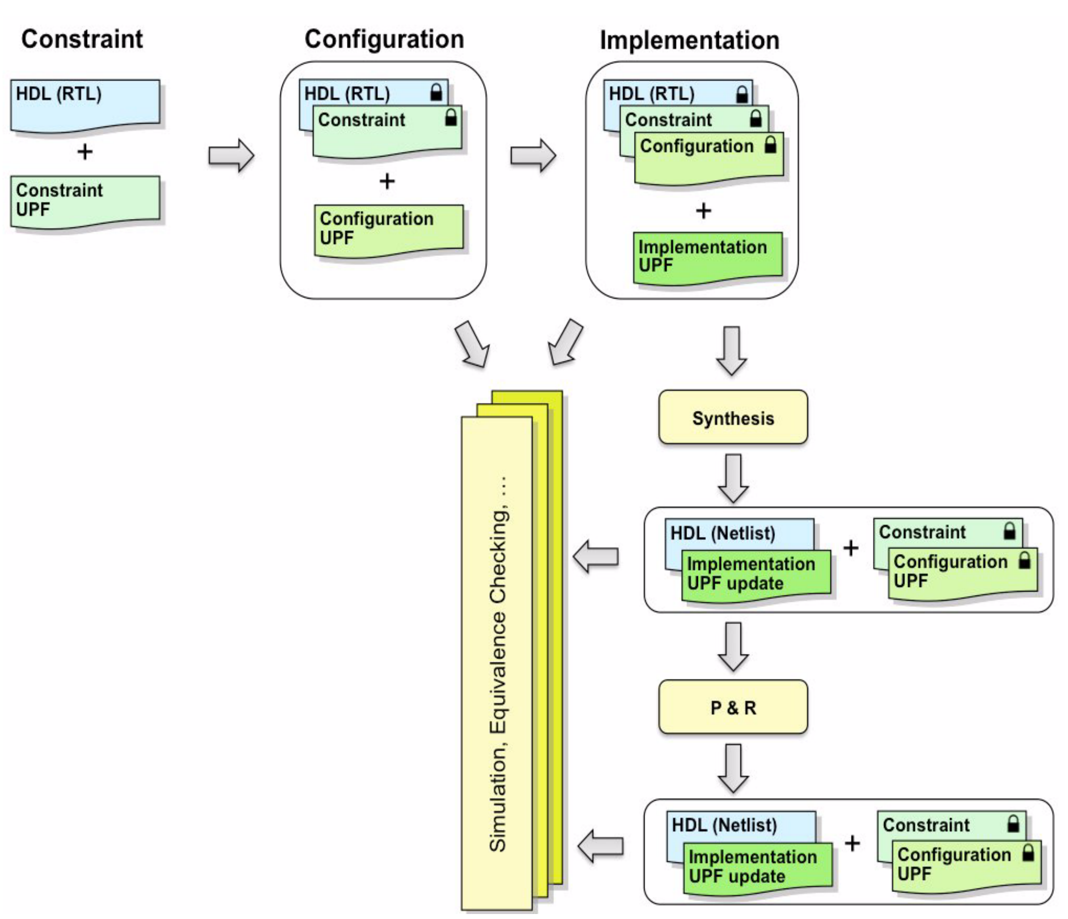


Figure 2—Successive refinement of power intent

The power intent specification for an IP block to be used in a larger design typically defines the power interface to the block and the power domains within the block. This specification also typically includes constraints on the use of the block in a power-managed environment. These constraints include (at least) the following:

a) The atomic power domains in the design.
These can be composed but not split during implementation. [Use **create_power_domain -atomic** (see 6.20).]

b) The state variables that need to have their values retained if a given power domain is powered down.
This does not involve specifying how such retention would be controlled. [Use **set_retention_elements** (see 6.50).]

c) The clamp values of signals that would need to be isolated if a given power domain is powered down.
This does not involve specifying how isolation is to be controlled. [Use **set_port_attributes -clamp_value** (see 6.47).]

d) The legal power states and power state transitions of the IP block's power domains.
This need not involve specifying absolute voltages for the power supplies involved. [Use **add_power_state** (see 6.5) and **add_state_transition** (see 6.7).]

A power intent specification containing such basic information about an IP block is often referred to as *constraint UPF*, or sometimes as the *platinum UPF*.

When an IP block is being prepared for use in a given system, information may be added to the specification to reflect the specific requirements of the block in the context of the system. For example, an instance of the block can be used in a manner that will definitely require isolation, level-shifting, retention, or repeater cell insertion. These strategies can be added to the constraint UPF for the block in order to configure the power intent of the block for use in this system. Such strategies impose a requirement to insert specified power-management cells for an instance of the IP block and typically include information about how such power-management cells are controlled.

A power intent specification containing this level of information is often referred to as *configuration* UPF, or sometimes as the *golden* UPF.

To drive implementation of a power-managed design, information may be added to the specification to define the power-distribution network for the system and the control logic for power-management cells. A power intent specification containing this kind of information is often referred to as *implementation* UPF, or sometimes as the *silicon* UPF.

4.9.2 Bottom-up specification

4.9.2.1 Introduction

While implementing a system it may be required to implement an instance separately from the top-level scope with the intention to integrate this block back into the system later in the flow. This flow style is often referred to as a *bottom-up flow*.

If using a bottom-up flow some considerations regarding UPF partitioning must be made. In particular the implementation of a lower level instance will be done without the parent scope being present. Therefore the block UPF power intent must be self-sufficient (see 4.9.2.2); in that it cannot rely on power intent defined in an ancestor scope and it cannot define power intent that is to be implemented in an ancestor scope.

In addition to a block instance requiring self-contained power intent, the block needs to be defined as a soft macro (see 4.9.2.4). This ensures that the block is not affected by ancestor level power intent which is not available during block implementation. By defining an instance to be a soft macro, the evaluation of certain UPF power intent commands are affected due to a macro being treated as a leaf cell boundary.

4.9.2.2 Self-contained UPF

In order for a block to be implemented standalone from its parent scope the UPF for this block must completely define the power intent. The power intent for a block instance can be deemed to be self-contained when:

a) It does not require power intent defined in an external scope to complete the power intent. However ancestor level power intent may still change or add power intent, unless the block is identified as a soft macro (see 4.9.2.4).

b) It defines its own top-level domain (e.g., create_power_domain -elements {.}).

c) It does not reference any objects defined in a parent context.

d) It does not rely on the visibility of the real drivers and receivers respectively for the block primary inputs and outputs, i.e., all input and output ports have the required **driver_supply** and **receiver_supply** attributes respectively annotated to represent the assumptions about the supplies of external logic in the environment that drives the block inputs and receives the block outputs.

e) It does not attempt to change external power intent implementation, such as with the insertion of isolation cells.

4.9.2.3 Leaf cells

A leaf cell is a cell that is considered as having no descendants with regards to a top-level context.

For any leaf cell, the ports on the boundary are treated as drivers and receivers (see 4.4.3). In particular:

— Driver/receiver supply analysis in the parent context of a leaf cell instance (e.g., for evaluating filters of strategies) stops at the ports of the leaf cell instance and uses the output driver supply and input receiver supply attributes of those ports rather than the actual driving/receiving supplies inside the leaf cell.

— Driver/receiver supply analysis inside a soft macro cell instance (see 4.9.2.4) stops at the ports of the macro cell and uses the input driver supply and output receiver supply attributes of those ports rather than the actual driving/receiving supplies in the parent context in which the macro cell is instantiated.

The effect of some UPF commands are limited by a leaf cell boundary:

a) **find_objects** searches within a leaf cell instance.

b) Global supply net availability does not extend into a leaf cell instance.

c) During isolation or level-shifting insertion, location fanout evaluation terminates at the leaf cell boundary.

If a leaf cell instance has a UPF power intent specification, then in addition a parent context's power intent shall not affect the cell instance power intent;

d) It shall be an error if the UPF power intent of an ancestor

   1) uses set_scope to scope into the leaf cell.

e) It shall be an error if the UPF power intent of an ancestor context contains a command that

   1) defines new objects in the cell instance.
   2) refines the definitions of existing objects in the cell instance.
   3) inserts isolation or level-shifting cell into the cell instance (e.g., -location other).
   4) attempts to add/modify power domains in the cell instance.
   5) attempts to add/refine strategies of domains in the cell instance.
   6) attempts to add/refine power states of objects defined in the cell instance.

4.9.2.4 Macro cells

4.9.2.4.1 Introduction

A macro is a module or an instance of a module that has already been implemented. Each instance of a macro is a leaf cell and therefore defines a leaf cell boundary (see 4.9.2.3).

4.9.2.4.2 Hard macro cells

A macro cell typically has a Liberty model that defines its interface, including supply ports and the related supplies for its logic ports. The Liberty model may also include information about embedded power switches and the conditions under which those switches are on or off.

For verification purposes, a macro cell may be represented by a behavioral model that describes the behavior of the cell without representing the internal details. Since the internal structure of the implementation is not represented in detail in the behavioral model, the model is effectively a black box; only the interface of the model is visible to the parent context.

UPF may be specified for the macro to represent power intent that is not described in the functional model, however this intent is descriptive of the already implemented logic and therefore no further implementation of this cell should be needed.

This style of macro cell is referred to as a hard macro. The attribute {**UPF_is_hard_macro** TRUE} associated with a model indicates that the model is a hard macro. All instances of a hard macro model are treated as hard macro cell instances.

A macro defined using the Liberty **is_macro_cell** attribute will implicitly set the {**UPF_is_hard_macro** TRUE} attribute on the model (see 5.6).

4.9.2.4.3 Soft macro cells

A macro cell may also be represented by the original RTL and UPF from which its implementation was (or will be) derived. This style of macro cell is referred to as a soft macro. The attribute {**UPF_is_soft_macro** TRUE} associated with a model indicates that the model is a soft macro. All instances of a soft macro model are treated as soft macro instances.

A soft macro instance is considered to have a terminal boundary that restricts the scope of the object. As such, power intent objects expressed in an ancestor (such as domains and global supply sets) are not available to the block and therefore the power intent must be supplied explicitly (see 4.9.2.2).

4.9.3 File structure

For maximum reuse, it may be appropriate to keep constraint, configuration, and implementation UPF commands in separate files. The **load_upf** command (see 6.32) can be used to compose the files for a particular context.

For example, an IP block with a corresponding constraint UPF description might be configured for use in a given system by creating a configuration UPF file for it. The configuration UPF file would load the constraint UPF for the IP block and then continue with additional commands defining or updating the isolation, level-shifting, retention, and repeater strategies required for this configuration of the IP block. Different configuration UPF files can be constructed based on the same constraint UPF to define different configurations of the same IP block for use in different situations.

For implementation of the design, an implementation UPF file may be constructed by loading the configuration UPF for the various IP blocks involved in the system and then adding implementation details, such as supply ports, nets, sets, power switches, port attributes, and supply connections. Different implementation UPF files can be constructed using the same configuration UPF files to evaluate or verify alternative implementations.

For each implementation step, tools may update the implementation UPF to document the additions made to the design in that step to implement the power intent. To keep the implementation updates separate from the input UPF specification, a tool may generate an output UPF file that loads the input UPF file and then adds UPF command updates as required. Successive implementation steps may choose to append to this update file or generate a new update file that loads the previous one.

4.9.4 Tool flow

A UPF-based tool flow typically begins with verification of the design together with its power intent. Verification goals include the following:

- To confirm that IP blocks with UPF constraints are being used correctly in the design.
- To confirm that the logical, technology-independent aspects of the power intent are working as expected and enable the design to function correctly.
- To confirm that the technology-dependent implementation details specified in UPF correctly implement and enable the logical behavior of the power-management architecture.

Verification can begin as soon as the logical aspects of the design's power intent are specified in UPF. These include power domains, isolation and retention strategies, control inputs, power states of the domains, and power states and simstates of each domain's primary supply set. At this stage, isolation and retention strategies that do not have explicitly specified supplies may be modeled as having always-on supplies (see G.1.1), under the assumption that the implementation, when completed, will ensure that they are provided with supplies that are on whenever necessary. This assumption avoids having to make implementation decisions too early in the verification process.

When technology-specific implementation aspects of the design's power intent have been specified, verification can focus on the correctness and completeness of power-management implementation. These implementation aspects include supply ports, nets, switches, and their connections to supply set functions, level-shifting and repeater strategies, mapping of strategies to particular library cells, and port states and power state tables. At this stage, the assumption of an always-on supply for isolation and retention strategies no longer applies; verification will check that the actual supplies provided to those strategies as well as level-shifter and repeater strategies are indeed on when required. Other implementation-related checks are performed at this stage as well.

After verification of the design with its power intent has been completed, a series of implementation steps occur in which the RTL design is reduced to a gate-level implementation and the power intent is integrated into that implementation. After each implementation step, power-aware verification can be performed again, using the design representation output by that stage along with the UPF description corresponding to that design representation (see Figure 2).

The power intent expressed in UPF can be implemented incrementally in successive steps. Each step may add implementation details, such as power-management cells, control logic, or supply distribution networks. The design itself may also evolve during implementation, even after the RTL stage, as a result of implementation steps such as test insertion.

Implementation can be incremental at various levels of granularity as follows:

- By aspect: isolation, level-shifting, retention, repeaters, control logic, power distribution
- By command: isolation strategy A, isolation strategy B, etc.
- By element to which a command applies: isolation for port `p1`, for port `p2`, etc.

For any given tool run, the tool needs to know the following:

a) What part of the UPF power intent specification is supposed to be implemented already, and

b) What part of the UPF power intent specification is to be included in the processing done by this tool.

This standard does not define how the preceding information is made available to a tool; this is tool/flow information that is outside the scope of the standard. Typically, such information would be provided to the tool either explicitly via command-line arguments or other control inputs, or implicitly as part of the specification of the tool itself.

A tool also shall be able to determine what part of the UPF specification has been implemented so far. This standard defines a method for documenting what has been done so far to implement the power intent, by identifying ports, nets, and instances in the design that represent implementations of UPF commands.

5. Language basics

5.1 UPF is Tcl

UPF is based on Tool Command Language (Tcl). UPF commands are defined using syntax that is consistent with Tcl, such that a standard Tcl interpreter can be used to read and process UPF commands.

Compliant processors reading UPF files use full Tcl interpreters to process the UPF files. Compliant processors shall use Tcl version 8.4 or above. The following also apply:

— UPF power intent commands are executed in the order of occurrence, just as Tcl commands are executed and return values can be used by subsequent commands.

— The only UPF commands that support regular expressions are **find_objects** (see 6.30) and **query_upf** (see 11.1.2).

— All of the commands and techniques of Tcl may be used, including procs and libraries of procs. However, the procs and libraries of procs should ultimately only rely on UPF commands for design information.

— **find_objects** (see 6.30) shall be the only source used to programmatically access the HDL when defining the power intent. The processing of information returned by **find_objects** using standard Tcl commands (Tcl language syntax summary [B5]), such as `regexp`, is allowed.

— UPF is intended to be used across many tools, so it is erroneous to use proprietary tool-specific commands when constructing power intent.

— Once the Tcl processing has completed, the end result can be expressed as a series of UPF commands.

Libraries used for design or methodology standardization or ease of expression that define additional procs are considered to be part of the design file and need to be visible to any processor interpreting the UPF file.

5.2 Conventions used

5.2.1 Introduction

Each UPF command in Clause 6 and Clause 7 consists of a command keyword followed by one or more parameters. All parameters begin with a hyphen (-). The meta-syntax for the description of the syntax rules uses the conventions shown in Table 1.

Table 1—Document conventions

Visual cue	Represents
`courier`	The `courier` font indicates UPF or HDL code. For example, the following line indicates UPF code: `create_power_domain PD1`
bold	The **bold** font is used to indicate keywords that shall be typed exactly as they appear. For example, in the following command, the keyword **create_power_domain** shall be typed as it appears: **create_power_domain** *domain_name*
italic	The *italic* font represents user-defined UPF variables. For example, a supply net shall be specified in the following line (after the **connect_supply_net** keyword): **connect_supply_net** *net_name*
list	*list* (or *xyz_list*) indicates a Tcl list, which is denoted with curly braces **{….}** or as a double-quoted string of elements **"…."**. When a list contains only one non-list element (without special characters), the curly braces can be omitted, e.g., `{a}`, `"a"`, and `a` are acceptable values for a single element. See also 5.3.4.
xyz_ref	*xyz_ref* can be used when a symbolic name (i.e., using a handle) is allowed as well as a declared name, e.g., *supply_set_ref*.
time_literal	*time_literal* indicates a SystemVerilog or VHDL `time_literal`.
* asterisk	An asterisk (`*`) signifies that a parameter can be repeated. For example, the following line means multiple acknowledge delays can be specified for this command: [**-ack_delay** **{***port_name delay***}**]*
[ ] square brackets	Square brackets indicate optional parameters. If an asterisk (`*`) follows the closing bracket, the bracketed parameter may be repeated. For example, the following parameter is optional: [**-elements** *element_list*] The following is an example of optional parameter that can be repeated: [**-ack_port** **{***port_name net_name* [**{***logic_value***}**]**}**]*
[ ] bold square brackets	Bold square brackets are required. For example, in the following parameter, the bold square brackets (surrounding the `0`) need to be typed as they appear: *domain_name***.***isolation_name***.***isolation_supply***[0]**
{ } curly braces	Curly braces (`{ }`) indicate a parameter list that is required. In some (or even many) cases, they have (or are followed by) an asterisk (`*`), which indicates that they can be repeated. For example, the following shows one or more control ports can be specified for this command: {**-control_port** **{***port_name***}**}*
{ } bold curly braces	Bold curly braces are required, unless the argument is already a Tcl list. For example, in the following parameter, the bold curly braces need to be typed as they appear: [**-off_state** **{***state_name* **{***boolean_expression***}}**]* In cases where variable substitution is needed, Tcl's list command can be used, e.g., **-off_state** [list *$state_name* [list *$expression*]]
< > angle brackets	Angle brackets (< >) indicate a grouping, usually of alternative parameters. For example, the following line shows the **power** or **ground** keywords are possible values for the **-type** parameter: **-type** <**power** | **ground**>
| separator bar	The separator bar (`|`) character indicates alternative choices. For example, the following line shows the **in** or **out** keywords are possible values for the **-direction** parameter: **-direction** <**in** | **out**>

This standard uses a minimal amount of color to enhance readability. The coloring is not essential and does not affect the accuracy of this standard when viewed in pure black and white. Color is used as follows:

— Syntactic keywords and tokens in the formal language definitions are shown in **boldface-red text**.

— Command arguments that can be provided incrementally (*layered*) are shown in **boldface-green text**. See also 5.10.

— Syntactic keywords and tokens that have been explicitly identified as legacy or deprecated constructs (see 6.2) are shown in **brown text**.

5.2.2 Word usage

In this document, the word *shall* is used to indicate a mandatory requirement. The word *should* is used to indicate a recommendation. The word *may* is used to indicate a permissible action. The word *can* is used for statements of possibility and capability.

The words *must* and *will* do not indicate requirements that must be strictly followed in order to conform to the standard. The word *must* is used to describe unavoidable situations; the word *will* is only used in statements of fact.

5.3 Lexical elements

5.3.1 Introduction

Names created in UPF should not conflict with HDL reserved words.

Command names, parameter names, and their values are case-sensitive.

5.3.2 Identifiers

Identifiers adhere to the following rules:

a) The first character of an identifier shall be alphabetic.

b) All other characters of an identifier shall be alphanumeric or the underscore character (_).

c) Identifiers in UPF are case-sensitive.

5.3.3 Names

5.3.3.1 General

Names identify objects in the design and in the power intent specification.

5.3.3.2 Simple names

A simple name is a single identifier. An identifier is used when creating a new object in a given scope; the identifier becomes the simple name of that object.

In a given scope, a given simple name may only be defined once, with a unique meaning; it shall be an error if two objects are declared in the same scope with the same simple name.

A simple name, optionally followed by an index or record field specification as appropriate for the type of an object in a given HDL context, is an object name. An object name can be used to refer to an existing object or part of an existing object that is declared in the current scope. Object names also refer to objects defined in UPF that do not exist in a scope of the hierarchy.

The simple name of an instance in a given scope is an instance name.

The simple name of any of the following objects is defined within a unique global scope:

— an HDL model
— a UPF power model
— a UPF power-management cell
— a UPF VCT
— a UPF Retention Element List

An HDL model or instance is a scope in which simple names of the following may be defined:

— HDL ports, nets, instances, processes
— UPF ports, nets, switches, power state tables, supply sets, power domains

A UPF power switch is a scope in which simple names of switch control ports, input supply ports, input states, and switch supply sets may be defined.

A UPF power state table is a scope in which simple names of PST states may be defined.

A UPF supply set is a scope in which simple names of supply set functions, power states, and state transitions may be defined.

A UPF power domain is a scope in which simple names of supply sets, strategies, power states, and state transitions may be defined.

A UPF strategy is a scope in which simple names of various supply sets and control signals are predefined.

The following names are predefined in certain contexts:

a) Predefined names in a power domain scope
    1) **primary**
b) Predefined names in a power switch scope
    1) **switch_supply**
c) Predefined names in a level-shifter strategy scope
    1) **input_supply**
    2) **output_supply**
    3) **internal_supply**

d) Predefined names in an isolation strategy scope
   1) **isolation_supply**
   2) **isolation_signal**

e) Predefined names in a retention strategy scope
   1) **retention_supply**
   2) **primary_supply**
   3) **save_signal**
   4) **restore_signal**
   5) **UPF_GENERIC_CLOCK**
   6) **UPF_GENERIC_DATA**
   7) **UPF_GENERIC_ASYNC_LOAD**
   8) **UPF_GENERIC_OUTPUT**

f) Predefined names in a repeater strategy scope
   1) **repeater_supply**

Each name is defined within a particular scope.[16]

5.3.3.3 Dotted names

A dotted name is a compound name designating a UPF object. A dotted name is made up of simple names separated by `.` characters.

A dotted name is used to refer to a strategy associated with a power domain, a supply set associated with a strategy or a power domain, or a function of a supply set. A dotted name for a supply set associated with a strategy or domain is called a *supply set handle*. A dotted name for a supply set function is called a *supply net handle*.

— Power-domain strategy names
```
<domain name> . <strategy name>
```
— Supply set handles
```
<domain name> . <supply set name>
<domain name> . <strategy name> . <supply set name>
```
— Supply net handles
```
<supply set name> . <function name>
<domain name> . <supply set name> . <function name>
<domain name> . <strategy name> . <supply set name> . <function name>
```

A dotted name is also an object name.

[16] In this clause, the term *scope* refers to any region in which names can be defined, not just to instances in the logic hierarchy.

5.3.3.4 Hierarchical names

A hierarchical name is a name that refers to an object declared in a non-local scope. A hierarchical name consists of an optional leading `/` character, followed by a series of one or more instance names, each followed by the hierarchy separator character `/`, followed by an object name.

A hierarchical name that starts with an instance name is a scope-relative hierarchical name. A scope-relative hierarchical name is interpreted relative to the current scope. The first instance name is the name of an instance in the current scope; each successive instance name is the name of an instance declared in the scope of the previous instance. The trailing object name is the simple name or dotted name of an object declared in the scope of the last instance. A scope-relative hierarchical name is also called a *rooted name*.

A hierarchical name that starts with a leading `/` character is a design-relative hierarchical name. A design-relative hierarchical name is interpreted relative to the current design top instance by removing the leading `/` character and interpreting the remainder as a rooted name in the scope of the current design top instance.

5.3.3.5 Name references

Many command arguments require references to object names, such as the names of instances, ports, registers, nets, etc., in the design, or the names of power domains, strategies, supply sets, supply nets, etc., in the power intent. Unless otherwise specified or contextually restricted, an object name reference can be a simple name, a dotted name, or a hierarchical name. In particular, a supply set handle is a form of supply set name and a supply net handle is a form of supply net name. In the absence of any statement to the contrary, a supply set handle can be used wherever a supply set name may appear, and a supply net handle can be used wherever a supply net name may appear.

5.3.4 Lists and strings

A Tcl list is an ordered sequence of zero or more elements, where each element can itself be a list. In Tcl, a string can be thought of as a list of words.

Tcl strings can be specified in two different ways: by enclosing the words within double-quotes (`""`) or between curly braces (`{}`). Upon finding a list of words within double-quotes, Tcl continues to parse the string, looking for variable (strings started with `$`), command (strings between square brackets `[]`), and back-slash (strings contain `\`) substitutions. To use any of the special characters within design object names, first wrap them in curly braces (`{}`). Upon finding a list of words between curly braces, Tcl treats the list as a literal list of words, preventing further processing on the list before it is used.

Therefore, in the syntax for UPF, the construct **-option** *xyz_list* can be satisfied by any of the following, when no special characters are used in the object names:

```
-option foo
-option "foo"
-option "foo bar bat"
-option {foo}
-option {foo bar bat etc.}
```

5.3.5 Special characters

Special lexical elements (see Table 2) can be used to delimit tokens in the syntax.

Table 2—Special characters

Type	Character
Logic hierarchy delimiter	/
Escape character	\ (only escapes the next character)
Bus delimiter, index operator, or within a `regex`	[]
Range separator (for bus ranges)	:
Record field delimiter	.

When Tcl special characters need to be used literally for design object names, always escape the special character or wrap the name with {}, even if a single value is used, to protect from Tcl interpretation, e.g., `-elements [list foo {foo/bar} a\[0\]]`.

5.4 Boolean expressions

A Boolean expression may be used to define a control condition or a supply state. A Boolean expression may include references to the following.

a) VHDL names, values, and literals of the following types or any subtype thereof:

   `std.Standard.Boolean`

   `std.Standard.Bit`

   `std.Standard.Real` for voltage values

   `std.Standard.Time` for use with the interval function

   `ieee.std_logic_1164.std_ulogic`

   `ieee.UPF.state`

b) SystemVerilog names, values, and literals of the following types:

   `reg`

   `wire`

   `Bit`

   `Logic`

   `time_literal` for use with the `interval` function

   `real`, `shortreal` for voltage values

A VHDL or SystemVerilog name may also be the name of an element of any composite type object provided the element itself is of a supported type.

A Boolean expression may also contain special expression forms for referring to power states (see 6.5).

In certain commands, logic values X, 0, 1, Z can be specified. These represent values of a predefined logic type in the relevant hardware description language. For VHDL, the predefined logic type is type ieee.std_logic_1164.std_ulogic, or any subtype thereof. For SystemVerilog, the predefined logic type is type Logic.

A name of an object referred to in a Boolean expression may be prefixed by a pathname identifying the instance in the scope of which the name is declared. Any such pathname is interpreted relative to the current scope when the command defining the expression is executed. If no pathname prefix is present, the name shall refer to an object declared in the current scope.

In a Boolean expression used as a supply expression in the definition of a power state of a supply set (handle), the name of any function of that supply set (handle) may be referred to directly without a prefix, unless such a reference would be ambiguous.

In a Boolean expression used as a logic expression in the definition of a power state of a power domain, the name of any supply set handle associated with that power domain may be referred to directly without a prefix, unless such a reference would be ambiguous.

A Boolean expression may include the operators shown in Table 3, which map to their corresponding equivalents in SystemVerilog or VHDL, as appropriate for the objects involved in each subexpression.

Table 3—Boolean operators

Operator	SystemVerilog equivalent	VHDL equivalent	Meaning
!	`!`	`not`	Logical negation
~	`~`	`not`	Bit-wise negation
<	`<`	`<`	Less than
<=	`<=`	`<=`	Less than or equal
>	`>`	`>`	Greater than
>=	`>=`	`>=`	Greater than or equal
==	`==`	`=`	Equal
!=	`!=`	`/=`	Not equal
&	`&`	`and`	Bit-wise conjunction
^	`^`	`xor`	Bit-wise exclusive disjunction
|	`|`	`or`	Bit-wise disjunction
&&	`&&`	`and`	Logical conjunction
||	`||`	`or`	Logical disjunction

A Boolean expression shall be provided as a string, as indicated in the syntax for each command in which a Boolean expression can appear. Subexpressions may be grouped with parentheses (`()`). Logical operators have lowest precedence; bit-wise operators have next higher precedence; relational operators have next higher precedence; negation operators have highest precedence.

A Boolean expression or subexpression is considered to evaluate to the logical value *True* if evaluation of the expression (according to the semantics of the VHDL or SystemVerilog operators and types involved, as appropriate) results in a bit or logic value of `1` or a Boolean value of *True*; otherwise it is considered to evaluate to the logical value *False*.

A Boolean expression may contain references to objects in different language contexts provided that any given subexpression that evaluates to a logical (*True*/*False*) value contains only references to one language context. Logical negation, conjunction, and disjunction of logical values shall be performed according to standard Boolean logic semantics and need not be implemented with language-specific operators.

A simple expression is a Boolean expression containing an optional negation operator (**!** or ~), followed by optional white space and a single object name.

Examples

```
{ top/sv_inst/ena == 1'b1 && top/vhdl_inst/ready == '0' }
{ supply1.state == FULL_ON && supply1.voltage > 0.8 }
{(top/sv/wall.supply[0] != FULL_ON) || (top/vhdl/battery.supply(1) ==
    UNDEFINED)}
```

5.5 Object declaration

All UPF commands are executed in the current scope, except as specifically noted.

As a result, most objects created by a UPF command are created in the current scope within the design; therefore, the names of those objects shall not conflict with a name that is already declared within the same scope.

Some UPF objects are implicitly created. *Implicitly created objects* result from implied or inferred semantics and are not the direct result of creating a named UPF object. For example, supply nets are routed throughout the extent of a power domain as needed to implement the implicit and automatic connection semantics. This routing results in the creation of implicit supply ports and supply nets. UPF automatically names implicitly created objects to avoid creating a name conflict. The **name_format** command (see 6.37) can be used to provide a template for some implicitly created objects (such as isolation). Supply nets may be implicitly created and connected to supply ports, and logic nets may be implicitly created and connected to logic ports (see 4.5.3).

UPF objects may have record fields. These records comprise a name and a set of zero or more values. Record field names are in a local name space of the UPF object, e.g., a power domain may have strategies and supply set handles. Strategies themselves may also have supply set handles.

The **.** character is the delimiter for the hierarchy of UPF record fields, e.g., `top/a/PDa.MY_SUPPLY_SET` refers to the supply set `MY_SUPPLY_SET` in power domain `PDa` in the logical scope `top/a`.

5.6 Attributes of objects

UPF supports the specification of *attributes*, or properties, of objects in a design. These attributes provide information that supports or affects the meaning of related UPF commands. Such attributes can also be defined with HDL attribute specifications in design code or with Liberty attribute specifications in a Liberty model.

Table 4 enumerates the attributes that have a predefined meaning in UPF and for each attribute, the UPF command that can be used to define that attribute.

Table 4—Attribute and command correspondence

UPF predefined attribute name	Attribute value specification	Equivalent UPF command arguments	See
UPF_clamp_value	<**0** | **1** | **Z** | **latch** | **any** | *value*>	**set_port_attributes -clamp_value**	6.47
UPF_sink_off_clamp_value	<**0** | **1** | **Z** | **latch** | **any** | *value*>	**set_port_attributes -sink_off_clamp_value**	6.47
UPF_source_off_clamp_value	<**0** | **1** | **Z** | **latch** | **any** | *value*>	**set_port_attributes -source_off_clamp_value**	6.47
UPF_pg_type	*pg_type_value* (see 4.5.4.6)	**set_port_attributes -pg_type**	6.47
UPF_related_power_port	*supply_port_name*	**set_port_attributes -related_power_port**	6.47
UPF_related_ground_port	*supply_port_name*	**set_port_attributes -related_ground_port**	6.47
UPF_related_bias_ports	*supply_port_name_list*	**set_port_attributes -related_bias_ports**	6.47
UPF_driver_supply	*supply_set_ref*	**set_port_attributes -driver_supply**	6.47
UPF_receiver_supply	*supply_set_ref*	**set_port_attributes -receiver_supply**	6.47
UPF_literal_supply	*supply_set_ref*	**set_port_attributes -literal_supply**	6.47
UPF_feedthrough	<**TRUE** | **FALSE**>	**set_port_attributes -feedthrough**	6.47
UPF_unconnected	<**TRUE** | **FALSE**>	**set_port_attributes -unconnected**	6.47
UPF_is_isolated	<**TRUE** | **FALSE**>	**set_port_attributes -is_isolated**	6.47
UPF_is_analog	<**TRUE** | **FALSE**>	**set_port_attributes -is_analog**	6.47
UPF_retention	<**required** | **optional**>	**set_design_attributes -attribute {UPF_retention required}** **set_design_attributes -attribute {UPF_retention optional}**	6.40
UPF_simstate_behavior	<**ENABLE** | **DISABLE**>	**set_design_attributes -attribute {UPF_simstate_behavior ENABLE}** **set_design_attributes -attribute {UPF_simstate_behavior DISABLE}**	6.40
UPF_is_soft_macro	<**TRUE** | **FALSE**>	**set_design_attributes -is_soft_macro**	6.40
UPF_is_hard_macro	<**TRUE** | **FALSE**>	**set_design_attributes -is_hard_macro**	6.40
UPF_switch_cell_type	<**fine_grain** | **coarse_grain**>	**set_design_attributes -switch_type fine_grain** **set_design_attributes -switch_type coarse_grain**	6.40

The attributes in Table 4 all take values that are string literals. Where a list of names is required, the names in the list should be separated by spaces and without enclosing braces (`{}`). These attributes can also be specified using the attribute mechanism in SystemVerilog code or using attribute specifications in VHDL code. To attach a attribute to an object in a VHDL context, the attribute shall be declared first, with a data type of `STD.Standard.String` (or the equivalent), before any attribute specification for that attribute.

For determination of precedence (see 5.7), attributes specified in HDL code are treated as if they were implicitly specified using the UPF command **set_port_attributes -model -ports** (for port attributes) or the UPF command **set_design_attributes -models** (for design attributes).

Some of these attributes may also be implied by attributes in a Liberty model. Specifically, the following Liberty attributes imply definition of the corresponding UPF predefined attribute:

Liberty attribute name	implies	**UPF predefined attribute name**
pg_type		UPF_pg_type
related_power_pin		UPF_related_power_port
related_ground_pin		UPF_related_ground_port
related_bias_pins		UPF_related_bias_ports
short		UPF_feedthrough
is_hard_macro		UPF_is_hard_macro
is_isolated		UPF_is_isolated
is_analog		UPF_is_analog
switch_cell_type		UPF_switch_cell_type

For determination of precedence (see 5.7), attributes specified in Liberty models are treated as if the corresponding UPF attribute name were implicitly specified using the UPF command **set_port_attributes -model -ports** (for port attributes) or the UPF command **set_design_attributes -models** (for design attributes).

Certain attributes represent characteristics of a module or cell that apply universally to all instances of that module or cell. Such attributes are called characteristic attributes. The following predefined attributes are always characteristic attributes:

UPF_pg_type

UPF_related_power_port

UPF_related_ground_port

UPF_related_bias_ports

UPF_feedthrough

UPF_unconnected

UPF_is_isolated

UPF_is_analog

UPF_is_hard_macro

UPF_is_soft_macro

UPF_retention

UPF_switch_cell_type

In addition, any attribute specified either explicitly or implicitly with **set_port_attributes -model** or **set_design_attributes -models** is a characteristic attribute.

Non-characteristic attributes are overridable as specified by the precedence rules for attribute specifications (see 5.7). Characteristic attributes are non-overridable.

NOTE—The above definitions imply that any attribute derived from a Liberty attribute or specified in an HDL model cannot be overridden by a higher precedence attribute specification in UPF (see 5.7).

It shall be an error if any of the attributes in Table 4 is defined multiple times with different values for the same object, regardless of whether the attribute is defined as an HDL attribute or using UPF commands or both.

Examples

A port-supply relationship can be annotated in HDL using the following attributes:

Attribute name: **UPF_related_power_port** and **UPF_related_ground_port**.

Attribute value: "*supply_port_name*", where *supply_port_name* is a string whose value is the simple name of a port on the same interface as the attributed port.

SystemVerilog or Verilog-2005 attribute specification:

```
(* UPF_related_power_port = "my_VDD",

   UPF_related_ground_port = "my_VSS" *)

output my_Logic_Port;
```

VHDL attribute specification:

```
attribute UPF_related_power_port : STD.Standard.String;

attribute UPF_related_power_port of my_Logic_Port : signal is
"my_VDD";

attribute UPF_related_ground_port : STD.Standard.String;

attribute UPF_related_ground_port of my_Logic_Port : signal is
"my_VSS";
```

Attribute name: **UPF_related_bias_pin**.

Attribute value: "*supply_port_name_list*", where *supply_port_name_list* is a string whose value is a space-separated list of one or more simple names of port(s) on the same interface as the attributed port.

SystemVerilog or Verilog-2005 attribute specification:

```
(* UPF_related_bias_ports = "my_VNWELL my_VPWELL" *)

output my_Logic_Port;
```

VHDL attribute specification:

```
attribute UPF_related_bias_ports : STD.Standard.String;

attribute UPF_related_bias_ports of my_Logic_Port : signal

is "my_VNWELL my_VPWELL";
```

The same attributes can be specified in UPF, using the **set_port_attributes** command and its generic **-attribute** option, or they can also be specified in UPF using the **set_port_attributes** command and its specific options **-related_power_port**, **-related_ground_port**, and **-related_bias_ports**, respectively (see 6.47).

Isolation clamp value port properties can be annotated in HDL using the following attributes:

Attribute name: **UPF_clamp_value**

Attribute value: <**0** | **1** | **Z** | **latch** | **any** | *value*>

SystemVerilog or Verilog-2005 attribute specification:

```
(* UPF_clamp_value = "1" *) output my_Logic_Port;
```

VHDL attribute specification:

```
attribute UPF_clamp_value : STD.Standard.String;

attribute UPF_clamp_value of my_Logic_Port : signal is "1";
```

The same attributes can be specified in UPF, using the **set_port_attributes** command and its generic **-attribute** option, or it can also be specified in UPF, using the **set_port_attributes** command and its specific option **-clamp_value** (see 6.47).

pg_type port properties can be annotated in HDL using the following attributes:

Attribute name: **UPF_pg_type**

Attribute value: <**primary_power** | **primary_ground** | **backup_power** | **backup_ground** >

SystemVerilog or Verilog-2005 attribute specification:

```
(* UPF_pg_type = "primary_power" *) output myVddPort;
```

VHDL attribute specification:

```
attribute UPF_pg_type : STD.Standard.String;

attribute UPF_pg_type of myVddPort : signal is "primary_power";
```

The same attributes can be specified in UPF, using the **set_port_attributes** command and its generic **-attribute** option, or it can also be specified in UPF using the **set_port_attributes** command and its specific option **-pg_type** (see 6.47).

The UPF leaf cell treatment of a model or instance can be annotated in HDL using the following attributes:

Attribute name: **UPF_is_hard_macro**

Attribute value: **<TRUE | FALSE>**

SystemVerilog or Verilog-2005 attribute specification:

```
(* UPF_is_hard_macro="TRUE" *) module FIFO (<port list>);
```

VHDL attribute specification:

```
attribute UPF_is_hard_macro : STD.Standard.String;

attribute UPF_is_hard_macro of FIFO : entity is "TRUE";
```

The same attribute can be specified in UPF, using the **set_design_attributes** command (see 6.40).

When any register (specified or implied) with the **UPF_retention** attribute value set to **required** is included in a power domain that has at least one retention strategy, the register shall be included in a retention strategy defined for the domain.

Elements requiring retention can be attributed in HDL as follows:

Attribute name: **UPF_retention**

Attribute value: <**required** | **optional**>

SystemVerilog or Verilog-2005 attribute specification:

```
(* UPF_retention = "required" *) module my_flip;
```

VHDL attribute specification:

```
attribute UPF_retention : STD.Standard.String;

attribute UPF_retention of my_flip : variable is "required";
```

The same attribute can be specified in UPF, using the **set_design_attributes** command (see 6.40).

```
set_design_attributes -elements {my_flip} \
         -attribute {UPF_retention required}
```

5.7 Precedence

To support concise, easily written low-power specifications, UPF commands can range from very specific to very generic in their scope of application. This enables specification of generic defaults that apply widely except where more specific commands provide more focused information. This subclause describes the precedence relations that determine which of several commands that potentially apply in a given situation shall actually apply.

A **create_power_domain** command (see 6.20) that explicitly includes a given instance in its extent shall take precedence over one that applies to an instance transitively (i.e., applies to an ancestor of the instance, and therefore to all of its descendants). A **create_power_domain** command that creates an atomic power domain takes precedence over one that creates a non-atomic power domain.

A **set_retention** command (see 6.49) that explicitly includes a given instance in its *element_list* shall take precedence over one that applies to an instance transitively (i.e., applies to an ancestor of the instance, and therefore to all of its descendants), which takes precedence over one that applies to an entire domain.

If multiple **set_isolation** commands (see 6.44), or multiple **set_level_shifter** commands (see 6.45), or multiple **set_repeater** commands (see 6.48) potentially apply to the same port, the following criteria (listed in order from highest precedence to lowest precedence) determine the relative precedence of the commands, and only the command(s) with the highest precedence shall actually apply:

a) Command that applies to part of a multi-bit port specified explicitly by name

b) Command that applies to a whole port specified explicitly by name

c) Command that applies to all ports of an instance specified explicitly by name

d) Command that applies to a port of a specified power domain with a given sink and source

e) Command that applies to a port of a specified power domain with a given sink or source

f) Command that applies to all ports of a specified power domain with a given direction

g) Command that applies to all ports of a specified power domain

If multiple strategies of the same type have the same highest precedence, then all of those commands actually apply to the port or part thereof, to the extent allowed by the strategy.

A prefix or suffix to be used to create names for inserted isolation, level-shifter, and repeater cells that is specified by the **-name_prefix** or **-name_suffix** options, respectively, of **set_isolation**, **set_level_shifter**, and **set_repeater**, takes precedence over any user-defined prefix or suffix for these commands specified by the **name_format** command (see 6.37). A prefix or suffix explicitly specified using the **name_format** command in turn takes precedence over the default prefix or suffix specified in the definition of the **name_format** command.

If multiple supply connections potentially apply to the same port, the actual application is determined by the following precedence order, from highest to lowest precedence:

h) Command that explicitly connects to part of a port

i) Command that explicitly connects to a whole port
(e.g., `connect_supply_net -ports`)

j) Command that automatically connects to ports of an instance
(e.g., `connect_supply_set -connect -elements`)

k) Command that automatically connects to ports of any instance in a given region
(e.g., `connect_supply_set -connect` to connect a handle associated with a domain or `connect_supply_net -pg_type -cells -domain`)

l) Command that automatically connects to ports of any instance
(e.g., `connect_supply_net -pg_type -cells`)

Any explicit connection command takes precedence over implicit connections made by default.

If multiple set_port_attributes commands potentially specify the same overridable attribute of a given port, whether specified explicitly in UPF or implied by HDL or Liberty attribute specifications, only the command(s) with the highest precedence will actually apply. The following criteria (listed in order from highest precedence to lowest precedence) determine the relative precedence of the commands.

The command references:

m) A part of the given port, specified explicitly by name in the **-ports** list (without **-model)**

n) The whole given port, specified explicitly by name in the **-ports** list (without **-model)**

o) The given port, implied by specifying an instance name in the **-elements** list with a given direction

p) The given port, implied by specifying an instance name in the **-elements** list

q) A part of the given port of the named module or library cell, specified explicitly by name in the **-ports** list (with **-model)**

r) The whole given port of the named module or library cell, specified explicitly by name in the **-ports** list (with **-model)**

s) The given port of the instance corresponding to the current scope if none of the options **-ports**, -**elements**, **-model** are present

If a given user-defined attribute is defined on both (a port of) a model and (a port of) an instance of that model, the instance attribute definition takes precedence over the model attribute definition.

It shall be an error if the precedence rules fail to uniquely identify the value of the UPF attribute that applies to a port. In other words, it shall be an error if two UPF attribute specifications with the same highest precedence specify different values for the same attribute of the same port.

It shall be an error if a non-overridable attribute is specified with two different values for the same object, regardless of the precedence rules for attribute specifications.

For simstates that apply to a given object at any given time, a more conservative (i.e., more corrupting) simstate takes precedence over a less conservative (less corrupting) simstate.

The following also apply:

— The precedence of a command is independent of the current scope during the command processing.

— It shall be an error if the precedence rules fail to uniquely identify the power intent that applies to an object.

— The **find_objects** command (see 6.30) returns a list of explicit names; these names can refer to whole objects or to elements thereof. When *list* arguments to command options are created using **find_objects**, the level of precedence is based on the expanded value used as the argument, not as the pattern or regular expression used in **find_objects**.

— The symbol **.** in **-elements {.}** is an explicit reference to the instance corresponding to the current scope.

5.8 Generic UPF command semantics

All **map_*** commands specify the elements to be used in implementation. These specifications override the elements that may be inferred through a strategy. The behavior of this manual mapping may lead to an implementation that is different from the RTL specification. Therefore, it may not be possible for logical equivalence checking tools to verify the equivalence of the mapped element to its RTL specification.

5.9 effective_element_list semantics

The *effective_element_list* is the set of elements to which a command applies. The *effective_element_list* is constructed from the arguments provided to the command. The terms used in the description of this construction include: *element_list, exclude_list, aggregate_element_list, aggregate_exclude_list, prefilter_element_list,* and *effective_element_list.* The *element_list* and *exclude_list* are lists that contain the elements specified by an instance of the command. The *effective_element_list*, *aggregate_element_list*, and *aggregate_exclude_list* are associated with the named object of the command.

The following arguments can determine the *effective_element_list*:

a) **-elements** *element_list* adds the rooted names in *element_list* to the *aggregate_element_list*. It is not an error for an element to appear more than once in this list.

b) **-model** *model_name* adds the rooted name of each instance that is an instance of the model to the *aggregate_element_list*.

c) **-models** *model_list* or **-model** *model_name* adds the rooted name of each instance that is an instance of the *model name* or any of the models in *model_list* to the *aggregate_element_list*. It is not an error for a model to appear more than once in this list.

d) **-lib** *lib_name* selects all models from the specified *lib_name*. If only **-lib** *lib_name* is specified, the rooted name of each instance that is an instance of every model present in *lib_name* is added to the *aggregate_element_list*.

e) If **-lib** *lib_name* is specified along with **-model** *model_name* or **-models** *model_list*, the model is selected only if it is present in *lib_name*. This results in rooted names for only those models that are present in the *lib_name* library.

f) If **-lib** *lib_name*, **-model**, or **-models** is specified with an **-elements** option, the *aggregate_element_list* is constructed by adding the rooted names from **-elements** and rooted names resulting from any **-lib**/**-model**/**-models** options.

g) **-exclude_elements** *exclude_list* adds the rooted names in *exclude_list* to the *aggregate_exclude_list*. It is not an error for an element to appear more than once in this list. It is not an error for an element in the exclude list to not be in the *aggregate_element_list*.

h) When **-elements** *element_list* is specified with a period (`.`), the current scope is included as a rooted instance in the *aggregate_element_list*.

i) It shall be an error if the *element_list* is not specified as one of **{}**, **{.}**, or **{***list***}**.

j) When **-transitive** is specified with the (default or explicit) value **TRUE**, elements (see 5.9.1) in *aggregate_element_list* that are not leaf cells are processed to include the child elements (see 5.9.2).

k) The *prefilter_element_list* comprises the *aggregate_element_list* with any matching elements from the *aggregate_exclude_list* removed (see 5.9.2).

l) The command arguments identified as filters are predicates that shall be satisfied by elements in the *effective_element_list*. The *prefilter_element_list* is filtered by the predicates to produce the *effective_element_list* (see 5.9.2).

m) The range of legal element types is command dependent for each command that uses **-elements**. Each command specifies the effect of an empty *aggregate_element_list*. An explicitly empty list may be specified with **{}**.

5.9.1 Transitive TRUE

The detailed semantics of **-transitive TRUE** are described using Figure 3, Figure 4, and Figure 5. The figures are exemplary; the text provides a semantic for the validation of the result.

Given a design as shown in Figure 3 with a instance `A` in the current scope, where `A` has child elements `B`, `C`, and `D`; `B` has child elements `E` and `F`; `C` has child elements `G` and `H`; and `D` has child elements `I` and `J`.

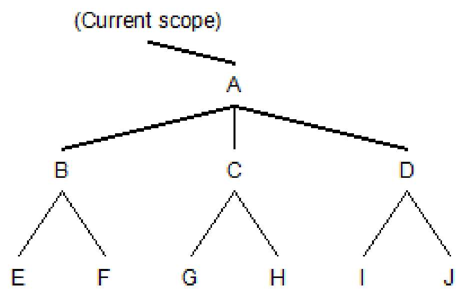


Figure 3—Element processing example design fragment

If the specification:

```
-elements {A A/C/H} -exclude_elements {A/C A/D} -transitive TRUE
```

is applied to the design fragment shown in Figure 3, then Figure 4 shows the four specified elements by indicating them as boxed; those specified with exclude are shown with strike-through text.

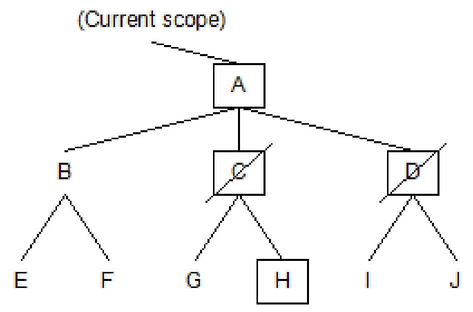


Figure 4—Element processing specification

Figure 5 shows the results of the *effective_element_list*. The list includes `{A A/B A/B/E A/B/F A/C/H}`

The elements included or excluded by transitivity are shown as dashed boxes or with strike-through text, respectively.

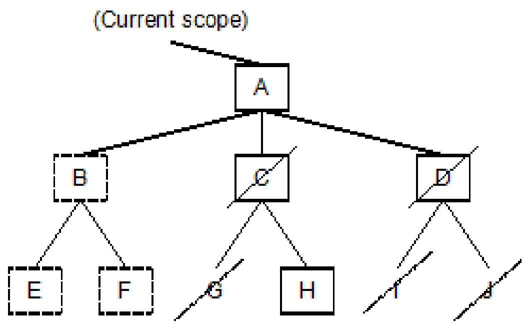


Figure 5—Element processing result

5.9.2 Result

The required result is derived as follows:

```
Begin // at the current scope.
   Initialize by traversing the hierarchy and set element.mark := exclude
   For each element in the aggregate_element_list do
      set element.mark := includeP
      if (transitive = TRUE AND element NOT Leaf_Cell) then
         foreach child in element call mark_child(child, include)
      end if
   done
   For each element in the aggregate_exclude_list do
      set element.mark := excludeP
      if (transitive = TRUE AND element NOT Leaf_Cell) then
         foreach child in element call mark_child(child, exclude)
      end if
   done
   For each element in the aggregate_element_list call
   check_and_add(element)
done

proc mark_child(element, value)
   if (element.mark != excludeP AND element.mark != includeP ) then
      element.mark := value
      if (element NOT Leaf_Cell) then
         foreach child in element call mark_child(child, value)
      end if
   end if
end proc

proc check_and_add(element)
   if (element.mark = includeP OR element.mark = include) then
      if (for all filters filter(element) = TRUE) then
         add element to effective_element_list
         if (transitive = TRUE AND element NOT Leaf_Cell) then
            foreach child in element call check_and_add(child)
         end if
      end if
   end if
end proc
```

NOTE—Implementations may use any data structure or algorithm that produces the same results as the preceding method.

5.10 Command refinement

Some UPF commands support incremental refinement. Commands that support incremental refinement are called *refinable commands*. A refinable command may be invoked multiple times on the same object and each invocation may add additional arguments to those specified in previous invocations. The arguments of a refinable command that may be added after the first invocation are called *refining arguments*; these are shown in **boldface-green text** and labeled with an **R** in their respective *arguments* listings. Certain commands have refinable arguments; such arguments may have additional information about that argument added after the first invocation of the command, in much the same way that refinable commands may have additional arguments added later.

The first instance of a refinable command identifies the object to which it applies; all mandatory arguments shall be declared in this call and any other arguments may also be included. Subsequent occurrences of the command that identify the same object shall be executed in the same scope and shall include the **-update** option and refining arguments as required. The mandatory arguments that identify the object to which the command applies (the object name following the command or option name, and for strategies, the domain specification as well) shall also be included in each subsequent occurrence, but other mandatory arguments are not required in subsequent occurrences of the command. The end result shall be as if all of the arguments, other than the **-update** argument, had been included in the initial occurrence of the command, either individually (e.g., **-clamp_value** or **-isolation_supply**) or merged together into a single argument (e.g., **-elements** or **-exclude_elements**).

For example, the **set_isolation** command (see 6.44) can be invoked for the first time in a given scope to define a strategy name for a particular domain. Subsequent **set_isolation** commands executed in the same scope can specify the same strategy and domain names and also specify additional arguments to further characterize the isolation strategy defined by the previous command. Similarly, the **add_power_state** command (see 6.5) can be invoked initially in a given scope to define a set of power states for a supply set. A subsequent invocation of **add_power_state** in the same scope and for the same supply set may use the **-update** option to add a **-simstate** specification to each power state definition.

When **-update** is used for command refinement, the following apply:

— It shall be an error if **-update** is specified on the first command of a given kind that applies to a given object.

— It shall be an error if **-update** is not specified on subsequent commands of the same kind that apply to the same object.

— Except for those command arguments that aggregate (see 5.9 and 6.5), it shall be an error if subsequent commands specify a value for a given argument that conflicts with or contradicts a previously specified value for the same argument.

Example

This shows a multiple-part refinement for a usage of **set_isolation** (see 6.44).

a) Constraint specification using port attributes
```
set_port_attributes
    -elements {a b c d}
    -clamp_value 0
```
b) Logical configuration
```
set_isolation demo_strategy -domain pda
    -elements {a b c d}
    -clamp_value 0
    -isolation_signal {iso_en}
    -isolation_sense {LOW}
```
c) Adding elements to the strategy
```
set_isolation demo_strategy -domain pda -update
    -elements {e f g}
```
d) Supply set implementation
```
set_isolation demo_strategy -domain pda -update
    -isolation_supply pda_isolation_supply
```

The implementation-independent part of the power intent, shown in a) above, could also be declared in the SystemVerilog HDL using the following attributes:

```
(* UPF_clamp_value = "0" *) out a;
```

```
    (* UPF_clamp_value = "0" *) out b;
    (* UPF_clamp_value = "0" *) out c;
    (* UPF_clamp_value = "0" *) out d;
```

In this case, the declaration shall have identical semantics to the equivalent UPF command.

5.11 Error handling

If an error condition occurs, e.g., an incorrect command-line option is specified, then a `TCL_ERROR` exception shall be raised. This exception can be caught using the Tcl `catch` command, so these errors can be prevented from aborting the active **load_upf** command (see 6.32). These errors shall have no impact on further commands. Processing may continue after the error is caught. Sequencing of the error `catch` and the choice of continuation is tool-dependent. The state of the design after an error is not defined. Specifically, a command that raises an error may partially complete before aborting.

In general, all commands that fail shall raise a `TCL_ERROR`. As described in the Tcl documentation, the global variables accessible after an error occurs include *errorCode* and *errorInfo*.

5.12 Units

Voltage values are expressed as real number literals that represent voltage measurements with the implicit unit of 1 V. For example, the literal 1.3 represents 1.3 V, or equivalently 1300 mV, or 1 300 000 µV.

5.13 SystemC language basic

IEEE Std 1801-2015 support for SystemC is limited to power analysis in system-level design use models.

6. Power intent commands

6.1 Introduction

Clause 6 documents the syntax for each UPF command. For details concerning the simstate semantics, see Clause 9.

6.2 Categories

Each command in Clause 6 is categorized based on the following definitions. Unless otherwise mentioned, all *constructs* (commands and/or options) in this standard are considered *current*. Constructs considered as *legacy* or *deprecated* shall be explicitly denoted.

a) *Current*—A construct defined in the standard with the following characteristics:
   1) It is recommended for use.
   2) Its semantics fully support the latest concepts.
   3) Its interaction with other related constructs is well defined.
   4) It is expected to be part of the standard and be considered for extension in future versions.

b) *Legacy*—A construct defined in the standard with the following characteristics:

1) It is *not recommended* for use for new code.

2) Its semantics are not interoperable with all of the latest UPF concepts.

3) It will not be considered for extensions in future versions.

4) It is included for backward compatibility only, e.g., **set_isolation -isolation_power_net** (see 6.44).

Legacy constructs (commands and/or options) have not had their syntax and/or semantics updated to be consistent with other commands in this version of the standard, so their descriptions may contain significant obsolete information and their semantics may not be interoperable with the latest UPF concepts.

c) *Deprecated*—A construct defined in the standard with the following characteristics:

1) It is *not recommended* for use for any code.

2) It will not be considered for extensions in future versions.

3) It may be deleted from future versions, e.g., **describe_state_transition** (see 6.28).

Deprecated commands are noted in this standard without syntax definitions or semantic explanations. Deprecated options of current commands are noted in the syntax definition of those commands, but are not mentioned in the semantic explanations of those commands.

For recommendations on how to use current constructs to replace legacy and deprecated ones, see Annex D.

6.3 add_parameter

Purpose	Define parameters for use within the system-level IP power model.	
Syntax	**add_parameter** *parameter_name* **-type** < **buildtime** | **runtime** | **rate** > **-default** value **-description** *string*	
Arguments	*parameter_name*	The name of the parameter.
	-type <**buildtime** | **runtime** | **rate**>	The type of parameter being defined. The default is **buildtime**.
	-default value	Specify the default value for the parameter in floating point form.
	[-description]	A description of the parameter represented as a string.
Return value	Return a 1 if successful or raise a `TCL_ERROR` if not.	

The **add_parameter** command is used to define parameters for use within a system-level IP power model. The parameter scope is within the power model only and power models and power functions cannot access parameters that are defined outside of the power model in which they are used. Three types of parameters can be defined as follows:

— Build time—used to define parameters that remain unchanged during run time

— Run time—used to define parameters that can change during run time

— Rate—used to define parameters that represent rate-based quantities that can change during run time

Both run time and rate-based parameters can form a part of the sensitivity list for a power function. Any change in the value of such a parameter forces an invocation of the power function and a recalculation of power (or current) consumption. The units in which the parameter is defined are included within the parameter definition. Standard SI units shall be used where required, for parameters that are defined within a system-level IP power model.

It shall be an error if:

a) *parameter_name* has already been defined within the power model

b) A default value of the parameter is not provided

c) A parameter defined using **-type buildtime** changes value during simulation

Syntax example

```
add_parameter process -type buildtime -default 1.0 -description "Process
    Scaling Factor"
add_parameter CPUVoltage -type runtime -default 900mv -description "CPU
    Supply Voltage"
add_parameter CacheMiss -type rate -default 0.02 -description "Cache Miss
    Rate"
```

6.4 add_port_state (legacy)

Purpose	Add states to a port.	
Syntax	**add_port_state** *port_name* {**-state** {*name* <*nom* | **off**>}}*	
Arguments	*port_name*	The name of the supply port. Hierarchical names are allowed.
	-state {*name* <*nom* | **off**>}	The *name* and value for a state of the supply port. The value can be a nominal voltage or **off**.
Return value	Return the fully qualified name (from the current scope) of the created port or raise a `TCL_ERROR` if any of the port states are not added.	

This is a legacy command; see also 6.2 and Annex D.

The **add_port_state** command adds state information to a supply port. If the voltage values are specified, the supply net state is **FULL_ON** and the voltage value is the single nominal value or within the range of min to max; otherwise, if **off** is specified, the supply net state is **OFF**.

The add_port_state command defines a named supply state for a supply port. If a voltage is specified, the supply net state is FULL_ON and the voltage value is the specified value; otherwise if off is specified, the supply net state is OFF.

It shall be an error if *port_name* does not already exist.

NOTE—The **add_supply_state** command (see 6.8) is a generalization of **add_port_state**; **add_supply_state** can be used to define named supply states for supply ports, supply nets, and supply set functions.

Syntax example

```
add_port_state VN1
    -state {active_state 0.90}
    -state {off_state off}
```

6.5 add_power_state

6.5.1 Overview

Purpose	Define power state(s) of an object.		
Syntax	**add_power_state** [**-supply** | **-domain** | **-group** | **-model** | **-instance**] *object_name* [**-update**] [**-state** {*state_name* [**-logic_expr** {*boolean_expression*}] [**-supply_expr** {*boolean_expression*}] [**-power_expr** {*power_expression*}] [**-simstate** *simstate*] [**-legal** | **-illegal**] }]* [**-complete**]		
Arguments	*object_name*	Simple name of an object.	
	-supply | **-domain** | **-group** | **-model** | **-instance**	These arguments specify the kind of object to which this command applies.	
	-state {*state_name* ...}	*state_name* is the simple name of the state being defined or refined.	
	-supply_expr {*boolean_expression*}	**-supply_expr** specifies a Boolean expression defined in terms of supply ports, supply nets, and/or supply set handle functions that evaluates to *True* when the object is in the state being defined.	**R**
	-logic_expr {*boolean_expression*}	**-logic_expr** specifies a Boolean expression defined in terms of logic nets and/or power states of supply sets and/or power domains that evaluates to *True* when the object is in the state being defined.	**R**
	-simstate *simstate*	**-simstate** specifies a **simstate** for the power states associated with a supply set. Valid values are **NORMAL**, **CORRUPT_ON_CHANGE**, **CORRUPT_STATE_ON_CHANGE**, **CORRUPT_STATE_ON_ACTIVITY**, **CORRUPT_ON_ACTIVITY**, **CORRUPT**, and **NOT_NORMAL**. See 4.5.4.7.	**R**
	-power_expr {*power_expression*}	Specifies the power consumption of this object in this power state, or a function for computing the power consumption.	
	-legal | **-illegal**	These options specify the legality of the state being defined as either legal or illegal. The default is **-legal**.	**R**
	-complete	Specifies that all fundamental power states to be defined for this object have been defined. This implies that all legal power states have been defined and any state of the object that does not match a defined state is an illegal state.	**R**
	-update	Indicates this command provides additional information for a previous command with the same *object_name* and executed in the same scope.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

Semantics

add_power_state defines one or more power states of an object. Power states may be defined for a supply set, a power domain, a composite domain, a group, a model, or an instance. Power states of a model are inherited by any instance of that model.

If **-supply** is specified, the *object_name* shall be the name of a supply set or a supply set handle. If **-domain** is specified, the *object_name* shall be the name of a power domain. If **-group** is specified, the *object_name* shall be the name of a group. If **-model** is specified, the *object_name* shall be the name of a model. If **-instance** is specified, the *object_name* shall be the name of an instance. If none of the above are specified, the type of *object_name* determines the kind of object to which the command applies.

The state name and the logic expression of a power state definition determine whether the power state is a deferred power state, a definite power state, or an indefinite power state (see 4.6.3).

The state_name in a power state definition shall be either a simple name or a hierarchical state name. A hierarchical state name is a name of the form *abstract_state_name.simple_name*. In the latter case, the hierarchical state name being defined is a refinement of the power state previously defined with the name *abstract_state_name*.

The defining expression for a power state with the simple name S of an object O is the logic expression given in the power state definition. The defining expression for a power state with the hierarchical state name A.S of an object O is the conjunction of the term O==A and the logic expression in the definition of state A.S.

A hierarchical state name shall be used only to define a definite state. The *abstract_state_name* shall be the name of another definite power state of the same object. A hierarchical state name allows for definition of a more refined power state without explicitly specifying the more abstract power state in the logic expression.

Example

```
    add_power_state -supply PD.primary \
      -state {ON.TURBO}    ;# a refinement of the predefined ON state
```

which is functionally equivalent to

```
    add_power_state -supply PD.primary \
      -state {TURBO -logic_expr {PD.primary==ON} }
```

NOTE 1—A hierarchical state name is the whole name of a power state; the entire state name must be used in any reference to that power state.

NOTE 2—Two different hierarchical state names for the same object may have the same suffix as long as the whole names are different. For example, ON.ECO.P1 and ON.TURBO.P1 can both be defined as refinements of power state ON of a given object.

The power states defined for a given object include only predefined power states for that class of object (see 4.7.4) and those defined explicitly for that object or an instance of that object. Power states defined for one object are not inherited implicitly by any related object (e.g., by a supply set handle with which a supply set has been associated or vice versa) except that power states of a model are inherited by each instance of the model. However, power states of one object can be defined in terms of power states of another object, to represent dependencies or correlation of power states.

The set of power states for a given object may be specified incrementally by using **-update**. The first **add_power_state** for that object may define one or more power states. Subsequent **add_power_state -update** commands for the same object may define additional power states.

A power state definition itself may also be specified incrementally by using **-update**. The initial definition of the power state defines at least the power state name and may specify additional information about this power state. Subsequent **add_power_state -update** commands for the same power state of the same object may specify additional details about that power state.

A power state definition may be specified as either a legal power state (**-legal**) or an illegal power state (**-illegal**). By default, a power state definition defines a legal power state. A legal power state initially defined either with or without **-legal** may be updated with **-illega**l to indicate that it is an illegal power state in a given context. In particular, a legal power state of an instance that was inherited from the corresponding model may be updated to indicate that this power state is not legal for the specific instance.

6.5.2 Logic expression

The **-logic_expr** *boolean_expression* shall be a Boolean expression (see 5.4) referencing control signals, clock signal intervals, and/or power states of an object. For convenience, the following expression forms may appear in this expression:

a) ***interval(****signal_name edge1 edge2****)***

   Equivalent to
   the time between the most recent two specified edges of *signal_name*
   (returns the largest supported time value until both edges have occurred)

   where
   edge1, *edge2* shall be one of **posedge** or **negedge**.

b) ***interval(****signal_name edge****)***

   Equivalent to
   interval(*signal_name edge edge****)***

c) ***interval(****signal_name****)***

   Equivalent to
   interval(*signal_name* **posedge posedge*****)***

d) *object* **==** *power_state*

   Evaluates to True if *power_state* of *object* is active

   where
   object is the name of a supply set, power domain, composite domain, group, model, or instance.

e) object != power_state

   Evaluates to True if *power_state* of *object* is not active

   where
   object is the name of a supply set, power domain, composite domain, group, model, or instance.

Examples

```
-logic_expr { enable == 1'b1 && interval(clk) < 5ps }
-logic_expr { core_pd.primary == ON_1d2v }
-logic_expr { core_pd == turbo && ram_pd != sleep }
```

Within a logic expression specified as part of a power state definition for a given power domain, the supply set handles of that power domain may be referenced directly without prefixing the name with the power domain name. To refer to an object declared in the current scope with the same name as a supply set handle of the power domain, the object name shall be prefixed with `./`.

A logical contradiction exists when a logic net or supply set or power domain is specified to be more than one value in the definition of a given power state, e.g., (enable == '1') and (enable == '0'). A power state definition is erroneous if it contains logical contradiction(s).

6.5.3 Simstate

A power state definition for a supply set may specify a simstate (see 4.8). The simstate of a power state of a supply set that is the primary supply of a given power domain determines the simulation behavior of elements in that domain when that power state is the current power state of the domain's primary supply set.

6.5.4 Supply expression

A power state definition for a supply set may also specify a supply expression. The supply expression specifies the supply states of the supplies that cause this power state to be active. The supply expression can also specify voltage values for each supply.

The **-supply_expr** *boolean_expression* shall be a Boolean expression (see 5.4) that may reference available supply nets, supply ports, and/or functions of supply sets or supply set handles. For convenience, the following expression forms may appear in this expression:

a) *supply_net* **==** *net_state*

   Equivalent to
   { supply_net.state == net_state }

   where
   supply_net is the name of a supply port or net or a supply set (handle) function
   net_state is the name of a state associated with *supply_net*.

b) *supply_net* **==** *{ net_state nom_voltage }*

   Equivalent to
   { supply_net.state == net_state && supply_net.voltage == nom_voltage }

   where
   supply_net is the name of a supply port or net or a supply set (handle) function
   net_state is the name of a state associated with *supply_net*

The first expression form may be used to specify the supply set of a supply set function, supply port, or supply net without indicating the voltage. The second expression form may be used to specify both supply state and voltage. A supply set power state defined with a supply expression involving the first expression form may be updated later with a supply expression involving the second expression form, provided that the second expression form specifies the same supply state as in the original definition.

Within a supply expression specified as part of a power state definition for a given supply set or supply set handle, the functions of that supply set or supply set handle may be referenced directly without prefixing the name with the power domain name. To refer to an object declared in the current scope with the same name as a function of the supply set or supply set handle, the object name shall be prefixed with `./`.

6.5.5 Power expression

A power state definition for a module or power domain may specify a power expression. The power expression is used to define the power (or current) consumption of the object (power domain or component) when this power state is the current power state.

The power expression can take one of two forms:

— A list of two literal values for power (or current) including the associated SI units. The first value in the list represents static power (or current); the second represents dynamic power (or current).

— Identification of a power function together with a list of parameters to which that power function is sensitive. Evaluation of the function returns a list of static and dynamic power (or current) values in the above format.

The power function calculates power (or current) consumption for the power state for which it is defined. The power function is evaluated on entry to the power state—i.e., when the power state for which it is defined becomes the current power state. It is re-evaluated at any time while this power state is the current power state and a parameter in the sensitivity list of the power function changes value. If the power function is defined without a sensitivity list, then power (or current) consumption shall only be calculated on entry to the power state.

Power expressions are used for component power modeling, which involves defining a power model using the **begin_power_model** and **end_power_model** commands and applying the power model to an instance using the **apply_power_model** command. The **add_parameter** command can be used in a power model to define the parameters of a power expression.

A power expression shall be specified only for a deferred power state (see 4.7.3).

It shall be an error if

— a power expression appears in a power state definition that is not contained within a power model

— a parameter specified in the power expression has not been defined within the power model

Syntax example

```
add_power_state -model CPU
    -state {ACTIVE
        -power_expr {UPF::power_functions::cpu
                    {voltage frequency temperature IPC L1DAccess }}
        }
        -state {DORMANT
            -power_expr {0.01mW 0.0mW}
        }
```

6.5.6 Power state definition restrictions

In addition to above-mentioned restrictions, the following apply:

a) If a supply expression is used to define a power state of a given supply set or supply set handle, it shall only refer to supply ports, supply nets, and/or functions of the given supply set or supply set handle. It shall be an error if such a supply expression refers to functions of another supply set or supply set handle. It is also an error if the supply expression does not refer to at least one of the power function or ground function of the supply set.

b) If a logic expression is used to define a power state of a given supply set or supply set handle, it shall only refer to logic ports, logic nets, interval functions, and/or power states of the given supply set or supply set handle. It shall be an error if such a logic expression refers to functions of a supply set or supply set handle, power states of another supply set or supply set handle, or power states of a domain.

c) If a logic expression is used to define a power state of a given power domain, it shall only refer to logic ports, logic nets, interval functions, power states of supply sets or supply set handles that are available in the domain, and/or power states of power domains. It shall be an error if such a logic expression refers to supply ports, supply nets, or functions of a supply set or supply set handle. It is also an error if the logic expression does not refer to the power states of all supply sets of the domain that have more than one legal power state.

d) If a logic expression is used to define a power state of a composite power domain, it shall only refer to logic ports, logic nets, interval functions, power states of available supply sets or supply set handles, and/or power states of its subdomains. It shall be an error if such a logic expression refers to supply ports, supply nets, or functions of a supply set or supply set handle, or power states of domains that are not subdomains. It is also an error if the logic expression does not refer to the power states of all subdomains of the composite domain that have more than one legal power state.

e) It shall be an error if a supply expression is used to define a power state of a power domain, composite domain, group, module, or instance.

f) It shall be an error if a simstate is associated with a power state of a power domain, composite domain, group, module, or instance.

g) When **-simstate**

    1) Is first specified for a named state, any of the arguments may appear.

    2) Is specified as **NOT_NORMAL**, the effect shall be the same as if **CORRUPT** had been specified (see 4.7.4), except that the definition may be subsequently refined to any simstate other than **NORMAL**.

    3) Has previously been specified as **NORMAL**, **CORRUPT**, **CORRUPT_ON_ACTIVITY**, **CORRUPT_ON_CHANGE**, **CORRUPT_STATE_ON_CHANGE**, or **CORRUPT_STATE_ON_ACTIVITY**, it shall be an error if an **add_power_state -update** command for the same object specifies any simstate other than that originally specified (e.g., once **CORRUPT** has been specified for a particular state, it shall remain as **CORRUPT** in any subsequent updates for the definition of that state).

h) The simstate for predefined power state **ON** is **NORMAL**.

i) The simstate for predefined power states **OFF** and **ERROR** is **CORRUPT**.

j) The predefined power state **UNDEFINED** is defined with no simstate.

k) There is no default simstate for a user-defined power state.

l) The supply set is in the **OFF** power state when it is not in one of the defined power states of the supply set that have simstates defined on them, including the **ON** predefined state.

m) If **-illegal** has been specified in the definition of a power state for a given object, it shall be an error if that object is in a state that matches the definition of that power state. A verification tool shall emit an error message when an object is in an illegal power state.

n) If **-complete** has been specified in an **add_power_state** command for a given object, it shall be an error if that object is in a state that does not match any of the defined power states. A verification tool shall emit an error message when an object is in such an undefined state.

o) If **-complete** has been specified on an **add_power_state** command for a given object, it shall be an error if a subsequent update to that command defines a new fundamental power state. It is not an error if a subsequent update to that command refines a previously defined power state, or defines a new power state that is a refinement of a previously defined power state.

p) It shall be an error if a logic expression used to define a given power state contains a direct or indirect reference to that same state.

NOTE 1—The choice of state name has no simstate implications.

NOTE 2—Implementation tools may optimize a design based on the presumption illegal states never occur. Such optimizations are allowed only if they do not change the behavior of the design.

NOTE 3—If the **add_power_state** command for the primary supplies of two interconnected domains are both defined as complete, this implies that all intended legal fundamental power states have been defined for each domain, and, therefore, all possible state combinations of the two domains have been defined.

Syntax examples

```
add_power_state PdA.primary -supply
  -state {GO_MODE
    -logic_expr {DM_ON}
    -simstate NORMAL
    -supply_expr {(power == {FULL_ON 0.8})
                   && (ground == {FULL_ON 0})
                   && (nwell == {FULL_ON 0.8})
     }
  -state {OFF_MODE
    -logic_expr {!DM_ON}
    -simstate CORRUPT
    -supply_expr {power == {OFF}}
     }
  -state {SLEEP_MODE
    -logic_expr {DM_ON && (interval(clk_dyn posedge negedge) >= 100ns)}
    -simstate CORRUPT_STATE_ON_CHANGE
    -supply_expr {(power == {FULL_ON 0.8})
                    && (ground == {FULL_ON 0})
                    && (nwell == {FULL_ON 1.0})}
      }
add_power_state PdA.primary -supply -update -complete
add_power_state PdTOP -domain
  -state {GOGO -logic_expr {u1/PdA.primary == GO_MODE}}
add_power_state PdTOP -domain -update
  -state {GOGO -illegal}
```

6.6 add_pst_state (legacy)

Purpose	Define the states of each of the supply nets for one possible state of the design.	
Syntax	**add_pst_state** *state_name* **-pst** *table_name* **-state** *supply_states*	
Arguments	*state_name*	The simple name of the state being defined.
	-pst *pst_name*	The power state table (PST) to which this state applies.
	-state *supply_states*	The list of supply net state names (see 6.24), listed in the corresponding order of the **-supplies** listing in the **create_pst** command (see 6.23). A * in place of a state name indicates this is a "don't care" for that supply.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

This is a legacy command; see also 6.2 and Annex D.

The **add_pst_state** command defines the name for a specific state of the supply nets defined for the PST *table_name*.

It shall be an error if:

— The number of *supply_states* is different from the number of supply nets within the PST.

— A *state_name* is defined more than once for the same PST.

— Any *supply_state* name is ambiguous (i.e., is defined for more than one of the supplies from which the value of the corresponding object in the **-supplies** list of **create_pst** is derived).

Syntax example

```
create_pst            pt -supplies { PN1   PN2   SOC/OTC/PN3 }
add_pst_state s1 -pst pt -state    { s08   s08   s08         }
add_pst_state s2 -pst pt -state    { s08   s08   off         }
add_pst_state s3 -pst pt -state    { s08   s09   off         }
```

6.7 add_state_transition

Purpose	Define named transitions among power states of an object.		
Syntax	**add_state_transition** [**-supply** | **-domain** | **-group** | **-model** | **-instance**] *object_name* [**-update**] [**-transition** {*transition_name* [**-from** *from_list* **-to** *to_list*] [**-paired** {{*from_state to_state*}*}] [**-legal** | **-illegal**] }]* [**-complete**]		
Arguments	*object_name*	The rooted name of the object for which state transitions will be defined.	
	-supply | **-domain** | **-group** | **-model** | **-instance**	These arguments specify the kind of object to which this command applies.	
	-update	Indicates this command provides additional information for a previous command with the same *object_name* and executed in the same scope.	**R**
	-transition *transition_name*	Simple name of a transition.	
	-from *from_list* **-to** *to_list*	*from_list* is an unordered list of power state names active before a state transition. *to_list* is an unordered list of power state names active after a state transition.	**R**
	-paired {{*from_state to_state*}*}	A list of *from_state* name and *to_state* name pairs.	**R**
	-legal | **-illegal**	These options specify the legality of the transition being defined as either legal or illegal. The default is **-legal**.	**R**
	-complete	Specifies that all state transitions to be defined for this object have been defined.	**R**
Return value	Return an empty string if successful or raise a TCL_ERROR if not.		

add_state_transition defines named state transitions between power states of an object.

If **-supply** is specified, the *object_name* shall be the name of a supply set or a supply set handle. If **-domain** is specified, the *object_name* shall be the name of a power domain. If **-group** is specified, the *object_name* shall be the name of a group. If **-model** is specified, the *object_name* shall be the name of a model. If **-instance** is specified, the *object_name* shall be the name of an instance. If none of the above are specified, the type of object_name determines the kind of object to which the command applies.

The option **-from** and **-to** may be used to specify one-to-one, one-to-many, many-to-one, or many-to-many transitions. The option **-paired** specifies one or more one-to-one transitions. At least one of these two choices shall be specified for each named transition.

If an empty list is specified in either the **-from** or **-to** *list*, it shall be expanded to all legal named power states for the specified *object_name*.

Verification tools shall emit an error when an illegal state transition occurs.

It shall be an error if a *from_state* or a *to_state* or a name in a *from_list* or *to_list* does not refer to a power state of the specified object (see 6.5).

If **add_state_transition** is specified for an instance of a macro (see 4.9.2.4) the command may only specify that a transition defined for the corresponding model is illegal for the specified instance of that model.

Example

```
add_state_transition -domain PDA
  -transition {turn_on  -from OFF_MODE    -to NORMAL_MODE}
  -transition {suspend  -from NORMAL_MODE -to SLEEP_MODE}
  -transition {resume   -from SLEEP_MODE  -to NORMAL_MODE}
  -transition {turn_off -from NORMAL_MODE -to OFF_MODE}
add_state_transition -domain PDA -update
  -transition {error1   -from OFF_MODE    -to SLEEP_MODE -illegal}
add_state_transition -domain PDA -update -complete
```

A self-transition (from one state to the same state) cannot be detected. It shall be an error if the same state is specified as both the **-from** state and the **-to** state.

A transition from a given state to a refinement of that state can occur.

A legal transition defined from a given state A to a refinement R of state A occurs when A is the current power state (and therefore R is not active), and then the additional conditions required to satisfy R become true, at which point R becomes active (and therefore A is no longer the current power state, although it is still active). Such a transition may include intermediate current power states that are refinements of A and abstractions of R, as well as the UNDEFINED state.

An illegal transition defined from A to R occurs when A is active and R becomes active without any intermediate step in which neither A nor R are active. Since R is a refinement of A, A remains active when R becomes active, so any sequence in which R becomes active while A is already active will satisfy the illegal transition definition.

A legal transition from a given state to an abstraction of that state can also occur.

A legal transition defined from a given state R to an abstraction A of state R occurs when R is the current power state (and therefore A is active also), and then conditions required to satisfy refinements of A become False, but the conditions required to satisfy A remain True. When this occurs, A becomes the current power state, and R is no longer active. Such a transition may include intermediate current power states that are abstractions of R and refinements of A, as well as the UNDEFINED state.

An illegal transition defined from a given state to an abstraction of that state cannot occur, because when the given state is active, the abstract state is also active, and therefore the abstract state cannot become active while the given state is active.

6.8 add_supply_state

Purpose	Add states to a supply port, a supply net, or a supply set function.	
Syntax	**add_supply_state** *object_name* {**-state** {*name* <*nom* | **off**>}}*	
Arguments	*object_name*	The name of the supply port, supply net, or supply set function. Hierarchical names are allowed.
	-state {*name* <*nom* | **off**>}	The *name* and value for a state of the supply object. The value can be a nominal voltage or **off**.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **add_supply_state** command defines a named supply state for a supply object. If a voltage value is specified, the supply net state is **FULL_ON** and the voltage value is the specified value; otherwise, if **off** is specified, the supply net state is **OFF**.

It shall be an error if *object_name* does not already exist.

Syntax example

```
add_supply_state PD.primary.power
  -state {active_state 0.90 }
  -state {off_state off}
```

6.9 apply_power_model

Purpose	Binds system-level IP power models to instances in the design and connects the interface supply set handles of a previously loaded power model.	
Syntax	**apply_power_model** *power_model_name* [**-elements** *instance_name_list*] [**-supply_map** {{*lower_scope_supply_set upper_scope_supply_set*}*}] [**-parameters** {*power_model_parameter design_object*}*}]	
Arguments	*power_model_name*	The simple name of a previously defined power model. See 6.11.
	-elements *instance_name_list*	The list of instances to which the specified power model applies.
	-supply_map {{*lower_scope_supply_set upper_scope_supply_set*}*}	How the interface supply sets of the lower scope connect with supply sets in the upper scope.
	-parameters {{*power_model_parameter design_object*}*}	The binding of design objects to power model parameters.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **apply_power_model** command describes the connections of the interface supply set handles of a previously loaded power model with the supply sets in the scope where the corresponding macro cells are instantiated.

The **apply_power_model** command sets the scope to each of a specified set of instances and executes the set of UPF commands in the power model *power_model_name*. Upon return, the current scope is restored to what it was prior to invocation. If a scope specified in *instance_name_list* is not found, further processing of remaining scopes in the *instance_name_list* is terminated and a `TCL_ERROR` is raised.

apply_power_model does not create a new name space for the loaded UPF file. The loaded UPF file is responsible for ensuring the integrity of both its own and the caller's name space as needed using existing Tcl name space management capabilities.

If **-elements** is specified, each instance name in the instance name list shall be a simple name or a hierarchical name rooted in the current scope. In this case, for the duration of the **apply_power_model** command, the current scope and design top instance are both set to the instance specified by the instance name and the design top module is set to the module type of that instance.

If **-elements** is not specified, then the system-level IP power model binding is not supported and the specified supply association is applied to all instantiations of targeted macro cells by the specified power model (see 6.11) under the current scope. The general precedence rules in 5.7 apply here as well.

When the **apply_power_model** command completes, the current scope, design top instance, and design top module all revert to their previous values.

Each pair in the **-supply_map** option implies an **associate_supply_set** command (see 6.10) of the following general form:

```
   associate_supply_set {lower_scope_supply_set upper_scope_supply_set}
```

The arguments of the **-supply_map** option need to be such that the implied **associate_supply_set** commands are legal.

The following also apply:

— The processing of this command shall follow the description in Clause 8.

— When **apply_power_model** is used with **-elements**, it shall be an error if the corresponding model for each instance does not match the model name specified in the **-for** option of **begin_power_model** (see 6.11) or the *power_model_name* when the **-for** option (of **begin_power_model**) is not specified.

The following also apply:

a) It shall be an error if **apply_power_model** is used more than once to apply a power model to a given instance.

b) It shall be an error if **apply_power_model** is used to apply a power model to an instance and **load_upf -scope** is also used to load a UPF file for the same instance.

Syntax example

```
apply_power_model upf_model -elements I1
  -supply_map {{PD.ssh1 ss1} {PD.ssh2 ss2}}
```

For other examples of using these commands, see Annex E.

6.10 associate_supply_set

Purpose	Associate two or more supply sets.	
Syntax	**associate_supply_set** *supply_set_name_list* [**-handle** *supply_set_handle*]	
Arguments	*supply_set_name_list*	A list of rooted names of supply sets.
	-handle *supply_set_handle*	The rooted name of a supply set of a power domain, power switch, or strategy.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **associate_supply_set** command associates two or more supply sets. Supply set association implicitly connects corresponding functions and as a result makes them electrically equivalent (see 4.5.5.3).

NOTE—Association of two supply sets is equivalent to explicitly connecting each pair of corresponding functions to a single intermediate supply port for that pair of functions using connect_supply_net.

Each supply set name in the *supply_set_name_list* can be either a supply set name or a supply set handle. A supply set handle may also be provided as the argument of the -**handle** option.

Supply set names are defined with the **create_supply_set** command (see 6.26). Supply set handles are dotted names (see 5.3.3.3) that refer to supply sets defined as part of a power domain, a power switch, or a strategy.

The following forms of supply set handle may be used:

a) The predefined supply set handle

    domain_name.**primary**

   is predefined for a power domain *domain_name* (see 6.20). Supply set handles for user-defined supply sets of a power domain are also permitted.

b) The predefined supply set handle for a power-switch *switch_name* (see 6.21) is

    switch_name.**switch_supply**.

c) The predefined supply set handles for an isolation cell strategy *isolation_name* (see 6.44) of a power domain *domain_name* are

    domain_name.isolation_name.**isolation_supply**

   if there is only one isolation supply set, or

    domain_name.isolation_name.**isolation_supply**[index]

   where index starts at 0, if there are multiple isolation supply sets.

d) The predefined supply set handles for a level-shifter strategy *level_shifter_name* (see 6.45) of power domain *domain_name* are

    domain_name.level_shifter_name.**input_supply**,
    domain_name.level_shifter_name.**output_supply**, and
    domain_name.level_shifter_name.**internal_supply**.

e) The predefined supply set handle for a retention strategy *retention_name* (see 6.49) of power domain *domain*_name is

    *domain_name.retention_name.***retention_supply**.

f) The predefined supply set handle for a repeater strategy *repeater_name* (see 6.48) of power domain *domain_name* is

    domain_name.repeater_name.**repeater_supply**.

When -**handle** is used, it shall be an error if the supply set handle is defined for a strategy or a power switch and more than one supply set is associated with that supply set handle.

Syntax examples

```
associate_supply_set {AON_SS PD1.primary PD2.backup PD3.isolation}
associate_supply_set {ISO_SS U1/PD1.my_iso.isolation_supply}
associate_supply_set  ISO_SS
  -handle U1/PD1.my_iso.isolation_supply\[1\]
```

6.11 begin_power_model

Purpose	Define a power model.	
Syntax	**begin_power_model** *power_model_name* [**-for** *model_list*]	
Arguments	*power_model_name*	The simple name of the power model.
	-for *model_list*	The names of the models to which the power model applies.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **begin_power_model** and **end_power_model** (see 6.11 and 6.29) commands define a power model containing other UPF commands. A power model is used to define the power intent of a model and shall be used in conjunction with one or more model representations. A power model defined with **begin_power_model** is terminated by the first subsequent occurrence of **end_power_model** in the same UPF file.

The **-for** option indicates that the power model represents the power intent for a family of model definitions. When **-for** is not specified, the *power_model_name* shall also be a valid model name.

A power model can be referenced by its simple name from anywhere in a power intent description. It shall be an error to have two power models with the same name.

To specify supplies coming into or out of the model, or a supply that has at least one data port related to it, use the **-supply** option of the **create_power_domain** command (see 6.20) for the top-scope power domain of the power model. Power states defined upon these supply set handles become the power state definition at the interface of the power model, which shall be consistent with the upper-scope system power states into which the corresponding upper-scope supply sets are mapped (see 6.9). The defined supply set handles are also called *interface supply handles* of the power model. Finally, the simstate simulation semantics described in 9.5 applies to all supply sets or supply set handles defined within a power model.

A power model can be used to represent one of following:

— A hard macro, indicated by the fact that the power model defines the attribute **UPF_is_hard_macro TRUE** on the model to which it applies. In this case, the UPF commands within a power model describe power intent that has already been implemented within the instances to which this power model is applied. The hard macro interface is a hard boundary; the parent context shall not modify the power intent specification inside the macro. In particular, no new logic or design objects shall be inferred within the cell instances targeted by such a power model.

— A soft macro, indicated by the fact that the instance to which this power model is applied has the attribute **UPF_is_soft_macro TRUE**. In this case, the UPF commands within the power model describe power intent that remains to be implemented. However, this power intent is intended to be used for separate implementation, and therefore the soft macro interface is still treated as a hard boundary; the parent context shall not modify the power intent specification inside the macro, and no new logic or design objects shall be inferred within the cell instances targeted by such a power model.

An encapsulation of UPF to be used together and possibly further modified by the parent context that applies this power model to an instance. This is indicated by the fact that the instance to which this power model is applied has neither **UPF_is_hard_macro TRUE** nor **UPF_is_soft_macro TRUE**.

A component power model used for defining power states and power consumption functions in order to model power consumption of a system in various states of its components. This is indicated by the presence of **add_parameter** commands to define the parameters used for power expression in the model, and the presence of power expressions as part of the power states of the model.

A power model can be applied to specific instances using **apply_power_model** (see 6.9). One power model applied to a given instance may apply another power model to a descendant instance.

A power model that is not referenced by an **apply_power_model** command does not have any impact on the power intent of the design.

Syntax example

```
begin_power_model upf_model -for cellA
    create_power_domain PD1 -elements {.} -supply {ssh1} -supply {ssh2}
    ;# other commands …
end_power_model
```

For more examples of using these commands, see Annex E and Annex H.

6.12 bind_checker

Purpose	Insert checker modules and bind them to instances.	
Syntax	**bind_checker** *instance_name* **-module** *checker_name* [**-elements** *element_list*] [**-bind_to** *module* [**-arch** *name*]] [**-ports** {{*port_name net_name*}*}] [**-parameters** {{*param_name param_value*}*}]	
Arguments	*instance_name*	The name used to instance the checker module in each instance.
	-module *checker_name*	The name of a SystemVerilog module containing the verification code. The verification code itself shall be written in SystemVerilog, but it can be bound to either a SystemVerilog or VHDL instance.
	-elements *element_list*	The list of instances.
	-bind_to *module* [**-arch** *name*]	The SystemVerilog module or VHDL entity/architecture for which all instances are the target of this command.
	-ports {{*port_name net_name*}*}	The association of signals to the checker's ports.
	-parameters {{*param_name param_value*}*}]	The specification of parameter values on the checker model where *param_name* is name of the parameter and *param_value* is the value of that parameter.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **bind_checker** command inserts checker modules into a design without modifying the design code or introducing functional changes. The mechanism for binding the checkers to instances relies on the SystemVerilog `bind` directive. The `bind` directive causes one module to be instantiated within another without having to explicitly alter the code of either. This facilitates the complete separation between the design implementation and any associated verification code.

Signals in the target instance are bound by position to inputs in the bound checker module through the port list. Thus, the bound module has access to any and all signals in the scope of the target instance by simply adding them to the port list, which facilitates sampling of arbitrary design signals.

If **-parameters** option is specified, the parameter by the name of *param_name* shall be set with the value *param_value*. For SystemVerilog it shall apply to parameter and for VHDL it shall apply to generics. The *param_value* shall be a constant value.

If **-bind_to** is specified, an instance of checker is created in every instance of the module. Otherwise, an instance of the checker is only created within the current scope.

port_name is a port defined on the interface of *checker_name* and *net_name* is a name of a net relative to the scope where *checker_name* is being instantiated.

It shall be an error if:

— *instance_name* already exists in **-bind_to** *module*.

— param_name does not exist on the checker module.

— param_value does not match with the type of param_name.

— param_value is not a constant value.

This command is for verification only; implementation tools shall ignore it.

Syntax example

```
bind_checker chk_p_clks
    -module assert_partial_clk
    -bind_to aars
    -ports {{prt1 clknet2} {port3 net4}}
    -parameters {
        {pd_name_string "pd_dut"}
        {int_param 12}
        {bit_param 1}
        {vec_param 2'b11}}
```

Modeling mutex assertions

To model mutex assertions (see 6.12 and 6.49), the assertions can be put in a SystemVerilog `checker_module` with following interface:

```
module checker_module ( save, restore, reset_a, clock_a );
input save, restore, reset_a, clock_a;
... different mutex assertions ...
endmodule
```

The **bind_checker** command would look like the following:

```
bind_checker mutex_checker_inst -module checker_module \
-ports { {save PDA.test_retention.save_signal } \
{ restore PDA.test_retention.restore_signal } \
{ reset_a reset_a } \
{ clock_a clock_a } }
```

6.13 connect_logic_net

Purpose	Connect a logic net to logic ports.	

Syntax	**connect_logic_net** *net_name*
	  -ports *port_list*
	  [**-reconnect**]	

Arguments	*net_name*	A simple name.
	-ports *port_list*	A list of ports on the interface of the current scope and/or on instances that are located in the current scope and its descendants.
	-reconnect	Allows a port that is already driven by a constant representing a default value to be driven instead by control signal *net_name*.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **connect_logic_net** command connects a logic net to the specified ports. The net is propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant subtree of the active UPF scope as required to support connections created by **connect_logic_net**. The connection from *net_name* in the active UPF scope to any element in *port_list* shall not cross any power-domain boundaries.

The net and ports shall be of a compatible type. The following HDL types are compatible with each other:

— SystemVerilog `logic`

— VHDL `std_ulogic`

It shall be an error if:

a) *net_name* is not the name of a logic net defined in the current HDL scope either explicitly or implicitly as a result of a **create_logic_net** command.

b) A HighConn port in *port_list* is already connected to a different net than *net_name*.

c) A HighConn input port in *port_list* is already driven by a constant value, unless the -reconnect option is specified.

d) A LowConn port in *port_list* is already connected to a different net than *net_name*.

e) The same port name occurs in the *port_list* of multiple **connect_logic_net** commands with different *net_name* arguments.

NOTE 1—Use **create_logic_port** (see 6.19) to create new logic ports on power-domain boundaries.

NOTE 2—This command exists to allow for the propagation of signals from a power-management block. Using this command to provide non-power control connections could cause the logic function to diverge from the HDL and is strongly discouraged.

Syntax example

```
connect_logic_net ena
  -ports {a U1/b}
```

6.14 connect_supply_net

Purpose	Connect a supply net to supply ports.	
Syntax	**connect_supply_net** *net_name* [**-elements** *element_list*] [**-ports** *port_list*] [**-pg_type** *pg_type_list*]* [**-vct** *vct_name*] [**-cells** *cell_list*] [**-domain** *domain_name*]	
Arguments	**-elements** *element_list*	A list of instance names to use for **-pg_type**.
	net_name	A simple name.
	-ports *port_list*	A list of rooted port names.
	-pg_type *pg_type_list*	An indirect connection specification via the *pg_type* on the instance's ports.
	-vct *vct_name*	A value conversion table (VCT) defining how values are mapped from UPF to an HDL model or from the HDL model to UPF.
	-cells *cell_list*	A list of cells to use for **-pg_type**.
	-domain *domain_name*	The domain indicates the scope to use for **-pg_type**.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **connect_supply_net** command connects a supply net to the specified ports. The net is propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant subtree of the current scope if -domain is not specified or in the descendant subtree of the scope of the domain specified with -domain, as required to support supply port/net connections made explicitly, automatically, or implicitly (see 9.2) This explicit connection overrides (has higher precedence than) the implicit and automatic connection semantics (see 9.2) that might otherwise apply. If **connect_supply_net** is used to connect a supply net defined with **create_supply_net -domain** (see 6.24) to a pg pin of an instance, then the instance shall be in the extent of power domain `D`.

Use the following:

— **-ports** to connect to supply ports.

— **-cells** to connect to all pins of the appropriate type (power or ground) on all the instances of the specified cells.

— **-domain** to connect to all pins of the appropriate type (power or ground) existing on the instances which are in the extent of the specified domain.

— **-pg_type** to connect to ports on the instances that have the specified *pg_type.*

— **-vct** to indicate that for every HDL port to which the net is connected, the supply net state shall be converted if it is being propagated into the HDL port (see 6.27) or the HDL port value shall be converted if it is being propagated onto the supply net (6.17). **-vct** is ignored for any connections of the supply net to supply ports defined in UPF.

— **-elements** to connect all pins of the appropriate type (power or ground) on the specified instances.

The following also apply:

- It shall be an error if any cell, domain, port, supply net, or instance specified in this command does not exist.
- It shall be an error if the value conversions specified in the VCT do not match the type of the HDL port.
- It shall be an error if neither **-ports** nor **-pg_type** is specified in a **connect_supply_net** command.
- The **-ports** option is mutually exclusive with the **-cells**, **-domain**, **-elements**, and **-pg_type** options.
- Automatic propagation of a supply net throughout the extent of a power domain is determined by its usage within the domain, such as primary supply, retention supply, etc.
- It shall be an error if *net_name* has not been previously created.
- If **-pg_type** is specified, it shall be an error if an instance does not exist or the specified attribute does not exist on any port of the instance.
- If **-ports** is not specified, **-pg_type** and one or more of **-cells**, **-domains**, and **-elements** shall be specified.

Syntax examples

```
connect_supply_net fb
  -ports {jk jb}

connect_supply_net mc
  -ports {rl}
  -vct SV_TIED_HI
```

The following command connects the supply net `VDDX` to the `VDD` port of a hierarchical instance `I1/I2`:

```
connect_supply_net VDDX -ports I1/I2/VDD
```

The following command connects the supply net `VDDX` to the `VDD` ports of all instances within hierarchical instance `I1/I2`:

```
connect_supply_net VDDX -ports [find_objects I1/I2 -pattern "*/VDD" -
    object_type port]
```

NOTE—Since a supply net handle such as PD.primary.power can be referenced anywhere a supply net is required, it is possible to use connect_supply_net to connect a supply set function to a port. This may be useful when hardening the interface to a block within a design. In particular, if a supply set SS in the parent context of an instance of a block B has been associated with a supply set handle inside of that instance, and it becomes necessary to harden block B for separate implementation as a macro, explicit supply ports can be defined on the interface to B, and the functions of supply set SS in the parent context can be connected to those ports using connect_supply_net. The functions of the supply set handle within B can be connected to those ports in the same manner. This maintains the association of the outer and inner supply sets, but at the same time explicitly shows the connections via ports on the interface of the block.

Since both supply set association and supply net connection make two supply objects electrically equivalent and have no other side effects, a supply set SS can be associated with a supply set handle and its functions can be connected via ports to the corresponding functions of the supply set handle, and both the association and the connections can coexist.

6.15 connect_supply_set

Purpose	Connect a supply set to particular elements.	
Syntax	**connect_supply_set** *supply_set_ref* {**-connect** {*supply_function pg_type_list*}}* [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-transitive** [<**TRUE** | **FALSE**>]]	
Arguments	*supply_set_ref*	The rooted name of the supply set.
	-connect {*supply_function pg_type_list*}	Define automatic connectivity for a *supply_function* of the *supply_set_ref* as ports having the specified *pg_type_list* attributes (see 6.14).
	-elements *element_list*	The list of instance names to add.
	-exclude_elements *exclude_list*	The list of instances to exclude from the *effective_element_list*.
	-transitive [<**TRUE** | **FALSE**>]	If **-transitive** is not specified at all, the default is **-transitive TRUE**. If **-transitive** is specified without a value, the default value is **TRUE**.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **connect_supply_set** command connects a supply set to the specified elements. The nets of the set are propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant subtree of the current scope as required to implement the supply net connection (see 9.2) This explicit connection overrides (has higher precedence than) the implicit and automatic connection semantics (see 9.2) that might otherwise apply.

This command applies to elements in the *effective_element_list* (see 5.9) as follows:

a) When *supply_set_ref* refers to a handle associated with a domain, the *prefilter_element_list* is filtered to only include elements within the extent of the domain.

b) When *supply_set_ref* refers to a handle associated with a strategy, the *prefilter_element_list* is filtered to only include all elements connected to the strategy's supply.

c) When *supply_set_ref* refers to a handle associated with a domain and -elements is not specified in the base command or any update, then all elements in the extent of the domain are added to the *aggregate_element_list*.

d) When *supply_set_ref* refers to a handle associated with a strategy and the *aggregate_element_list* is empty, all elements connected to the respective strategy supply are added to the *aggregate_element_list*.

-connect is additive, i.e., on a particular supply function, a subsequent invocation setting *pg_type_list* adds the additional *pg_type_list*.

NOTE—The *exclude_list* in **-exclude_elements** can specify elements that have not already been explicitly or implicitly specified via an explicit or implied *element_list*.

It shall be an error if:

- A particular *pg_type_list* is associated with more than one supply net for any given instance in **-connect**.
- More than one supply net is connected to the same port in an instance, even if the connection is the result of more than one command that connects supply nets, e.g., **connect_supply_set**, **connect_supply_net**, etc.
- Any element of *element_list* or *exclude_list* is not in a specified domain or strategy referenced in the *supply_set_handle*.

Syntax example

```
connect_supply_set some_supply_set
   -elements {U1/U_mem}
   -connect {power {primary_power}}
   -connect {ground {primary_ground}}
```

6.16 create_composite_domain

Purpose	Define a composite domain composed of one or more subdomains.		
Syntax	**create_composite_domain** *composite_domain_name* [**-subdomains** *subdomain_list*] [**-supply** {*supply_set_handle* [*supply_set_ref*]}] [**-update**]		
Arguments	*composite_domain_name*	The name of the composite domain; this shall be a simple name.	
	-subdomains *subdomain_list*	The **-subdomains** option specifies a list of rooted domain names, including any previously created composite domains.	**R**
	-supply {*supply_set_handle* [*supply_set_ref*]}	The **-supply** option specifies the *supply_set_handle* for *composite_domain_name*. If *supply_set_ref* is also specified, the domain *supply_set_handle* is associated with the specified *supply_set_ref*. The *supply_set_ref* may be any supply set visible in the current scope. See also 6.10.	**R**
	-update	Use **-update** if the *composite_domain_name* has already been defined.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

A *composite power domain* is a simple container for a set of power domains. Unlike a power domain, a composite domain has no corresponding physical region on the silicon. Attributes like power states and the primary *supply_set_handle* can be specified on a composite domain, but these attributes shall not be applied to subdomains. However, operations performed on the composite domain shall be applied to each subdomain, e.g., defining a strategy.

The following commands, applied to a composite domain, are applied to each subdomain if and only if the application of that command does not result in an error in any subdomain:

connect_supply_net

map_power_switch

map_retention_cell

set_isolation

set_level_shifter

set_repeater

set_retention

use_interface_cell

Only the primary supply handle can be specified in the **-supply** option. The following also apply:

a) Composite power domains can be used as a subdomain of other composite power domains.
b) Since a composite domain is simply a container, commands can still be applied to subdomains after composition.
c) For each subdomain: If a supply set is associated with the primary *supply_set_handle* of a subdomain, that supply set shall be equivalent to the primary supply set of the composite domain or declared as equivalent to the primary supply set of the composite domain (see also 6.43).
d) Commands applied to a subdomain do not affect any other subdomain or the composite domain.
e) Subdomains of a composite domain can still be referenced after composition, in the sense that their elements lists are valid after composition, and all aspects of the subdomain (e.g., strategies defined on them) can be referenced.

When the primary *supply_set_handle* and a *supply_set_ref* are specified in **-supply**, it is equivalent to the following:

```
associate_supply_set supply_set_ref
   -handle composite_domain_name.primary
```

Syntax example

```
create_composite_domain my_combo_domain_name
   -subdomains {a/pd1 b/pd2}
   -supply {primary could_be_on_ss}
```

6.17 create_hdl2upf_vct

Purpose	Define a VCT that can be used in converting HDL logic values into `state` type values.	
Syntax	**create_hdl2upf_vct** *vct_name* **-hdl_type** {<**vhdl** | **sv**> [*typename*]} **-table** {{*from_value to_value*}*}	
Arguments	*vct_name*	The VCT name.
	-hdl_type {<**vhdl** | **sv**> [*typename*]}	The HDL type for which the value conversions are defined.
	-table {{*from_value to_value*}*}	A list of the values of the HDL type to map to UPF `state` type values.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **create_hdl2upf_vct** command defines a VCT from an HDL logic type to the `state` type of the supply net value (see Annex B) when that value is propagated from HDL port to a UPF supply net. It shall provide a conversion for each possible logic value that the HDL port can have. **create_upf2hdl_vct** does not check that the set of HDL values are complete or compatible with any HDL port type.

vct_name provides a name for the value conversion table for later use with the **connect_supply_net** command (see 6.14). A VCT can be referenced by its simple name from anywhere in a power intent description. It shall be an error to have two VCTs with the same name. The predefined VCTs are shown in Annex B.

-hdl_type specifies the HDL type for which the value conversions are defined. This information allows a tool to provide completeness and compatibility checks. If the *typename* is not one of the language's predefined types or one of the types specified in the next paragraph, then it shall be of the form *library*.*pkg*.*type*.

The following HDL types shall be the minimum set of types supported. An implementation tool may support additional HDL types.

a) VHDL
   1) `Bit`, `std_[u]logic`, `Boolean`
   2) Subtypes of `std_[u]logic`
b) SystemVerilog
   `reg/wire`, `Bit`, `Logic`

-table defines the 1:1 conversion from HDL logic value to the UPF partially on and on/off states. The values are consistent with the HDL type values.

For example

— When converting from SystemVerilog *logic type*, the legal values are `0`, `1`, `X`, and `Z`.

— When converting from SystemVerilog or VHDL `bit`, the legal values are `0` or `1`.

— When converting from VHDL `std_[u]logic`, the legal values are `U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, and `-`.

The conversion values have no semantic meaning in UPF. The meaning of the conversion value is relevant to the HDL model to which the supply net is connected.

Syntax examples

```
create_hdl2upf_vct
    vlog2upf_vss
    -hdl_type {sv reg}
    -table {{X OFF} {0 FULL_ON} {1 OFF} {Z PARTIAL_ON}}
create_hdl2upf_vct
    stdlogic2upf_vss
    -hdl_type {vhdl std_logic}
    -table {{'U' OFF}
         {'X' OFF}
         {'0' OFF}
         {'1' FULL_ON}
         {'Z' PARTIAL_ON}
         {'W' OFF}
         {'L' OFF}
         {'H' FULL_ON}
         {'-' OFF}}
```

6.18 create_logic_net

Purpose	Define a logic net.
Syntax	**create_logic_net** *net_name*
Arguments	*net_name* A simple name.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.

The **create_logic_net** command creates a logic net in the current scope or identifies a logic net in the current scope.

The net's type is determined by the language of the scope where it is created. If the scope is

— SystemVerilog, the type is `logic`

— VHDL, the type is `std_ulogic`

NOTE—This command exists to allow for the propagation of signals from a power-management block. Using this command to provide non–power-control connections could cause the logic function to diverge from the HDL and is strongly discouraged.

Syntax example

```
create_logic_net iso_ctrl
```

6.19 create_logic_port

Purpose	Define a logic port.	
Syntax	**create_logic_port** *port_name* [**-direction** <**in** | **out** | **inout**>]	
Arguments	*port_name*	A simple name.
	-direction <**in** | **out** | **inout**>	The direction of the port. The default is **in**.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **create_logic_port** command creates a logic port in the current scope. Logic ports are effectively created before isolation and level-shifting strategies are applied (see 4.5.5); therefore, any isolation or level-shifting strategy defined for a power domain may apply to logic ports created on the boundary of that power domain, regardless of the order in which the **create_logic_port** command and the **set_isolation** (see 6.44) or **set_level_shifter** (see 6.45) commands occur, provided the logic port matches the criteria specified in the strategy.

The port's type is determined by the language of the scope where it is created. If the scope is

- SystemVerilog, the type is `logic`
- VHDL, the type is `std_ulogic`

The created port is equivalent to a module port created in SystemVerilog or VHDL with the same name and direction. Logic ports are sources, sinks, or both.

a) The LowConn of an input port is a source.
b) The HighConn of an input port is a sink.
c) The LowConn of an output port is a sink.
d) The HighConn of an output port is a source.
e) The LowConn of an inout port is both a source and a sink.
f) The HighConn of an inout port is both a source and a sink.

NOTE—This command exists to allow for the propagation of signals from a power-management block. Using this command to provide non–power-control connections could cause the logic function to diverge from the HDL and is strongly discouraged.

Syntax example

```
create_logic_port test_lp
-direction out
```

6.20 create_power_domain

Purpose	Define a power domain and its characteristics.		
Syntax	**create_power_domain** *domain_name* [**-atomic**] [**-elements** *element_list*] [**-subdomains** *domain_list*] [**-exclude_elements** *exclude_list*] [**-supply** {*supply_set_handle* [*supply_set_ref*]}]* [**-available_supplies** *supply_set_ref_list*] [**-define_func_type** {*supply_function pg_type_list*}]* [**-update**]		
Arguments	*domain_name*	The name of the power domain; this shall be a simple name rooted in the current scope.	
	-atomic	Define the minimum extent of the power domain.	
	-elements *element_list*	The list of instances to add.	**R**
	-subdomains *domain_list*	A list of rooted domain names.	**R**
	-exclude_elements *exclude_list*	The list of instances to exclude from the *effective_element_list*.	**R**
	-supply {*supply_set_handle* [*supply_set_ref*]}	The **-supply** option specifies the *supply_set_handle* for *domain_name*. If *supply_set_ref* is also specified, the domain *supply_set_handle* is associated with the specified *supply_set_ref*. The *supply_set_ref* may be any supply set visible in the current scope.	**R**
	-available_supplies *supply_set_ref_list*	A list of additional supply sets that are available for use by implementation tools to power cells inserted in this domain.	**R**
	-define_func_type {*supply_function pg_type_list*}	Define automatic connectivity for a *supply_function* of *domain_name*.**primary** (see 6.10) having the specified attributes in *pg_type_list*.	**R**
	-update	Use **-update** if the *domain_name* has already been defined.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

create_power_domain defines a power domain and the set of instances that are in the extent of the power domain. It may also specify whether the power domain can be partitioned further by subsequent commands.

-elements specifies a set of rooted instances included in the extent of the power domain. It shall be an error if any of these instances is already in the extent of an atomic power domain. For each instance in the extent of the power domain, any immediate descendant of that instance is also included in the extent of the power domain unless the descendant is an instance of a soft macro or is an instance that is already included in a power domain.

The following also apply:

— *element_list* shall contain instance names rooted in the current scope.

— Each design top instance (see 4.3.7) and each of its descendant instances shall be in the extent of exactly one power domain.

— When **-atomic** is specified, all elements originally included in the extent of the power domain shall always remain in the extent of that power domain.

— The power domain shall be created in the current scope.

— The **-elements** option shall be used at least once in the specification of a power domain using **create_power_domain**; this can be in the first invocation (i.e., without the **-update** option) or during the subsequent updates (i.e., with the **-update** option).

— If the value of *effective_element_list* (see 5.9) is an empty list, a domain with the name *domain_name* is created, but with an empty extent.

— If the value of the *effective_element_list* (see 5.9) is a period (`.`), the current scope is included in the extent of the domain.

NOTE 1—A design top instance can be included in the extent of a power domain created in the scope of that instance by specifying `-elements {.}` in the **create_power_domain** command.

NOTE 2—If the current scope is set to instance `i0`, then `create_power_domain PD -elements {.}` would include the current scope (`i0`) and all of its descendants in the power domain `PD`. In contrast, `create_power_domain PD -elements {i1 i2 ... ik}` would not include `i0` in the power domain, but would only include its descendants `i1`, `i2`, `...`, `ik`. In either case, the scope of the power domain `PD` is the same, because in both cases the current scope was `i0` when the **create_power_domain** command was executed.

-subdomains creates a simple container for domains. This is semantically equivalent to create_composite_domain (see 6.16). It shall be an error for both **-elements** and **-subdomains** to be specified in the same **create_power_domain** command.

An instance that has no parent or whose parent is in the extent of a different power domain is called a *boundary instance*.

The upper boundary of a power domain consists of

— the LowConn side of each port of each boundary instance in the extent of this domain.

The lower boundary of a power domain consists of

— the HighConn side of each port of each boundary instance in the extent of another power domain, where the parent of the boundary instance is in the extent of this domain, together with

— the HighConn side of each port of any macro cell instance in this power domain, for which the related supply set is neither identical to, nor equivalent to, the primary supply set of this domain.

The interface of a power domain consists of the union of the upper boundary and the lower boundary of the power domain.

create_power_domain also defines the supply sets that are used to provide power to instances within the extent of the power domain. The **-supply** option defines a supply set handle for a supply set used in the power domain.

A domain *supply_set_handle* may be defined without an association to a *supply_set_ref*. The association can be completed separately (see 6.10).

When both a *supply_set_handle* and a *supply_set_ref* are specified with **-supply**, the following supply set association is implied:

```
associate_supply_set supply_set_ref
  -handle domain_name.supply_set_handle
```

Each power domain has a predefined primary supply set. For verification, the primary supply set is implicitly connected to instances and logic inferred from the instances in the power domain. However, the primary supply set shall not be implicitly connected when any of the following apply:

a) An instance has at least one supply net explicitly or automatically connected and UPF simstate behavior (see 6.52) has not been enabled.

b) An instance has UPF simstate behavior disabled.

c) An instance is created as a result of a UPF command, e.g., isolation cells, level-shifters, power switches, and retention registers.

Implicit connections imply simulation semantics as specified in 4.7.2.

For implementation, the primary supply is realized as the supply connections that are common to all cells in the domain that require or propagate the primary supply rails of the domain.

Within a power domain, the predefined **primary** supply set is available for use by implementation tools as required to power instances in the extent of the domain, including isolation, level-shifter, retention, or repeater cells placed in the domain. Supply sets identified by command options of **set_repeater** (see 6.48) and **set_retention** (see 6.49) are also available to power repeater and retention cells, respectively, inserted into the domain. Collectively, the predefined primary supply set of a power domain and the supply sets identified by options of repeater and retention strategies associated with the domain are referred to as the *locally available supplies* of that domain.

The **-available_supplies** option specifies whether any additional supplies are also available for use, and if so, which supplies are available. If **-available_supplies** does not appear, all supply sets and supply set handles defined in or above the scope of the power domain are available for use by tools to power cells inserted into the power domain. If **-available_supplies** appears with an empty string argument, only the locally available supplies are available for use by tools to power cells inserted into the power domain. If **-available_supplies** appears with a non-empty string, the string shall be a list of the names of additional supply sets or supply set handles defined at or above the scope of the power domain that are also available for use by tools to power cells inserted into the power domain, in addition to the locally available supplies.

Any restrictions on the availability of supply sets or supply set handles for use by tools to power cells inserted into a given domain have no effect on the legality of referencing such supply sets or supply set handles in UPF commands to associate supply sets with supply set handles or to connect supply set functions explicitly, implicitly, or automatically to supply pins of an instance.

-define_func_type specifies the mapping from functions of the domain's primary supply set to `pg_type` attribute values in the *pg_type_list*. This mapping determines the automatic connection semantics used to connect the domain's primary supply to instances within the extent of the domain.

-update may be used to add elements and supplies to a previously created domain. It shall be an error if **-update** is used during the initial creation of *domain_name*.

It shall be an error

- — for any instance in the descendant subtree of an atomic power domain to be included in the extent of another power domain, unless that instance name is, or is in the descendant subtree of, an instance whose name appears in the *exclude_list*.
- — to remove an element from an atomic power domain.
- — to specify **-atomic** with **-update**.
- — to specify **-elements** or **-exclude_elements** with **-update** for an atomic power domain.

Syntax example

```
create_power_domain PD1 -elements {top/U1}
  -supply {primary}
  -supply {mem_array ss.mem}
create_power_domain PD2 -elements {.}
```

The following two examples are syntactically equivalent:

```
create_power_domain PD_COMB
  -subdomains {a/PD1 b/PD2}
  -supply {primary var_ss}

create_composite_domain PD_COMB
  -subdomains {a/PD1 b/PD2}
  -supply {primary var_ss}
```

6.21 create_power_state_group

Purpose	Create a name for a group of related power states.	
Syntax	**create_power_state_group** *group_name*	
Arguments	*group_name*	The simple name of the group to be created in the current scope.
Return value	Return the name of the created group or raise a TCL_ERROR if the group is not created.	

The **create_power_state_group** command defines a group name that can be used in the **add_power_state** command. The group *group_name* is defined in the current scope.

A power state group is used to collect related power states defined by **add_power_state**. The legal power states of a power state group define the legal combinations of power states of other objects in this scope or the descendant subtree, i.e., those combinations of states of those objects that can be active at the same time during operation of the design.

A power state group may be used to represent a virtual component made up of more than one instance. Power states defined for the power state group can represent the legal power states of the virtual component without having to change the design hierarchy to create a single instance for that component.

Power states of a power state group may be defined in terms of power states of supply sets, power domains, composite domains, instances, and other groups. Power states of two or more different power state groups may refer to power states of the same object.

It shall be an error if more than one fundamental state of the power state group is active at the same time (see 4.7.3).

If the power states of a power state group are defined as complete (see 6.5), it shall be an error if a situation occurs in which none of the legal power states of the group are active.

Example

```
create_power_state_group CPU_cluster
add_power_state -group CPU_cluster
  -state {RUN1
    -logic_expr {CPU0==RUN && CPU1==SHD && CPU2==SHD && CPU3==SHD}}
  -state {RUN2
    -logic_expr {CPU0==RUN && CPU1==RUN && CPU2==SHD && CPU3==SHD}}
  -state {RUN3
    -logic_expr {CPU0==RUN && CPU1==RUN && CPU2==RUN && CPU3==SHD}}
  -state {RUN4
    -logic_expr {CPU0==RUN && CPU1==RUN && CPU2==RUN && CPU3==RUN}}
```

6.22 create_power_switch

Purpose	Define a power switch.	
Syntax	**create_power_switch** *switch_name* [**-switch_type** <**fine_grain** | **coarse_grain** | **both**>] [**-output_supply_port** {*port_name* [*supply_net_name*]}] {**-input_supply_port** {*port_name* [*supply_net_name*]}}* {**-control_port** {*port_name* [*net_name*]}}* {**-on_state** {*state_name input_supply_port* {*boolean_expression*}}}* [**-off_state** {*state_name* {*boolean_expression*}}]* [**-supply_set** *supply_set_ref*] [**-on_partial_state** {*state_name input_supply_port* {*boolean_expression*}}]* [**-ack_port** {*port_name net_name* [*boolean_expression*]}]* [**-ack_delay** {*port_name delay*}]* [**-error_state** {*state_name* {*boolean_expression*}}]* [**-domain** *domain_name*] [**-instances** *instance_list*] [**-update**]	
Arguments	*switch_name*	The name of the switch instance to create; this shall be a simple name.
	-switch_type <**fine_grain** | **coarse_grain** | **both**>	The type of switch being defined. The default is **coarse_grain**.
	-output_supply_port {*port_name* [*supply_net_name*]}	The output supply port of the switch and, optionally, the supply net where this port connects. *supply_net_name* is a rooted name of a supply net or supply port. It shall be an error if the *supply_net_name* is not defined in the current scope.
	-input_supply_port {*port_name* [*supply_net_name*]}	An input supply port of the switch and, optionally, the net where this port is connected. *net_name* is a rooted name of a supply net or supply port. It shall be an error if the *net_name* is not defined in the current scope.
	-control_port {*port_name* [*net_name*]}	A control port on the switch and, optionally, the net where this control port connects. *net_name* is a rooted name of a logic net or logic port. It shall be an error if the *net_name* is not defined in the current scope.
	-on_state {*state_name input_supply_port* {*boolean_expression*}}	A named on state, the *input_supply_port* for which this is defined, and its corresponding Boolean expression.
	-off_state {*state_name* {*boolean_expression*}}	A named off state and its corresponding Boolean expression.
	-supply_set *supply_set_ref*	A supply set associated with the switch. *supply_set_ref* is a rooted name of a supply set or a supply set handle. It shall be an error if the *supply_set_ref* is not defined in the current scope.
	-on_partial_state {*state_name input_supply_port* {*boolean_expression*}}	A named partial-on state, the *input_supply_port* for which this is defined, and its corresponding Boolean expression.

	-ack_port {*port_name net_name* [*boolean_expression*]}	The acknowledge port on the switch and the logic net to which this port connects. A simple Boolean expression (see 5.4) can also be specified. *net_name* is a rooted name of a logic net or logic port. It shall be an error if the *net_name* is not defined in the current scope. If a null string is used as the *net_name* for **-ack_port**, the port and its Boolean expression are defined, but the port itself is unconnected.	
	-ack_delay {*port_name delay*}	The acknowledge delay for a given acknowledge port.	
	-error_state {*state_name* {*boolean_expression*}}	A named error state and its corresponding Boolean expression.	
	-domain *domain_name*	If specified, the scope of the domain is the scope in which the switch instance is created.	
	-instances *instance_list*	A list of technology leaf cell instance names that implements all or part of the specified switch. Instance names are the hierarchical names of the switch instances.	**R**
	-update	Use **-update** to allow the addition of **-instances**.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

The **create_power_switch** command defines an abstract model of a power switch. An implementation may use detailed power-switching structures that involve multiple, distributed power switches in place of a single abstract power switch, and/or re-order the specified embedded power switch chain.

The **-switch_type** option specifies the type of power switches (see 3.1) described by the **create_power_switch** command. The **switch_type** of an instance shall be determined by the Liberty attribute **switch_cell_type** or by the **UPF_switch_cell_type** attribute in the power model. If **-instances** is specified, **-switch_type** selects the instances defined in the *instance_list* based on the switch cell type attribute.

The following rules apply:

— **-switch_type coarse_grain**: This is the default. Only coarse grain switches shall be described by the **create_power_switch** command. It shall be an error if **-output_supply_port** is not specified.

— **-switch_type fine_grain**: Only fine grain switches shall be described by the **create_power_switch** command. It shall be an error if **-instances** is not specified. The **-output_supply_port** is ignored in this case.

— **-switch_type both**: Both coarse grain and fine grain switches shall be described by the **create_power_switch** command. It shall be an error if **output_supply_port** is not specified. **create_power_switch** does not specify the **-output_supply_port** for **fine_grain** switches.

Power-switch port names and port state names are defined in the scope of the switch instance and, therefore, can be referenced with a hierarchical name in the same way that any other instance ports can be referenced. For example, the command

```
create_power_switch PS1
    -output_supply_port {outp}
    -input_supply_port {inp}
...
```

creates an instance `PS1` in the current scope and creates supply ports `outp` and `inp` within the `PS1` instance. The switch supply ports can then be referred to as `PS1/inp` and `PS1/outp`.

The abstract power-switch model has one or more input supply ports and one output supply port. Output supply port is specified only when the switch type is **coarse_grain** or **both**. Each input supply port is effectively gated by one or more control expressions defined by *on_state* or *on_partial_state* expressions. An *on_state* expression specifies when a given input supply contributes to the output without limiting current. An *on_partial_state* expression specifies when a given input supply contributes to the output in a current-limited manner. Each input supply may have multiple *on_state* and/or *on_partial_state* expressions.

The abstract power-switch model may also have one or more error_state expressions defined. Any *error_state* expressions defined for a given power switch represent control input conditions that are illegal for that switch.

The abstract power-switch model may also have a single off_state expression defined. The *off_state* expression represents the condition under which no *on_state* or *on_partial_state* expression is *True*. If not specified explicitly, the *off_state* expression defaults to the complement of the disjunction of all the *on_state*, *on_partial_state*, and *error_state* expressions defined for the power switch.

It shall be an error if the *off_state* expression is explicitly defined and it evaluates to True when an *on_state* or *on_partial_state* expression also evaluates to *True*.

An *on_state* or *on_partial_state* specification for a power switch contributes a value to the computation of the power switch output port's value at any given time. If an *on_state* or *on_partial_state* Boolean expression for a given input supply port refers to an object with an unknown (X or Z) value, and that input supply port has a net state other than OFF, then the contributed value is {**UNDETERMINED,** unspecified}. If an *on_state* Boolean expression for a given input supply port evaluates to True, then the contributed value is the value of that input supply port. If an *on_partial_state* Boolean expression for a given input supply port evaluates to *True*, then the contributed value is the degraded value of that input supply port. The degraded value of an input supply port is the value of that port, except that if the port value's net state is **FULL_ON**, the degraded value's net state is **PARTIAL_ON**.

The value of the output supply port of a power switch is determined as follows. At any given time:

a) The output supply takes on the value {**UNDETERMINED**, unspecified} if:
   1) any *error_state* condition is *True*, or
   2) an explicit *off_state* condition and any *on_state* or *on_partial_state* condition are both *True*, or
   3) any contributed value has a net state of **UNDETERMINED**, or
   4) any two contributed values have different voltage values.
b) Otherwise, the switch output takes on any contributed value whose net state is **FULL_ON**, if there is one.
c) Otherwise, the switch output takes on any contributed value whose net state is **PARTIAL_ON**, if there is one.
d) Otherwise, the switch takes on the value {**OFF**, unspecified}.

A power switch is in an off state when the (explicit or default) **-off_state** is True. An off power switch begins to turn on when an **-on_state** or **-on_partial_state** condition becomes True. A power switch is in a fully on state when some **-on_state** condition is True. A fully on power switch begins to turn off when the last remaining -on_state condition that was True becomes False, or when an explicit **-off_state** condition becomes True.

If an **-ack_port** argument is specified, an acknowledge value is driven onto the specified *port_name delay* time units after the switch begins to turn on and the inverse acknowledge value is driven onto the specified *port_name delay* time units after the switch begins to turn off. For verification, the initial value of the specified ack port is the inverse acknowledge value, which indicates that the power switch is in the OFF state at time zero.

If the supply set of the power switch is in a power state with a **NORMAL** simstate, then the acknowledge value is a logic `0` or logic `1`. If a *Boolean expression* is specified for **-ack_port**, it shall be a simple Boolean expression (see 5.4). That expression shall determine the acknowledge value for a transition to **FULL_ON**, and its negation shall determine the acknowledge value for a transition to **OFF**; otherwise the acknowledge value defaults to logic `1` for a transition to **FULL_ON** and logic `0` for a transition to **OFF**. If **-ack_delay** is specified, the delay may be specified as a time unit, or it may be specified as a natural integer, in which case the time unit shall be the same as the simulation precision; otherwise, the delay defaults to `0`.

If **-supply_set** is specified for a switch, it powers logic or timing-control circuitry within the switch. When the supply set simstate is anything other than **NORMAL**, the acknowledge ports are corrupted. If a supply set is not associated with a switch, then the following shall apply:

— It shall be an error if any acknowledge ports are specified.

— The receiving supply of the control ports is not defined.

-instances specifies that the power-switch functionality exists in the HDL design and *instance_list* denotes the list of instances providing part or all of this functionality. If **-instances** is specified, and a list of instances is given, then the switch may be implemented as multiple switches, in which case the multiple instances may have characteristics different from those specified by the **create_power_switch** command, particularly with regard to input and output supply connections. Each element in the *instance_list* shall be a hierarchical name rooted in the current scope.

If an empty string appears in an *instance_list*, this indicates that an instance may have been created and then optimized away. Such an instance shall not be re-inferred or reimplemented by subsequent tool runs.

Updating **-instances** adds the new instance names to the existing instance list. **-update** adds information to the base command executed in the same scope in which the object exists or is to be created.

If **-switch_type** is fine_grain or both, the following shall apply to all power switch instances:

— Share the same supply net connected to **input_supply_port**.

— Share the same supply set specified in **-supply_set** used for ack port association.

— Share the same **control_port** and **ack_port** pin names.

— If only one **control_port** is specified, the control port shall be broadcasted to all instances.

— If one **control_port/ack_port** pair is specified, the instances shall be connected in a sequence such that the **ack_port** of one instance is connected to the **control_port** of the next instance (order unspecified).

— If more than one **control_port** is specified, the control ports shall be connected based on *port_name.*

— If more than one **control_port/ack_port** pair is specified, the instances shall be connected in a sequence such that each **ack_port** is connected to the corresponding **control_port** of the next instance (order unspecified).

The following also apply:

— Any name in a *boolean_expression* shall refer to a control port of the switch.
— All states not covered by the on, on_partial, off, and error states are anonymous error states.
— If the implementation of a switch can not be inferred, **map_power_switch** (see 6.34) can be used to specify it.
— If *net_name* is not specified for any of the switch's port definitions, **connect_logic_net** (see 6.13) or **connect_supply_net** (see 6.14) can be used to create the port connections.
— Each state name shall be unique for a particular switch.
— Any *port_name*s specified in this command are user defined (e.g., `input_supply`).

NOTE 1—**create_power_switch** can be used to define an abstract power switch that implementation tools may expand into multiple switches. **create_power_switch** can also be used to specify the need for a specific switch that can then be mapped to a specific switch implementation using **map_power_switch**. It is not meant to be used as a single definition representing multiple physical switches to be mapped with **map_power_switch**.

NOTE 2—**create_power_switch** provides relatively simple, general abstract functionality. HDLs can be used to model switch functionality that cannot be captured with **create_power_switch**.

Power-switch examples

Example 1: Simple switch

This switch model has a single supply input and a single control input. The switch is either on or off, based on the control input value. Since net names are not specified for each port, **connect_supply_net** (see 6.14) can be used to connect a net to each port.

```
create_power_switch simple_switch
-output_supply_port {vout}
-input_supply_port {vin}
-control_port {ss_ctrl}
-on_state {ss_on vin { ss_ctrl }}
-off_state {ss_off { ! ss_ctrl }}
```

The following is a variant of the simple switch in which the nets associated with the ports are defined as part of the **create_power_switch** command (see 6.21).

```
create_power_switch simple_switch2
-output_supply_port {vout VDD_SW}
-input_supply_port {vin VDD}
-control_port {ss_ctrl sw_ena}
-on_state {ss_on vin { ss_ctrl }}
-off_state {ss_off { ! ss_ctrl }}
```

Example 2: Two-stage switch

This switch model represents a switch that turns on in two stages. The switch has one supply input and two control inputs. One control input represents the enable for the first stage; the other represents the control for the second stage. When only the first control is on, the switch output is in a partial on state; when the second is on, the switch output is in a fully on state. The switch is off if neither control input is on.

```
create_power_switch two_stage_switch
-output_supply_port {vout}
-input_supply_port {vin}
-control_port {trickle_ctrl}
-control_port {main_ctrl}
-on_partial_state {ts_ton vin { trickle_ctrl }}
-on _state {ts_mon vin { main_ctrl }}
-off_state {ts_off { ! trickle_ctrl && ! main_ctrl }}
```

The following is a variant of the two-stage switch model in which an **-ack_port** signals completion of the switch turning on. The time required for the switch to turn on is modeled by the **-ack_delay**. Since an **-ack_port** is involved, the command needs to include specification of the supply set that powers the logic driving the ack signal. The ack signal is defined separately. In this model, as in the preceding simple switch variant, the supply and control ports are associated with corresponding nets, so they do not need to be connected in a separate step.

```
create_power_switch two_stage_switch2
-output_supply_port {vout VDD_SW}
-input_supply_port {vin VDD}
-control_port {trickle_ctrl t_ena}
-control_port {main_ctrl m_ena}
-on_partial_state {ts_ton vin { trickle_ctrl }}
-on_state {ts_mon vin { main_ctrl }}
-off_state {ts_off { ! trickle_ctrl && ! main_ctrl }}
-ack_port {ts_ack "" 1}
-ack_delay {ts_ack 100ns}
-supply_set ss_aon
```

Example 3: Muxed switch

This switch model represents a mux that determines which of two different input supplies is connected to the output supply port at any given time. The two input supplies can be driven by different root supply drivers and may have different state/voltage values. One control input determines which of the two input supplies is selected; the other control input gates the selected supply to the output supply.

```
create_power_switch muxed_switch
-output_supply_port {vout}
-input_supply_port {vin0}
-input_supply_port {vin1}
-control_port {ms_sel}
-control_port {ms_ctrl}
-on_state {ms_on0 vin0 { ms_ctrl && ! ms_sel }}
-on_state {ms_on1 vin1 { ms_ctrl && ms_sel }}
-off_state {ms_off { ! ms_ctrl }}
```

The following is a variant of the muxed switch in which there are two independent selection control inputs, and an error state is defined to check for mutual exclusion.

```
create_power_switch muxed_switch2
-output_supply_port {vout}
-input_supply_port {vin0}
-input_supply_port {vin1}
-control_port {ms_sel0}
-control_port {ms_sel1}
-control_port {ms_ctrl}
-on_state {ms_on0 vin0 { ms_ctrl && ms_sel0 }}
```

```
-on_state {ms_on1 vin1 { ms_ctrl && ms_sel1 }}
-off_state {ms_off { ! ms_ctrl }}
-error_state {conflict { ms_sel0 && ms_sel1 }}
```

Example 4: Overlapping muxed switch

This switch model represents a supply mixer that allows a smooth transition between two different supplies. Like the muxed switch, it has two supply inputs and both selecting and gating control inputs, but in this case it can select both input supplies at the same time. The input supplies may have different states, and may even be driven by different root supply drivers, provided that their voltages are the same when both inputs are enabled (in an on state or on_partial state).

```
create_power_switch overlapping_muxed_switch
-output_supply_port {vout}
-input_supply_port {vin0}
-input_supply_port {vin1}
-control_port {oms_sel0}
-control_port {oms_sel1}
-control_port {oms_ctrl}
-on_state {oms_on0 vin0 { oms_ctrl && oms_sel0 }}
-on_state {oms_on1 vin1 { oms_ctrl && oms_sel1 }}
-off_state {oms_off { !oms_ctrl || { !oms_sel0 && !oms_sel1 } }}
```

Example 5: Chain of embedded macro switches

This example represents a switch model that is composed of only embedded macro switches, where the embedded macro switches have a single control port and a single ack port. Please note that the order of connection is not specified by UPF.

```
set mem_inst [find_objects . -object_type model -pattern "*MEM*SW*"
  -transitive TRUE]

create_power_switch ram_chain_0
  -instances $mem_inst
  -switch_type fine_grain
  -input_supply_port {vin VDD}
  -control_port {pwr_on ram_on}
  -ack_port {pwr_on_ack ram_on_ack}
  -on _state {ts_on vin { pwr_on }}
  -off_state {ts_off { !pwr_on }}
  -supply_set ss_aon
```

Example 6: Mixed chain of logic switches and embedded macro switches

This example represents a switch model where logic switches are connected along with embedded macro switches in the same power switch chain. Please note that the order of connection is not specified by UPF.

```
set mem_inst [find_objects . -object_type model -pattern "*MEM*SW*"
  -transitive TRUE]

create_power_switch ram_chain_0
  -instances $mem_inst
  -switch_type both
  -input_supply_port {vin VDD}
  -output_supply_port {vout VDDSW}
  -control_port {pwr_on ram_on}
```

```
-ack_port {pwr_on_ack ram_on_ack}
-on _state {ts_on vin { pwr_on }}
-off_state {ts_off { !pwr_on }}
-supply_set ss_aon
```

6.23 create_pst (legacy)

Purpose	Create a power state table (PST).	
Syntax	**create_pst** *table_name* **-supplies** *supply_list*	
Arguments	*table_name*	The PST name. *table_name* is a simple name in the current scope.
	-supplies *supply_list*	The list of supply nets or ports to include in each power state of the design. The supplies are listed as rooted names in the current scope.
Return value	Return the name of the created PST or raise a `TCL_ERROR` if the PST is not created.	

This is a legacy command; see also 6.2 and Annex D.

The **create_pst** command defines a PST name and a set of supply nets for use in **add_pst_state** commands (see 6.6). The PST *table_name* is defined in the current scope.

A PST is used for implementation—specifically for synthesis, analysis, and optimization. It defines the legal combinations of states, i.e., those combinations of states that can exist at the same time during operation of the design.

create_pst can only be used with **add_pst_state** (and vice versa). This combination and use of **add_power_state** (see 6.5) are two methods for specifying power state information. Power state specifications and default state definitions form an exhaustive specification of all of the legal power states of the system.

It shall be an error if

— *table_name* conflicts with any name declared in the current scope.

— a specified supply net or supply port specified in *supply_list* does not exist.

Syntax example

```
create_pst MyPowerStateTable -supplies {PN1 PN2 SOC/OTC/PN3}
```

6.24 create_supply_net

6.24.1 Overview

Purpose	Create a supply net.	
Syntax	**create_supply_net** *net_name* [**-domain** *domain_name*][**-reuse**] [**-resolve** <**unresolved** | **one_hot** | **parallel** | **parallel_one_hot** |*resolution_function_name* >]	
Arguments	*net_name*	A simple name.
	-domain *domain_name*	The domain in whose scope the supply net is to be created.
	-reuse	Extend availability of a supply net previously defined for another domain for use in the extent of this domain.
	-resolve <**unresolved** | **one_hot** | **parallel** | **parallel_one_hot** | *resolution_function_name*>	A resolution mechanism that determines the state and voltage of the supply net when the net has multiple supply sources (see 6.24.3). If no option is specified, the behavior for resolution is the same as for **unresolved**.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **create_supply_net** command creates a supply net. If **-domain** is not specified, the supply net is created in the current scope, and the supply net is available for use in the extent of any domain created at or below this scope.

If **-domain** is specified, the supply net is created in the scope of that domain and the supply net is available for use in the extent of the domain.

If **-reuse** is specified, the specified supply net shall have been created by a previously executed command, and this existing supply net is made available for use in the extent of the domain specified by the **-domain** option. In this case:

a) **-domain** shall also be specified on both this and the creating command;

b) **-resolve** shall not conflict with that of the creating command.

The net is propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant tree of the scope in which the net is created as required by implicit and automatic connections of supply sets (see 6.20).

The use of a supply net by implementation tools to power cells that they insert in the extent of a domain is subject to the supply set availability rules (see 6.20). A supply net is available for use in a domain only if it is associated with a function of an available supply set in that domain.

The following also apply:

— It shall be an error if *domain_name* is not the name of a previously created power domain.

— When **-reuse** is specified, it shall be an error if *net_name* is not defined for another power domain in the same scope by another **create_supply_net** command.

— When the parameter for **-resolve** is **unresolved**, the supply net shall have only one source (see 6.24.2). For all other parameters to **-resolve**, the requirements on the drivers and sources of the net are as defined in 6.24.3.

Syntax example

```
create_supply_net local_vdd_3
   -resolve one_hot
```

6.24.2 Supply net resolution

Supply nets are often connected to the output of a single switch. However, certain applications, such as on-chip voltage scaling, may require the outputs of multiple switches or other supply drivers to be connected to the same supply net (either directly or via supply port connections). In these cases, a resolution mechanism is needed to determine the state and voltage of the supply net from the state and voltage values supplied by each of the supply drivers to which the net is connected.

A supply net that specifies an **unresolved** resolution cannot be connected to more than one supply source.

6.24.3 Resolution methods

The semantics of each predefined resolution method are as follows:

a) **unresolved**

   The supply net shall be connected to at most one supply source. This is the default.

b) **one_hot**

   Multiple supply sources, each having a unique driver, may be connected to the supply net.

   A supply net with **one_hot** resolution has a deterministic state only when no more than one source drives the net at any particular point in time. If at any point in time more than one supply source driving the net is anything other than **OFF**, the state of the supply net shall be **UNDETERMINED**, the voltage value of the supply net shall be unspecified, and implementations may issue a warning or an error.

   1) If all supply sources are **OFF**, the state of the supply net shall be **OFF**, and the voltage value of the supply net shall be unspecified.
   2) If only one supply source is **FULL_ON** and all other sources are **OFF**, the state of the supply net shall be **FULL_ON**, and the voltage value of the corresponding source shall be assigned to the supply net.
   3) If only one supply source is **PARTIAL_ON** and all other sources are **OFF**, the state of the supply net shall be **PARTIAL_ON** and the voltage value of the corresponding source shall be assigned to the supply net.
   4) If any source is **UNDETERMINED**, the state of the supply net shall be **UNDETERMINED**, and the voltage value of the supply net shall be unspecified.

c) **parallel**

   Multiple supply sources, sharing a common root supply driver, may be connected to the supply net.

The **parallel** resolution allows more than one potentially conducting path to the same root supply driver, as if the switches had been connected in parallel. It shall be an error if any of these potentially conducting paths can be traced to more than one root supply driver.

1) If all of the supply sources are **FULL_ON**, then the supply net state is **FULL_ON** and the voltage value is the value of the root supply driver.
2) If all the supply sources driving the supply net are **OFF**, the state of the supply net shall be **OFF** and the voltage is unspecified.
3) If any of the sources is **UNDETERMINED**, the resolution is **UNDETERMINED**; otherwise,
    i) If there is at least one **PARTIAL_ON** source, the supply net shall be **PARTIAL_ON** and the voltage value is the value of the root supply driver.
    ii) If there is at least one source that is **OFF** and at least one that is **FULL_ON** or **PARTIAL_ON**, the supply net shall be **PARTIAL_ON** and the voltage value is the value of the root supply driver. The voltage value of the **PARTIAL_ON** supply net shall be the voltage value of the root supply driver.

d) **parallel_one_hot**

Multiple supply sources may be connected to the supply net. A source may share a common root supply driver with one or more other sources. At most one root supply driver shall be **FULL_ON** at any particular time with all sources sharing that driver resolved using parallel resolution.

The **parallel_one_hot** resolution allows resolution of a supply net that has multiple root supply drivers where each driver may have more than one path through supply sources to the supply net. Each unique root supply driver is identified and **one_hot** resolution shall be applied to the drivers, then **parallel** resolution shall be applied to each supply source connecting the **one_hot** root supply driver to the supply net.

Resolution semantics may also be specified by a user-defined resolution function. When a user-defined resolution function is specified, there are no restrictions on the number of input sources or the number of root supply drivers involved, and the function is responsible for defining any restrictions on the values of inputs as well as the algorithm for determining the output result.

Examples

The following supply net resolution functions support multi-source supply nets driven by different root supplies. This requires handling multiple inputs with potentially different voltages as well as different states. The resolution function treats inputs with the following precedence: **UNDETERMINED**, **FULL_ON, PARTIAL_ON**, **OFF**. The resolution function returns the average voltage of all **FULL_ON** or **PARTIAL_ON** inputs, as appropriate.

SystemVerilog resolution function

```
function automatic supply_net_type MultiSourceResolution (input
    supply_net_type sources[]);
  supply_net_type ResolvedValue;
  int FullOnVolts = 0;
  int PartOnVolts = 0;
  int FullOnCount = 0;
  int PartOnCount = 0;
  int UndetCount = 0;
  foreach (sources[i]) begin
    if (sources[i].state==UNDETERMINED) begin
        UndetCount++;
    end
```

```
      else if (sources[i].state==FULL_ON) begin
          FullOnVolts += sources[i].voltage;
          FullOnCount++;
      end
      else if (sources[i].state==PARTIAL_ON) begin
          PartOnVolts += sources[i].voltage;
          PartOnCount++;
      end
    end
    if (UndetCount > 0) begin
        ResolvedValue.state = UNDETERMINED;
        ResolvedValue.voltage = 0; // representing ‘unknown’
    end
    else if (FullOnCount > 0) begin
        ResolvedValue.state = FULL_ON;
        ResolvedValue.voltage = FullOnVolts / FullOnCount; // average value
    end
    else if (PartOnCount > 0) begin
        ResolvedValue.state = PARTIAL_ON;
        ResolvedValue.voltage = PartOnVolts / PartOnCount; // average value
    end
    else begin
        ResolvedValue.state = OFF;
        ResolvedValue.voltage = 0; // representing ‘irrelevant’
    end
    return (ResolvedValue);
  endfunction
```

VHDL resolution function

```
  function MultiSourceResolution (sources: supply_net_type_vector) return
     supply_net_type is
    variable ResolvedValue: supply_net_type;
    variable FullOnVolts: Natural := 0;
    variable PartOnVolts: Natural := 0;
    variable FullOnCount: Natural := 0;
    variable PartOnCount: Natural := 0;
    variable UndetCount: Natural := 0;
  begin
    for i in sources’length loop
      if (sources(i).state = UNDETERMINED) then
          UndetCount := UndetCount + 1;
      elsif (sources(i).state==FULL_ON) then
          FullOnVolts := FullOnVolts + sources(i).voltage;
          FullOnCount := FullOnCount + 1;
      elsif (sources(i).state==PARTIAL_ON) then
          PartOnVolts := PartOnVolts + sources(i).voltage;
          PartOnCount := PartOnCount + 1;
    end loop;
    if (UndetCount > 0) then
        ResolvedValue.state := UNDETERMINED;
        ResolvedValue.voltage := 0; -- representing ‘unknown’
    elsif (FullOnCount > 0) then
        ResolvedValue.state := FULL_ON;
        ResolvedValue.voltage := FullOnVolts / FullOnCount; -- average value
    elsif (PartOnCount > 0) then
        ResolvedValue.state := PARTIAL_ON;
        ResolvedValue.voltage := PartOnVolts / PartOnCount; -- average value
    else
        ResolvedValue.state := OFF;
        ResolvedValue.voltage := 0; -- representing 'irrelevant'
    end if;
    return ResolvedValue;
  end;
```

6.24.4 Supply nets defined in HDL

The declaration of any VHDL `signal` or SystemVerilog `wire` or `reg` as a `supply_net_type` from the package UPF (see 11.2) is equivalent to calling **create_supply_net** for every instance of that declaration, where the *net_name* is the name of the VHDL `signal` or SystemVerilog `wire` or `reg`, and the scope is the instance. If the VHDL or SystemVerilog declaration includes a resolution function, the equivalent **create_supply_net** command also includes the **-resolve** option with the specified resolution function name.

6.25 create_supply_port

Purpose	Create a supply port on a instance.	
Syntax	**create_supply_port** *port_name* [**-domain** *domain_name*] [**-direction** <**in** | **out** | **inout**>]	
Arguments	*port_name*	A simple name.
	-domain *domain_name*	The domain where this port defines a supply net connection point.
	-direction <**in** | **out** | **inout**>	The direction of the port. The default is **in**.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **create_supply_port** command defines a supply port at the scope of the power domain when **-domain** is specified or at the current scope if **-domain** is not specified.

-direction defines how state information is propagated through the supply network as it is connected to the port. If the port is an input port, the state information of the external supply net (see 6.24) connected to the port shall be propagated into the instance. Likewise, for an output port, the state information of the internal supply net connected to the port shall be propagated outside the instance.

Supply ports with direction **inout** shall be used to connect resolved supply nets (see 9.1). Supply ports are loads, sources, or both, as follows:

a) The LowConn of an input port is a source.

b) The HighConn of an input port is a sink.

c) The LowConn of an output port is a sink.

d) The HighConn of an output port is a source.

e) The LowConn of an inout port is both a source and a sink.

f) The HighConn of an inout port is both a source and a sink.

Supply ports may be defined in HDL. If a VHDL or SystemVerilog `port` is declared as a `supply_net_type` from the package UPF (see 11.2), this is equivalent to calling **create_supply_port** for every instance of that declaration, where the *port_name* is the name of the VHDL or SystemVerilog `port`, and the scope is the instance.

Syntax example

```
create_supply_port VN1
    -direction inout
```

6.26 create_supply_set

6.26.1 General

Purpose	Create or update a supply set, or update a supply set handle.		
Syntax	**create_supply_set** *set_name* [**-function** {*func_name net_name*}]* [**-update**]		
Arguments	*set_name*	The simple name of the supply set or a supply set handle.	
	-function {*func_name net_name*}	The **-function** option defines the function (*func_name*) a supply net provides for this supply set. *net_name* is a rooted name of a supply net or supply port or a supply net handle. It shall be an error if the net_name is not defined in the current scope.	**R**
	-update	Use **-update** if the *set_name* has already been defined.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

create_supply_set creates the supply set name within the current scope in the UPF name space. This command defines a *supply set* as a collection of supply nets each of which serve a specific function for the set.

-update is used to signify that this **create_supply_set** call refers to a supply set that was previously defined using **create_supply_set**, or to a supply set handle that was previously defined implicitly or explicitly using **create_power_domain** (see 6.20). Referencing a previously created supply set or supply set handle without the **-update** argument shall be an error. Using the **-update** argument for a supply set that has not been previously defined shall be an error. Specifying a supply set handle that has not been previously defined shall be an error.

When **-function** is specified, *func_name* shall be one of the following: **power**, **ground**, **nwell**, **pwell**, **deepnwell**, and **deeppwell**. The **-function** option associates the specified *func_name* of this supply set with the specified *supply_net_name*. If the same *func_name* is associated with two different supply nets, it shall be an error if those supply nets are not the same. The *supply_net_name* may be a reference to a supply net in the descendant hierarchy of the current scope using a supply net handle (see 5.3.3.3).

Syntax example

```
create_supply_set relative_always_on_ss
  -function {power vdd}
```

```
    -function {ground vss}
  create_supply_set PD1.primary -update
        -function {nwell bias}
```

NOTE 1—A supply set function may also be referenced using a supply net handle (see 5.3.3.3), regardless of whether or not a supply net has been associated with the function name, as follows:

```
supply_set_name.function_name
```

NOTE 2—A group of supply sets with a common ground can be represented as follows:

```
set_equivalent -nets {VSS SS1.ground SS2.ground}
```

However this intent must be explicitly specified for implementation:

```
create_supply_set SS1 -update -supply {ground VSS}
    create_supply_set SS2 -update -supply {ground VSS}
```

6.26.2 Implicit supply net

If no supply net is associated with a supply set's function and that function is used in the design, an implicit supply net with an anonymous name shall be created for use in verification and analysis. When the UPF specification is used for implementation, a supply net shall not be implicitly created for a supply set function that has no associated supply net. A tool may issue a warning or an error if a supply set's function does not have an explicit supply net association.

6.27 create_upf2hdl_vct

Purpose	Define VCT that can be used in converting UPF `supply_net_type` values into HDL logic values.	
Syntax	**create_upf2hdl_vct** *vct_name* **-hdl_type {<vhdl | sv>** [*typename*]**}** **-table {{***from_value to_value***}*}**	
Arguments	*vct_name*	The VCT name.
	-hdl_type {<vhdl | sv> [*typename*]**}**	The HDL type for which the value conversions are defined.
	-table {{*from_value to_value***}*}**	A list of UPF `state` type values to map to the values of the HDL type.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **create_upf2hdl_vct** command defines a VCT for the `supply_net_type.state` value (see Annex B when that value is propagated from a UPF supply net into a logic port defined in an HDL. It provides a 1:1 conversion for each possible combination of the partially on and on/off states. **create_upf2hdl_vct** does not check that the values are compatible with any HDL port type.

vct_name provides a name for the value conversion table for later use with the **connect_supply_net** command (see 6.14). The predefined VCTs are shown in Annex B.

-hdl_type specifies the HDL type for which the value conversions are defined. This information allows a tool to provide completeness and compatibility checks. If the *typename* is not one of the language's predefined types or one of the types specified in the next paragraph, then it shall be of the form *library*.*pkg*.*type*.

The following HDL types shall be the minimum set of types supported. An implementation tool may support additional HDL types.

a) VHDL
   1) `Bit`, `std_[u]logic`, `Boolean`
   2) Subtypes of `std_[u]logic`
b) SystemVerilog

   `reg/wire`, `Bit`, `Logic`

-table defines the 1:1 conversions from UPF supply net states to an HDL logic value. The values shall be consistent with the HDL type values. For example:

— When converting to SystemVerilog *logic type*, the set of legal values is `0`, `1`, `X`, and `Z`.

— When converting to SystemVerilog or VHDL `bit`, the legal values are `0` or `1`.

— When converting to VHDL `std_[u]logic`, the legal values are `U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, and `-`.

The conversion values have no semantic meaning in UPF. The meaning of the conversion value is relevant to the HDL model to which the supply net is connected.

Syntax examples

```
create_upf2hdl_vct upf2vlog_vdd
  -hdl_type {sv}
  -table {{OFF X} {FULL_ON 1} {PARTIAL_ON 0}}
create_upf2hdl_vct upf2vhdl_vss
  -hdl_type {vhdl std_logic}
  -table {{OFF 'X'} {FULL_ON '1'} {PARTIAL_ON 'H'}}
```

6.28 describe_state_transition (deprecated)

This is a deprecated command; see also 6.2 and Annex D.

6.29 end_power_model

Purpose	Terminate the definition of a power model.
Syntax	**end_power_model**
Arguments	N/A
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.

The **begin_power_model** (see 6.11) and **end_power_model** commands define a power model containing other UPF commands. A power model is used to define the power intent of a model and shall be used in conjunction with one or more model representations. A power model defined with **begin_power_model** is terminated by the first subsequent occurrence of **end_power_model** in the same UPF file.

6.30 find_objects

6.30.1 General

Purpose	Find logic hierarchy objects within a scope.	
Syntax	**find_objects** *scope* **-pattern** *search_pattern* [**-object_type** <**model** | **inst** | **port** | **supply_port** | **net** | **process**>] [**-direction** <**in** | **out** | **inout**>] [**-transitive** [<**TRUE** | **FALSE**>]] [**-regexp** | **-exact**] [**-ignore_case**] [**-non_leaf** | **-leaf_only**]	
Arguments	*scope*	The search is restricted to the specified scope.
	-pattern *search_pattern*	The string used for searching. By default, *search_pattern* is treated as an Tcl `glob` expression.
	-object_type <**model** | **inst** | **port** | **supply_port** | **net** | **process**>	Limits the objects returned. By default, instances, named processes, ports, and nets are returned; this can be restricted by specifying a specific **-object_type**.
	-direction <**in** | **out** | **inout**>	If **-object_type** is **port**, then **-direction** can be used to restrict the directions of the returned ports.
	-transitive [<**TRUE** | **FALSE**>]	If **-transitive** is not specified at all, the default is **-transitive FALSE**. If **-transitive** is specified without a value, the default value is **TRUE**.
	-regexp | **-exact**	**-regexp** enables support for regular expression in the specified *search_pattern*. **-exact** disallows wildcard expansion on the specified *search_pattern*. If neither **-regexp** or **-exact** are specified, then *search_pattern* is interpreted as a Tcl `glob` expression.
	-ignore_case	Performs case-insensitive searches. By default, all matches are case sensitive.
	-non_leaf | **-leaf_only**	If **-non_leaf** is specified, only non-leaf objects are returned. If **-leaf_only** is specified, only leaf-level objects are returned. By default, both leaf and non-leaf objects are returned. This option is applicable to only **-object_type [model | instance | port]** and does not apply to **-object_type [net | process]**
Return value	Returns a list of names (relative to the current scope) of objects that match the search criteria; when nothing is found that matches the search criteria, a *null string* is returned. The list contains just the object names, without any indication of object type. The list may contain names of more than one type of object.	

The **find_objects** command searches for instances, nets, ports, supply ports, or processes that are defined in the logic hierarchy. If **-object_type** port is specified, **find_objects** searches the logic hierarchy for the logic ports whose port name matches the *search_pattern*. If **-object_type** supply_port is specified, **find_objects** searches the logic hierarchy for the supply ports (see 4.5.3) whose port name matches the *search_pattern*.

Only logic ports and supply ports visible at the time of **find_objects** execution shall be processed by the command (see 8.3.2). If **-object_type** is specified with any other value, **find_objects** searches the logic hierarchy for the specified objects whose name matches the *search_pattern*.

By default, or if **-transitive FALSE** is specified explicitly, **find_objects** searches only the specified scope of the logic hierarchy. If **-transitive TRUE** is specified, **find_objects** searches the specified scope and its entire descendant subtree. If **-transitive** is specified without an argument, it is equivalent to specifying **-transitive TRUE**. A transitive search will stop at a leaf cell boundary (see 4.9.2.3).

HighConn pins on a leaf cell instance are not deemed to be inside a leaf cell instance and can be returned by a search.

NOTE—The scope in find_objects can be set to any scope that set_scope in a given UPF can reach. However, find_objects is prohibited from initiating a search that starts in a lower scope that is a leaf cell or is below a leaf cell with respect to the current scope.

A UPF_is_hard_macro attribute value of **TRUE** on a model or a UPF_is_soft_macro attribute value of **TRUE** on an instance indicates that it shall be treated as a leaf cell (see 4.9.2.3) by find_objects.

The **-non_leaf** and **-leaf_only** options return the following depending on the specified **-object_type**:

— If **-object_type** is model: **-non_leaf** returns instances of models that correspond to non-leaf instances. **-leaf_only** returns instances of models corresponding to leaf instances.

— If **-object_type** is instance: **-non_leaf** returns non-leaf instances. **-leaf_only** returns leaf instances.

— If **-object_type** is port: **-non_leaf** returns ports of non-leaf instances. **-leaf_only** returns ports of leaf instances.

The following conditions also apply:

— The specified *scope* cannot start with `..` or `/`, i.e., **find_objects** shall be referenced from the current scope, and reside in the current scope or below it.

— If *scope* is specified as **.** (a dot), the current scope is used as the root of the search.

— All elements returned are referenced to the current scope.

— It shall be an error if *scope* is neither the current scope nor is defined in the current scope. The specified scope may reference a generate block as the root of the search.

— While **find_objects** commands are executed and their results are used; the command itself is not saved. However, this does not prohibit the use of **find_objects** in output UPF.

Syntax examples

```
find_objects A/B/D -pattern *BW1*
   -object_type inst
   -transitive TRUE
```

6.30.2 Pattern matching and wildcarding

To improve usability and allow multiple objects (instances, ports, etc.) to be easily specified without onerous verbosity, pattern matching (wildcarding) is allowed (only) in **find_objects** and **query_upf** (see 11.1). Pattern matching is supported using the Tcl `glob` style, matching against the symbols in the scope rather than filenames. For `glob`-style wildcarding, the following special operators are supported:

> **?** matches any single character.
>
> ***** matches any sequence of zero or more characters.
>
> **[***chars***]** matches any single character in *chars*. If *chars* contains a sequence of the form `a-b`, any character between `a` and `b` (inclusive) shall match.
>
> \x matches the character *x*.
>
> {*a*,*b*,*c*} Matches any string that is matched by any of the patterns *a*, *b*, or *c*.
>
> The "*****" and "**?**" never match a hierarchy separator "/".

Tcl regular expression matching is described in the Tcl documentation for `re_syntax` (see Tcl language syntax summary [B5]).

The use of the "/" to match the hierarchy separator is only allowed with "glob" type matching; it is not allowed with **-regexp**.

NOTE 1—Some characters used as operators in either glob-style or regular expression style *search_patterns*, such as [ ], \, and { }, also have meaning for Tcl in general. To ensure that such characters are not interpreted by the Tcl processor, the whole pattern can be enclosed in curly braces. This inhibits variable, command, and backslash substitution within the pattern by the Tcl processor (see 5.3.4).

NOTE 2—Square brackets used within a *search_pattern* are interpreted as indicating a set of characters, any of which matches a single character in a name. To use square brackets to refer to one or more bits of a bus, the square brackets must be escaped. For example, B\[3\] refers to B[3]. The two interpretations of square brackets can also be used in combination. For example, B\[[1-3]\] refers to B[1], B[2], and B[3].

6.30.3 Wildcarding examples

Table 5 shows the pattern match for each of the following examples of **find_objects**.

```
find_objects top -pattern a
find_objects top -pattern {bc[0-3]}
find_objects top -pattern e*
find_objects top -pattern d?f
find_objects top -pattern {g\[0\]}
find_objects top -pattern a/b*/c* -transitive FALSE
find_objects top -pattern a*/b/c* -transitive TRUE
```

Table 5—Pattern matches

`a`	Only matches an instance called `a` in the current scope.
`bc[0-3]`	Matches any instance called `bc` followed by a numerical value from `0` to `3`, i.e., `bc0`, `bc1`, `bc2`, and `bc3`.
`e*`	Matches any instance starting with `e`, i.e., `e12`, `eab`, `ef`, etc.
`d?f`	Matches any instance starting with `d` followed by another character and ending in `f`, i.e., `daf`, `d4f`, etc.
`g\[0\]`	Matches an instance called `g[0]`.
`a/b*/c*` `-transitive` `FALSE`	Matches any instance whose hierarchical name relative to the specified scope matches a/b*/c*. Equivalent to: `lsearch -all -inline -regexp \` `[find_objects top \` `-object_type inst \` `-pattern * \` `-transitive FALSE] \` `{^a/b[\w]*/c[\w]*$}`
`a*/b/c*` `-transitive` `TRUE`	Matches any instance whose hierarchical name relative to the specified scope or any descendant scope matches a*/b/c* Equivalent to: `lsearch -all -inline -regexp \` `[find_objects top \` `-object_type inst \` `-pattern * \` `-transitive TRUE] \` `{a[\w]*/b/c[\w]*$}`

In particular, to return individual bus bits, instead of a bus name, the *search_pattern* pattern shall explicitly contain escaped brackets \[and \]. For example, for a design with the following objects:

```
xyz1 .......... a single bit net
xyz2[3:0] ..... a four-bit bus
xyz[1:0] ...... a two-bit bus
```

Table 6 shows the return value for each of the following examples of **find_objects**.

```
find_objects top -pattern xyz*
find_objects top -pattern xyz
find_objects top -pattern {xyz*\[*\]}
find_objects top -pattern {xyz\[*\]}
find_objects top -pattern {xyz*\[0\]}
```

Table 6—Bus patterns and return values

`xyz*`	Returns bus/single-bit net names only: `xyz1 xyz2 xyz`
`xyz`	Returns the bus `xyz` only (no wild card)
`xyz*\[*\]`	Returns individual bus bits: `xyz2[3] xyz2[2] xyz2[1] xyz2[0] xyz[1] xyz[0]`
`xyz\[*\]`	Returns individual bus bits: `xyz[1] xyz[0]`
`xyz*\[0\]`	Returns individual bus bits: xyz2[0] xyz[0]

6.31 load_simstate_behavior

Purpose	Load the simstate behavior defaults for a library.	
Syntax	**load_simstate_behavior** *lib_name* **-file** *file_list*	
Arguments	*lib_name*	The tool-specific library name for which the simstate behavior file is to be loaded.
	-file *file_list*	The list of files containing the **set_simstate_behavior** commands.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

Loads a UPF file that only contains **set_simstate_behavior** commands and applies these to the models in the library *lib_name*.

It shall be an error if:

— *lib_name* cannot be resolved.

— *file_list* does not exist.

— a model specified in *file_list* cannot be found.

— the **set_simstate_behavior** commands in *file_list* use the **-lib** argument.

— *file_list* contains UPF commands other than **set_simstate_behavior.**

Syntax example

```
load_simstate_behavior library1 -file simstate_file.upf
```

6.32 load_upf

Purpose	Execute commands from the specified UPF file in the current scope or in the scope of each specified instance.	
Syntax	**load_upf** *upf_file_name* [**-scope** *instance_name_list*] [**-hide_globals**] [**-parameters** {{*parameter_name* [*parameter_value*]}*}]	
Arguments	*upf_file_name*	The UPF file to execute.
	-scope *instance_name_list*	The list of instances where the UPF commands contained in *upf_file_name* are executed.
	-hide_globals	Enable global tcl variables to be unmodified by **load_upf**, unless the global variable is passed in parameters.
	-parameters {{*parameter_name* [*parameter_value*] }*}	A list of formal arguments to the load_upf command.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **load_upf** command executes the commands in the specified UPF file. Commands are executed either in the current scope or in the scope of each of the specified instances.

If **-scope** is not specified, **load_upf** executes the commands in the current scope. In this case, the current scope, design top instance, and design top module are not affected.

If **-scope** is specified, each instance name in the instance name list shall be a simple name or a hierarchical name rooted in the current scope. In this case, **load_upf** executes the commands in the scope of *each instance*, as follows:

a) The current scope and design top instance are both set to the instance, and the design top module is set to the module type of that instance;

b) The commands in the specified UPF file are then executed in the scope of the instance;

c) The current scope, design top instance, and design top module then revert to their previous values.

If an instance name specified in *instance_name_list* is not found, further processing of remaining instance names in the *instance_name_list* is terminated and a `TCL_ERROR` is raised.

-hide_globals is useful to suppress modifications to global variables when **load_upf** is loaded in a global namespace. If global variables are explicitly passed in **-parameter**, then the globals can be modified as a result of the **load_upf**. Unless the global variables/procs are explicitly accessed in the load upf (using the global :: scope operator), all variables and procs defined in the global namespace shall be invisible inside the loaded upf.

-parameters is the arguments list to load_upf and follows the syntax of tcl args. If any list item itself contains two items, the second item becomes the default value for that argument. When **load_upf** is invoked, each actual argument shall be stored in the variable named by the formal argument. After the first default value to a formal argument is encountered, all additional formal arguments must have default values.

load_upf does not create a new name space for the loaded UPF file. The loaded UPF file is responsible for ensuring the integrity of both its own and the caller's name space as needed using existing Tcl name space management capabilities.

The following also apply:

— It shall be an error if **load_upf -scope** is used more than once to load a UPF file for a given instance.

— It shall be an error if **load_upf -scope** is used to load a UPF file for an instance and **apply_power_model** is also used to apply a power model to the same instance.

NOTE—The **load_upf** command only has the same effect as the Tcl *source* command when **load_upf** is used without the **-scope** option. When **-scope** is used, an implicit context switch occurs (which changes current scope, design top instance, and design top module); this would not occur with the Tcl *source* command.

Syntax example

```
load_upf my.upf -scope {I1/I2 I3/I2}
load_upf design.upf -scope inst_a/inst_b/inst_design
-parameters {{N 64} {num_of_cores 4}}
```

6.33 load_upf_protected (deprecated)

This is a deprecated command; see also 6.2 and Annex D.

6.34 map_power_switch

Purpose	Specify which power-switch model is to be used for the implementation of the corresponding switch instance.	
Syntax	**map_power_switch** *switch_name_list* **-lib_cells** *lib_cell_list* [**-port_map** **{{***mapped_model_port switch_port_or_supply_net_ref***}***}]	
Arguments	*switch_name_list*	A list of switches [as defined by **create_power_switch** (see 6.21)] to map.
	-lib_cells *lib_cell_list*	A list of library cells.
	-port_map **{{***mapped_model_port switch_port_or_supply_net_ref***} *}**	*mapped_model_port* is a port on the model being mapped. *switch_port_or_supply_net_ref* indicates a supply or logic port on a switch: an input supply port, output supply port, control port, or acknowledge port; or it references a supply net from a supply set associated with the switch. See also **create_power_switch** (6.21).
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **map_power_switch** command can be used to explicitly specify which power-switch model is to be used for the corresponding switch instance.

-lib_cells specifies the set of library cells to which an implementation can be mapped. Each cell specified in **-lib_cells** shall be defined by a **define_power_switch_cell** command (see 7.6) or defined in the Liberty file with required attributes.

If **-port_map** is not specified, the ports of the switch instance are associated to library cell ports by matching the respective port names, this is *named association*. It shall be an error if any ports on either the switch instance or the library cell are not mapped when named association is used.

It shall be an error if *switch_name_list* is an empty list.

NOTE—All **map_*** commands specify the elements to be used rather than inferred through a strategy. The behavior of this manual mapping may lead to an implementation that is different from the RTL specification. Therefore, logical equivalence checking tools may not be able to verify the equivalence of the mapped element to its RTL specification.

Syntax example

```
map_power_switch switch_sw1
-domain test_suite
-lib_cells {sw1}
-port_map {{inp1 vin1} {inp2 vin2} {outp vout}
   {c1 ctrl_small} {c2 ctrl_large}}
```

6.35 map_repeater_cell

Purpose	Specify a list of implementation targets for repeaters.	
Syntax	**map_repeater_cell** *repeater_strategy_name* **-domain** *domain_name* [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-lib_cells** *lib_cell_list*]	
Arguments	*repeater_strategy_name*	The repeater strategy as defined by **set_repeater** command (see 6.48).
	-domain *domain_name*	The domain for which the set_repeater strategy is defined.
	-elements *element_list*	A list of ports from the repeater_strategy_name to which the command applies.
	-exclude_elements *exclude_list*	A list of ports from the repeater_strategy_name to which this command does not apply.
	-lib_cells *lib_cell_list*	A list of library cell names.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **map_repeater_cell** command provides user control for specifying implementation choices for the **set_repeater** strategy through **-lib_cells** option.

Both single-rail and dual-rail repeaters as well as any custom repeater can be specified in the *lib_cell_list.*

-elements identifies elements from the *effective_element_list* (see 5.9) from the repeater strategy *repeater_strategy_name*. If **-elements** is not specified, the *aggregate_element_list* for this command contains all the elements from the *effective_element_list* of the *repeater_strategy_name*.

It shall be an error if:

— *domain_name* does not indicate a previously created power domain.

— *repeater_strategy_name* does not exist.

— *element_list* is empty.

Syntax example

```
map_repeater_cell my_rep1_pd1 -domain PD1
-elements  { clk1 rst1 clkout1 rstout1 }
-lib_cells { aon_clk_bufx2 }
```

6.36 map_retention_cell

Purpose	Constrain implementation alternatives, or specify a functional model, for retention strategies.	

Syntax	**map_retention_cell** *retention_name_list*
	    -domain *domain_name*
	    [**-elements** *element_list*]
	    [**-exclude_elements** *exclude_list*]
	    [**-lib_cells** *lib_cell_list*]
	    [**-lib_cell_type** *lib_cell_type*]
	    [**-lib_model_name** *name* **-port_map** **{{***port_name net_ref***}** ***}**]	

Arguments	*retention_name_list*	A list of target retention strategy names defined in *domain_name* using **set_retention** commands (see 6.49).
	-domain *domain_name*	The domain in which the strategies are defined.
	-elements *element_list*	A list of instances, named processes, state elements, or signal names whose respective sequential elements shall be mapped as specified.
	-exclude_elements *exclude_list*	A list of instances, named processes, or state elements or signal names whose respective sequential elements shall be excluded from mapping.
	-lib_cells *lib_cell_list*	A list of library cell names. Each cell in the list has retention behavior and is otherwise identical to the inferred RTL behavior of the underlying sequential element.
	-lib_cell_type *lib_cell_type*	The attribute of the library cells used to identify cells that have retention behavior and are otherwise identical to the inferred RTL behavior of the underlying sequential element.
	-lib_model_name *model_name* **-port_map** **{{***port_name net_ref***}** ***}**	The name of the library cell or behavioral model and associated port connectivity.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **map_retention_cell** command constrains retention strategy implementation choices and may also specify functional retention behavior for verification.

-elements identifies state elements in the specified domain for which retention registers have been inferred from the *effective_element_list* (see 5.9) from a retention strategy in *retention_name_list*. If **-elements** is not specified, the *aggregate_element_list* for this command contains all state elements in the specified domain for which retention registers have been inferred from some strategy in the *retention_name_list*.

It shall be an error if at least one of **-lib_cells**, **-lib_cell_type**, or **-lib_model_name** is not specified.

- If **-lib_cells** is specified, each cell shall be either defined by the **define_retention_cell** command (see 7.7) or defined in the Liberty file with required attributes; If **-lib_cells** is specified, a retention cell from *lib_cell_list* shall be used; if **-lib_cell_type** is specified, a retention cell with the same type string specified by **define_retention_cell -cell_type** shall be used to implement the functionality specified by the corresponding retention strategy; if **-lib_cells** and **-lib_cell_type** are both specified, a retention cell from *lib_cell_list* that is also defined with the same type string in **define_retention_cell -cell_type** shall be used. Verification semantics are unchanged by the presence or absence of **-lib_cells** or **-lib_cell_type**.
- If **-lib_model_name** is specified, *model_name* shall be used as the verification model, and supply and logic ports shall be connected as specified by **-port_map** options; automatic corruption and retention verification semantics do not apply to a **-lib_model_name** model.
- If **-lib_model_name** is not specified, the verification semantic is that of the inferred RTL behavior of the underlying sequential element modified by the retention behavior prescribed by the applicable **set_retention** strategy.

Table 7 summarizes the semantics for combinations of **-lib_cells**, **-lib_cell_type**, and **-lib_model_name**.

Table 7—map_retention_cell option combinations

-lib_cells	-lib_cell_type	-lib_model_name	Verification semantic	Implementation cell constrained to
N	N	N	ERROR	ERROR
N	N	Y	*model_name*	*model_name*
N	Y	N	RTL with retention	*lib_cell_type*
N	Y	Y	*model_name*	*lib_cell_type*
Y	N	N	RTL with retention	*lib_cell_list*
Y	N	Y	*model_name*	*lib_cell_list*
Y	Y	N	RTL with retention	A cell from *lib_cell_list* that also has *lib_cell_type*
Y	Y	Y	*model_name*	A cell from *lib_cell_list* that also has *lib_cell_type*

For verification, an inferred register is assumed to have the following generic canonical interface:

- **CLOCK**—The signal whose rising edge triggers the register to load data.
- **DATA**—The signal whose value represents the next state of the register.
- **ASYNC_LOAD**—The signal that causes the register to load data when its value is one (`1`).
- **OUTPUT**—The signal that propagates the register output to the receivers of the register.

-port_map connects the specified *net_ref* to a *port* of the model. A *net_ref* may be one of the following:

a) A logic net name

b) A supply net name

c) One of the following symbolic references

  1) **retention_supply**.*function_name*

     This names a retention supply set function, where *function_name* refers to the supply net corresponding to the function it provides to the retention supply set of the retention cell (see 6.49).

  2) **primary_supply**.*function_name*

     This names a primary supply set function, where *function_name* refers to the supply net corresponding to the function it provides to the primary supply set of the domain.

  3) **save_signal**

     i) Refers to the save signal specified in the corresponding retention strategy.

     ii) To invert the sense of the save signal, the SystemVerilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly powered by the retention supply set of the retention cell (see 6.49).

  4) **restore_signal**

     i) Refers to the restore signal specified in the corresponding retention strategy.

     ii) To invert the sense of the restore signal, the SystemVerilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly powered by the retention supply set of the retention cell (see 6.49).

  5) **UPF_GENERIC_CLOCK**

     i) Refers to the canonical **CLOCK**.

     ii) To invert the sense of the clock signal, the SystemVerilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly powered by the primary supply set of the domain.

  6) **UPF_GENERIC_DATA**

     i) Refers to the canonical **DATA**.

     ii) To invert the sense of the data signal, the SystemVerilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly powered by the primary supply set of the domain.

  7) **UPF_GENERIC_ASYNC_LOAD**

     i) Refers to the canonical **ASYNC_LOAD**.

     ii) To invert the sense of the asynchronous load signal, the SystemVerilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be powered by the primary supply set of the domain.

  8) **UPF_GENERIC_OUTPUT**

     i) Refers to the canonical **OUTPUT**.

ii) To invert the sense of the output signal, the SystemVerilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly powered by the primary supply set of the domain.

If **UPF_GENERIC_OUTPUT** is not explicitly mapped and the model has exactly one output port, that output port shall automatically be connected to the net that propagates the register output to the receivers of the register.

NOTE—All **map_*** commands specify the elements to be used rather than inferred through a strategy. The behavior of this manual mapping may lead to an implementation that is different from the RTL specification. Therefore, it may not be possible for logical equivalence checking tools to verify the equivalence of the mapped element to its RTL specification.

It shall be an error if:

— *retention_name_list* is an empty list.
— *domain_name* does not indicate a previously created power domain.
— A retention strategy in *retention_name_list* does not indicate a previously defined retention strategy.
— An element in *element_list* is not included in the element list of a targeted retention strategy.
— Any retention strategy in *retention_name_list* does not specify signals needed to provide connection of the mapped functions.
— After completing the *port* and *net_ref* connections, any input port is unconnected, or no output port is connected to the net that propagates the register output to the receivers of the register.
— In implementation, none of the specified models in *lib_cell_list* implements the functionality specified by a targeted retention strategy.
— In implementation, none of the specified models having a *lib_cell_type* attribute implements the functionality specified by a targeted retention strategy.
— In implementation, none of the specified models in *lib_cell_list* that have a *lib_cell_type* attribute, when both are specified, implements the functionality specified by a targeted retention strategy.

Syntax example

```
map_retention_cell {my_PDA_ret_strat_1 my_PDA_ret_strat_2
   my_PDA_ret_strat_3}
   -domain PowerDomainA
   -elements {foo/U1 foo/U2}
   -lib_cells {RETFFIMP1 RETFFIMP2}
   -lib_cell_type FF_CKLO
   -lib_model_name RETFFVER -port_map {
      {CP       UPF_GENERIC_CLOCK}
      {D        UPF_GENERIC_DATA}
      {SET      UPF_GENERIC_ASYNC_LOAD}
      {SAVE     save_signal}
      {RESTORE  restore_signal}
      {VDDC     primary_supply.power}
      {VDDRET   retention_supply.power}
      {VSS      primary_supply.ground} }
```

6.37 name_format

Purpose	Define the format for constructing names of implicitly created objects.	
Syntax	**name_format** [**-isolation_prefix** *pattern*] [**-isolation_suffix** *pattern*] [**-level_shift_prefix** *pattern*] [**-level_shift_suffix** *pattern*] [**-implicit_supply_suffix** *string*] [**-implicit_logic_prefix** *string*] [**-implicit_logic_suffix** *string*]	
Arguments	**-isolation_prefix** *pattern*	The pattern used to construct a string that is prepended in front of an existing signal or port name to create a new name used during the introduction of a new isolation cell. The default value is the empty string `""` or `NULL`.
	-isolation_suffix *pattern*	The pattern used to construct a string that is appended to the end of an existing signal or port name to create a new name used during the introduction of a new isolation cell. The default value is the string `_UPF_ISO`.
	-level_shift_prefix *pattern*	The pattern used to construct a string that is prepended in front of an existing signal or port name to create a new name used during the introduction of a new level-shifter cell. The default value is the empty string `""` or `NULL`.
	-level_shift_suffix *pattern*	The pattern used to construct a string that is appended to the end of an existing signal or port name to create a new name used during the introduction of a new level-shifter cell. The default value is the string `_UPF_LS`.
	-implicit_supply_suffix *string*	The string appended to an existing supply net or port name to create a unique name for an implicitly created supply net or port. The default value is the string `_UPF_IS`.
	-implicit_logic_prefix *string*	The string prepended in front of an existing logic net or port name to create a unique name for an implicitly created logic net or port. The default value is `NULL`.
	-implicit_logic_suffix *string*	The string appended to an existing logic net or port name to create a unique name for an implicitly created logic net or port. The default value is the string `_UPF_IL`.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

Inferred objects have names in the logic design. The name for these objects is constructed as follows:

a) The base name of implicitly created objects is the name of the port or net being isolated or level-shifted, or the supply net, logic net, or port implicitly created to facilitate the connection of a net across hierarchy boundaries.

b) Any specified prefix is then prepended to the base name.

c) Any specified suffix is also appended to the base name.

d) If multiple prefixes or suffixes apply to the same object, they shall be added in the alphabetical order of the option name, e.g., **isolation_prefix** followed by **level_shift_prefix**.

When a *pattern* has been specified to be used for a prefix or suffix, then this pattern shall be used to generate the string. The pattern consists of a string with optional use of reserved variables that are evaluated at the time of cell insertion.

Variable	**Value**
`%d`	domain of the strategy inserting the cell
`%s`	name of the strategy inserting the cell
`%i`	instance index

If the generated name conflicts with another previously defined name in the same name space, the generated name is updated to include an instance index which consists of an underscore (_) followed by a positive integer. The value of the integer is the smallest number that makes the name unique in its name space. In a pattern the `%i` reserved variable shall be substituted for this instance index, otherwise it shall be added to the end of the suffix. An empty string (`""`) is a valid value for any prefix or suffix option.

Different prefixes and suffixes may be specified in multiple calls to **name_format** (using different arguments). When **name_format** is specified with no options, the name format is reset to the default values.

It shall be an error to specify an affix more than once.

Syntax example

```
  name_format -isolation_prefix "MY_ISO_" -isolation_suffix ""
```

A signal, `MY_ISO_FOO`, is created and connected to a new cell's output (to isolate the existing net `FOO`).

```
name_format -level_shift_prefix "shift_%d_%s_" -level_shift_suffix "%i_UPF_LS"
```

For a strategy_name `LS_IN`, defined for domain `PD`, a signal named `shift_PD_LS_IN_FOO_UPF_LS` is created and connected to a new cell's output (to shift the existing net `FOO`). Alternatively if there is a name conflict then the signal may use an index, e.g., `shift_PD_LS_IN_FOO_1_UPF_LS`.

6.38 save_upf

Purpose	Create a UPF file of the structures relative to the active or specified scope.	
Syntax	**save_upf** *upf_file_name* [**-scope** *instance_name*]	
Arguments	*upf_file_name*	The UPF file to write.
	-scope *instance_name*	The scope relative to which the UPF commands are written.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **save_upf** command creates a UPF file that contains the power intent specified for a given scope. The power intent for that scope is written to file *upf_file_name*. The output file is generated after the power intent model has been constructed (see 8.3.3).

If **-scope** *instance_name* is specified, the power intent is written for the specified scope. It shall be an error if this scope does not exist. Otherwise, the power intent is written for the current scope.

The following also apply:

a) Each invocation of **save_upf** generates a separate UPF output file.

b) If **save_upf** is invoked for two scopes and one is an ancestor of the other, then the file generated for the ancestor shall contain a duplicate of the information in the file generated for the other.

c) The following are equivalent:

```
save_upf <filename> -scope <instance>
```

and

```
set temp [set_scope <instance>]
save_upf <filename>
set_scope $temp
```

Syntax example

```
save_upf test_suite1_Jan14
-scope top/proc_1
```

6.39 set_correlated

Purpose	To declare that supply nets' or sets' voltage variation ranges are to be treated as correlated when being compared; min to min and max to max.	
Syntax	**set_correlated** [**-nets** **{{***supply_net_name_list***}***}**] [**-sets** **{{***supply_set_name_list***}***}**]	
Arguments	**-nets** **{{***supply_net_name_list***}***}**	A list of sublists with each sublist declaring which nets to declare as correlated.
	-sets **{{***supply_set_name_list***}***}**	A list of sublists with each sublist declaring which sets to declare as correlated.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **set_correlated** command declares that two or more supply ports, supply nets, or supply set functions are correlated (see 4.5.8).

If **-nets** is specified, the command defines correlation for the specified supply ports, supply nets, and/or supply set functions. If **-sets** is specified, the command defines correlation for the corresponding power functions and for the corresponding ground functions of the specified supply sets. One or the other of these options, but not both, shall be specified.

Supplies declared as equivalent (see 6.43) are always deemed to be correlated. However correlating supplies using set_correlated does not declare the supplies to also be equivalent.

Syntax example

```
set_correlated -nets {{VDD1 VDD2}}
set_correlated -nets {{VDD1 VDD2} {VDD3 VDD4}}
set_correlated -sets {{SS1 SS2}}
```

6.40 set_design_attributes

Purpose	Apply attributes to models or instances.	
Syntax	**set_design_attributes** [**-models** *model_list*] [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-attribute** {*name value*}]* [**-is_soft_macro** [<**TRUE** | **FALSE**>]] [**-is_hard_macro** [<**TRUE** | **FALSE**>]] [**-switch_cell_type** <**coarse_grain** | **fine_grain**>]	
Arguments	**-models** *model_list*	A list of models to be attributed.
	-elements *element_list*	A list of rooted names: instances, named processes, state elements, or signal names.
	-exclude_elements *exclude_list*	A list of rooted names: instances, named processes, state elements, or signal names to exclude from the *effective_element_list* (see 5.9).
	-attribute {*name value*}	For the specified models or elements, associate the attribute *name* with the value of *value*. See Table 4.
	-is_soft_macro [<**TRUE** | **FALSE**>]	If **-is_soft_macro** is not specified at all, the default is **FALSE**. If **-is_soft_macro** is specified without a value, the default value is **TRUE**. Equivalent to **-attribute {UPF_is_soft_macro** *value*} (see 5.6).
	-is_hard_macro [<**TRUE** | **FALSE**>]	If **-is_hard_macro** is not specified at all, the default is **FALSE**. If **-is_hard_macro** is specified without a value, the default value is **TRUE**. Equivalent to **-attribute {UPF_is_hard_macro** *value*} (see 5.6).
	-switch_cell_type <**coarse_grain** | **fine_grain**>	If specified, identifies the switch cell type of the model. Equivalent to **-attribute** {**UPF_switch_cell_type** *value*} (see 5.6).
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **set_design_attributes** command sets the specified attributes for models or elements. Models are referenced using **-models**; instances are referenced using **-elements**. If **-models** is specified and the *model_list* is **.** (a dot), the command applies to the model corresponding to the current scope.

Certain predefined attributes identify hard and soft macros. Other predefined attributes provide information about power switches, simulation semantics, or retention requirements. Predefined attributes **UPF_is_hard_macro**, **UPF_is_soft_macro**, **UPF_switch_cell_type**, and **UPF_simstate_behavior** can only be specified for models; predefined attribute **UPF_retention** can only be defined for instances, named processes, state elements, or signal names.

User-defined attributes may also be associated with a model or instance. The meaning of a user-defined attribute is not specified by this standard. User-defined attributes can be specified for either a model or an instance, or both.

If **-models** is specified, the command associates one or more attributes with each model in the *model_list*.

If **-elements** is specified, the command associates one or more attributes with each instance in the *element_list* that is not also in the *exclude_list*.

set_design_attributes -models can be specified in the topmost scope of a given model to define attributes of ports of that model. In this case, the specification applies to all instances of the model in any design or soft macro in which it is instantiated.

set_design_attributes -models can also be specified in a scope that is outside any of the models named in the model list to define an attribute of a model if that attribute is not already defined for that model. In this case, the specification applies to all instances of the model that are instantiated in the design or soft macro in which the attribute is specified, from the design top scope down to, but not including, the leaf cell instances of the design or soft macro. It shall be an error if an attribute of a given model is defined more than once with different values within a design or a soft macro.

-is_hard_macro defines the **UPF_is_hard_macro** attribute for the specified model(s). If the attribute **UPF_is_hard_macro** TRUE is associated with a model, then any instance of that model is considered to be a hard macro instance (see 4.9.2.4.2). This can affect whether a port of the hard macro instance is on a power domain boundary (see 6.20).

-is_soft_macro defines the **UPF_is_soft_macro** attribute for the specified model(s). If the attribute **UPF_is_soft_macro** TRUE is associated with a model, then any instance of that model is considered to be a soft macro instance (see 4.9.2.4.3). This creates a terminal boundary between the macro instance and its parent context such that the power intent of soft macro is not affected by the power intent of the parent context, and vice versa.

-switch_cell_type defines the **UPF_switch_cell_type** attribute for the specified model(s). This attribute affects the selection of switch type in **create_power_switch** (see 6.21).

-attribute can be used to define user-defined attributes or predefined attributes **UPF_retention** (see 6.50) and **UPF_simstate_behavior** (see 6.52).

It shall be an error if **set_design_attributes** is specified:

a) with neither **-models** nor **-elements**; or
b) with both **-models** and **-elements**; or
c) with **-exclude_elements**, but not **-elements**; or
d) without specifying at least one attribute.

Examples

```
set_design_attributes -models {lock_cache}
   -attribute {UPF_is_soft_macro TRUE}
set_design_attributes -models FIFO
   -attribute {UPF_is_hard_macro TRUE}
set_design_attributes -models -is_hard_macro
```

6.41 set_design_top

Purpose	Specify the design top module.
Syntax	**set_design_top** *design_top_module*
Arguments	*design_top_module* The top module for which a UPF file was written.
Return value	Return an empty string.

The **set_design_top** command specifies the module for which this UPF file was written. See 4.3.7.

It is not an error if the instance to which this UPF file is applied is not an instance of the specified module. In particular, as long as the actual module has the same structure as the specified module, it may be possible to apply this UPF file to that module without errors. In this case, a tool may choose to issue a warning message.

Syntax example

```
set_design_top ALU07
```

6.42 set_domain_supply_net (legacy)

Purpose	Set the default power and ground supply nets for a power domain.	
Syntax	**set_domain_supply_net** *domain_name* **-primary_power_net** *supply_net_name* **-primary_ground_net** *supply_net_name*	
Arguments	*domain_name*	The domain where the default supply nets are applied.
	-primary_power_net *supply_net_name*	The primary power supply net.
	-primary_ground_net *supply_net_name*	The primary ground net.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

This is a legacy command; see also 6.2 and Annex D.

The **set_domain_supply_net** command associates the power and ground supply nets with the primary supply set for the domain.

The primary supply set's power and ground functions for the specified domain are associated with the corresponding power and ground supply net.

It shall be an error if:

— *domain_name* does not indicate a previously created power domain.

— The primary supply set for *domain_name* already has a primary power or ground function association.

This command is semantically equivalent to

```
proc set_domain_supply_net {dn pp sn1 pg sn2} {
   if { string equal $pp "-primary_power_net" \
      && string equal $pg "-primary_ground_net"}{
      create_supply_set set_name -function {power $sn1}
             -function {ground $sn2}
      associate_supply_set set_name -handle $dn.primary
      return 1
   } else {
       return -code TCL_ERROR \
           -errorcode $ecode \
           -errorinfo $einfo \
           $resulttext
   } }
```

where any *italicized* arguments are implementation defined.

Syntax example

```
set_domain_supply_net PD1
-primary_power_net PG1
-primary_ground_net PG0
```

6.43 set_equivalent

Purpose	Declare that supply nets or supply sets are electrically or functionally equivalent.	
Syntax	**set_equivalent** [**-function_only**] [**-nets** *supply_net_name_list*] [**-sets** *supply_set_name_list*]	
Arguments	**-function_only**	Specifies that the supplies are functionally equivalent rather than electrically equivalent.
	-nets *supply_net_name_list*	A list of supply port and/or supply net names that are equivalent.
	-sets *supply_set_name_list*	A list of supply set names that are equivalent.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **set_equivalent** command declares that two or more supplies are *equivalent* (see 4.5.5).

If **-function_only** is specified, then the supplies are declared to be *functionally equivalent* only; otherwise the supplies are declared to be *electrically equivalent*, which implies that they are also functionally equivalent.

If **-nets** is specified, the command defines equivalence for a list of supply ports and/or supply nets. If **-sets** is specified, the command defines equivalence for a list of supply sets and/or supply set handles. One or the other of these options, but not both, shall be specified.

Equivalence of supply ports and nets can affect the number of sources for a given supply network and whether resolution is required (see 9.1). Equivalence of supply sets and supply set handles can affect various commands whose semantics are based on supply set identity or equivalence, including **create_composite_domain** (see 6.16), **create_power_domain** (see 6.20), **set_isolation** (see 6.44), **set_level_shifter** (see 6.45), **set_repeater** (see 6.48), and **set_port_attributes** (see 6.47).

The declaration of a supply equivalence is a user constraint. It shall be an error if other information in the HDL/UPF contradict the equivalence.

If the actual connections implementing electrical equivalence cannot be found in the HDL/UPF, then they must be present outside the design.

Syntax example

```
set_equivalent -nets { vss1 vss2 ground }
set_equivalent -function_only -nets { vdd_wall vdd_battery }
set_equivalent -function_only -sets { /sys/aon_ss mem/PD1.core_ssh }
```

6.44 set_isolation

Purpose	Specify an isolation strategy.		
Syntax	**set_isolation** *strategy_name* **-domain** *domain_name* [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-source** <*source_domain_name* | *source_supply_ref*>] [**-sink** <*sink_domain_name* | *sink_supply_ref*>] [**-diff_supply_only** [<**TRUE** | **FALSE**>]] [**-use_equivalence** [<**TRUE** | **FALSE**>]] [**-applies_to** <**inputs** | **outputs** | **both**>] [**-applies_to_boundary** <**lower** | **upper** | **both**>] [**-applies_to_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-applies_to_sink_off_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-applies_to_source_off_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-no_isolation**] [**-force_isolation**] [**-location** <**self** | **other** | **parent** | **fanout**>] [**-clamp_value** <**0** | **1** | **Z** | **latch** | *value* | **{**<**0** | **1** | **Z** | **latch** | *value*>***}**>] [**-isolation_signal** *signal_list* [**-isolation_sense** <**high** | **low** | **{**<**high** | **low**>***}**>]] [**-isolation_supply** *supply_set_list*] [**-name_prefix** *pattern*] [**-name_suffix** *pattern*] [**-instance** **{{***instance_name port_name***}*****}**] [**-update**]		
Arguments	*strategy_name*	The name of the isolation strategy.	
	-domain *domain_name*	The domain for which this strategy is defined.	
	-elements *element_list*	A list of instances or ports to which the strategy potentially applies.	**R**
	-exclude_elements *exclude_list*	A list of instances or ports to which the strategy does not apply.	**R**
	-source <*source_domain_name* | *source_supply_ref*>	The name of a supply set or power domain. When a domain name is used, it represents the primary supply of that domain.	**R**
	-sink <*sink_domain_name* | *sink_supply_ref*>	The name of a supply set or power domain. When a domain name is used, it represents the primary supply of that domain.	**R**
	-diff_supply_only [<**TRUE** | **FALSE**>]	Indicates whether ports connected to other ports with the same supply should be isolated. The default is **-diff_supply_only TRUE** if the option is not specified at all; if **-diff_supply_only** is specified without a value, the default value is **TRUE**.	**R**
	-use_equivalence [<**TRUE** | **FALSE**>]	Indicates whether to consider supply set equivalence. If **-use_equivalence** is not specified at all, the default is **-use_equivalence TRUE**; if **-use_equivalence** is specified without a value, the default value is **TRUE**.	**R**
	-applies_to <**inputs** | **outputs** | **both**>	A filter that restricts the strategy to apply only to ports of a given direction.	**R**
	-applies_to_boundary <**lower** | **upper** | **both**>	Restricts the application of filters to specified boundary. Default is both.	**R**
	-applies_to_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	A filter that restricts the strategy to apply only to ports with a particular clamp value requirement.	**R**

	-applies_to_sink_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	A filter that restricts the strategy to apply only to ports with a particular sink off clamp value requirement.	**R**
	-applies_to_source_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	A filter that restricts the strategy to apply only to ports with a particular source off clamp value requirement.	**R**
	-no_isolation	Specifies that isolation cells shall not be inserted on the specified ports.	**R**
	-force_isolation	Disables any implementation optimization involving isolation cells for a given strategy; used to force redundant isolation or to keep floating/constant ports that have an isolation strategy defined for them.	**R**
	-location <**self** | **other** | **parent** | **fanout**>	The location in which inferred isolation cells are placed in the logic hierarchy, which determines the power domain in which they shall be inserted. The default is **self**.	**R**
	-clamp_value <**0** | **1** | **Z** | **latch** | *value* | **{**<**0** | **1** | **Z** | **latch** | *value*>***}**>	The value(s) that the isolation cell can drive.	**R**
	-isolation_signal *signal_list* [**-isolation_sense** <**high** | **low** | **{**<**high** | **low**>***}**>]	The isolation control signal for the isolation cell.	**R**
	-isolation_sense **{**<**high** | **low**>***}**	The active level of the isolation control signal for the isolation cell. The default is **high**.	**R**
	-isolation_supply *supply_set_list*	The supply set that powers the isolation cell.	**R**
	-name_prefix *pattern* **-name_suffix** *pattern*	The name format (prefix and suffix) for generated isolation instances or nets related to implementation of the isolation strategy.	**R**
	-instance **{{***instance_name port_name***}*****}**	The name of a technology leaf cell instance and the name of the logic port that it isolates.	**R**
	-update	Indicates that this command provides additional information for a previous command with the same *strategy_name* and *domain_name* and executed in the same scope.	**R**
Legacy arguments	**-isolation_power_net** *net_name*	This option specifies the supply net used as the power for the isolation logic inferred by this strategy. This is a legacy option; see also 6.2 and Annex D.	**R**
	-isolation_ground_net *net_name*	This option specifies the supply net used as the ground for the isolation logic inferred by this strategy. This is a legacy option; see also 6.2 and Annex D.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

The **set_isolation** command defines an isolation strategy for ports on the interface of a power domain (see 6.20). An isolation strategy is applied at the domain boundary, as required, so that correct electrical and logical functionality is maintained when domains are in different power states.

-domain specifies the domain for which this strategy is defined.

-elements explicitly identifies a set of candidate ports to which this strategy potentially applies. The *element_list* may contain rooted names of instances or ports in the specified domain. If an instance name is specified in the *element_list*, it is equivalent to specifying all the ports of the instance in the *element_list*, but with lower precedence (see 5.7). Any *element_list*s specified on the base command and any *elements_list*s specified in any updates (see **-update**) of the base command are all combined into a single elements list. If **-elements** is not specified in the base command or any update, every port on the interface of the domain is included in the *aggregate_element_list* (see 5.9).

-exclude_elements explicitly identifies a set of ports to which this strategy does not apply. The *exclude_list* may contain rooted names of instances or ports in the specified domain. If an instance name is specified in the *exclude_list*, it is equivalent to specifying all the ports of the instance in the *exclude_list*. Any *exclude_list*s specified on the base command or any updates of the base command are combined into the *aggregate_exclude_list* (see 5.9).

The arguments **-source**, **-sink**, **-diff_supply_only**, **-applies_to**, **-applies_to_clamp**, **-applies_to_sink_off _clamp**, and **-applies_to_source_off_clamp** serve as filters that further restrict the set of ports to which a given **set_isolation** command applies. The command only applies to those ports that satisfy all of the specified filters.

The **-source** option specifies the simple name, rooted name, or design-relative hierarchical name (see 5.3.3.4) of a power domain or supply set. **-source** is satisfied by any port that is driven by logic powered by a supply set that matches (see **-use_equivalence**) the specified supply set, ignoring any isolation or level-shifting cells that have already been inferred or instantiated from an isolation or level-shifting strategy.

The **-sink** option specifies the simple name, rooted name, or design-relative hierarchical name (see 5.3.3.4) of a power domain or supply set. **-sink** is satisfied by any port that is received by logic powered by a supply set that matches (see **-use_equivalence**) the specified supply set, ignoring any isolation or level-shifting cells that have already been inferred or instantiated from an isolation or level-shifting strategy.

NOTE—A port that does not have a driver will never satisfy the **-source** filter. A port that does not have a receiver will never satisfy the **-sink** filter.

-diff_supply_only TRUE is satisfied by any port for which the driving logic and receiving logic are powered by supply sets that do not match (see **-use_equivalence**), or for which either driving or receiving or both supply sets cannot be determined. **-diff_supply_only FALSE** is satisfied by any port.

-use_equivalence specifies whether supply set equivalence is to be considered in determining when two supply sets match. If **-use_equivalence** is specified with the value *False*, the **-source** and **-sink** filters shall match only the named supply set; the **-diff_supply_only TRUE** filter shall be satisfied only if the driver supply and receiver supply of the port are not identical. Otherwise, the **-source** and **-sink** filters shall match the named supply set or any supply set that is equivalent to the named supply set; the **-diff_supply_only TRUE** filter shall be satisfied only if the driver supply and receiver supply of the port are neither identical nor equivalent.

-applies_to is satisfied by any port that has the specified mode. For upper boundary ports, this filter is satisfied when the direction of the port matches. For lower boundary ports, this filter is satisfied when the inverse of the direction of the port matches. For example, a lower boundary port with a direction `OUT` would satisfy the **-applies_to inputs** filter, because an output from a lower boundary port is an input to this domain. **-applies_to** is always relative to the specified domain.

-applies_to_clamp, **-applies_to_sink_off_clamp**, and **-applies_to_source_off_clamp** are satisfied by any port that has the specified value for the **UPF_clamp_value**, **UPF_sink_off_clamp_value**, or **UPF_source_off_clamp_value** port attribute, respectively.

-**applies_to_boundary** restricts the application of filters to specified boundary. The default value is **both**. It shall be an error if **-applies_to_boundary lower** is specified and there is no lower boundary associated with the power domain interface.

The *effective_element_list* (see 5.9) for this command consists of all the port names in the *aggregate_element_list* that are not also in the *aggregate_exclude_list* and that satisfy all of the filters specified in the command. If a port in the *effective_element_list* is not on the interface of the specified domain, it shall not be isolated.

If a given port name is referenced in the *effective_element_list* of more than one isolation strategy of a given domain, the precedence rules (see 5.7) determine which of those strategies actually apply to that port name. If the precedence rules identify multiple strategies that apply to the same port name, then those strategies shall each have a **-sink** filter that matches the receiving supply of a different sink domain for the specified port. It shall be an error if the precedence rules identify multiple strategies that apply to the same port name such that more than one strategy applies to the same sink domain for that port.

If **-no_isolation** is specified, then isolation is not inferred for any port in the *effective_element_list*.

If **-force_isolation** is specified, then isolation is inferred for each port in the *effective_element_list* and the inferred isolation cells are not to be optimized away, even if such optimization does not change the behavior of the design.

If neither **-no_isolation** nor **-force_isolation** is specified, then isolation is inferred for each port in the *effective_element_list*, and implementation tools are free to optimize away isolation cells that are redundant, provided that such optimization does not change the behavior of the design.

-**location** determines the location domain into which an isolation cell is to be inserted.

> **self**—the isolation cell shall be placed inside the self domain, i.e., the domain whose port is being isolated (the default).
>
> **parent**—the isolation cell shall be placed in the parent domain (see 3.1) of the port being isolated. It shall be an error if the port is a port of a design top module, or if the port is a lower boundary port.
>
> **other**—the isolation cell shall be placed in the parent domain (see 3.1) for an upper boundary port, and in the child domain (see 3.1) for a lower boundary port. It shall be an error if an upper boundary port is a port of a design top module, or if a lower boundary port is a port of a leaf cell.
>
> **fanout**—the isolation cell shall be placed in each fanout domain (see 3.1).

An isolation cell shall be inserted within the location domain at a port (or ports) on the location domain boundary. The isolation cell shall be inserted into the instance that contains the port at which the isolation cell is inserted.

If -**location fanout** is specified, the isolation cell shall be inserted at the port on the location domain boundary that is closest to the receiving logic. If the receiving logic is in a macro cell instance, the isolation cell shall be inserted at the input port of that macro cell instance, on the lower boundary of the location domain; otherwise the isolation cell shall be inserted at the location domain port that is driven by the port to which the strategy applies.

If -**location fanout** is not specified, and -**sink** *domain_name* is specified, then the sink domain determines whether the isolation cell is inserted at an input port or an output port of the location domain. If *domain_name* is the name of the location domain, then the isolation cell is inserted at the location domain input port; otherwise an isolation cell is inserted at each location domain output port that drives domain *domain_name*.

If neither **-location fanout** nor **-sink** *domain_name* are specified, then the isolation cell is inserted at the port of the location domain that is (for the self domain), or corresponds to (for the parent or child domain), the port to which the strategy applies.

If any pair of isolation cells are inferred from two different isolation strategies for ports of two different power domains along the same path from a driver to a receiver, and the **-location** specified results in both cells being inserted into the same domain, then the two isolation cells shall be inserted such that the isolation cell contributed by the source domain is placed closer to the driving logic and the isolation cell contributed by the sink domain is placed closer to the receiving logic.

If isolation cell insertion is inferred for different paths from a port, the **-location** specified explicitly or implicitly by the strategy shall be such that the isolation cell(s) can be inserted without splitting the port into multiple ports. It shall be an error if an isolation strategy for a port cannot be implemented without duplicating the port.

The **-clamp_value**, **-isolation_signal** and **-isolation_sense**, and **-isolation_supply** options are each specified as a single value or a list. If any of these options specify a list, then all lists specified for these options shall be of the same length and any single value specified is treated as a list of values of the same length. The tuples formed by associating the positional entries from each list shall be used to define separate isolation requirements for the strategy. These tuples are applied to the isolation cell from the isolation cell's data input port to its data output port in the order in which they appear in each list. The output of the isolation cell shall be the right-most value in the **-clamp_value** list whose corresponding isolation signal is active.

-clamp_value specifies the value of the inferred isolation cell's output when isolation is enabled. The specification may be a single value or a list of values. Any of the following may be specified:

0 (the logic value `0`)

1 (the logic value `1`)

Z (the logic value `Z`)

latch (the value of the non-isolated port when the isolation signal becomes active)

value specifies a value that is legal for the type of the port, e.g., `255` might be specified for an integer-typed port (perhaps constrained to an unsigned 8-bit range).

It shall be an error if **-clamp_value** is not specified.

Verification shall issue an error when a **UPF_sink_off_clamp_value**, **UPF_source_off_clamp_value**, or **UPF_clamp_value** requirement is violated.

-isolation_signal identifies the control signal for each clamp value specified by **-clamp_value**.

-isolation_sense specifies the value that enables isolation, for each signal specified by **-isolation_signal**.

-isolation_supply specifies the supply set(s) that shall be used to power the inferred isolation cell, including the logic receiving the isolation signal(s). The isolation supply set(s) specified by **-isolation_supply** are implicitly connected to the isolation logic inferred by this command.

In verification, while the isolation control signal is asserted,

— if **-isolation_supply** is not specified, then the output of the inferred isolation cell shall be corrupted only if the isolation control input is corrupted.

— if **-isolation_supply** specifies a supply set, then the output of the inferred isolation cell shall be corrupted if the isolation control signal is corrupted or if the current power state of the specified isolation supply set has a non-NORMAL simstate.

— if **-isolation_supply** is specified as an empty list (e.g., {}), then the output of the inferred isolation cell shall be corrupted if the isolation control signal is corrupted or if the rail of the primary supply set of the location domain required for the isolation cell's clamp value has a supply state other than FULL_ON. In this case, it shall be an error if it is not possible to determine the state of the rail that is required for the clamp value.

For implementation tools, **-isolation_supply** shall be specified explicitly with either a supply set or an empty list argument. It shall be an error if **-isolation_supply** is not explicitly specified for an isolation strategy present in a UPF power intent specification that is input to an implementation tool.

Implementation tools shall implement the power intent such that the behavior of the implementation is consistent with the behavior defined above for verification. The specific implementation may vary based on the available cells in the target technology library and optimization decisions made by the tool.

-name_prefix specifies the pattern to generate the substring to place at the beginning of any generated name implementing this strategy (see 6.37).

-name_suffix specifies pattern to generate the substring to place at the end of any generated name implementing this strategy (see 6.37).

-instance specifies that the isolation functionality exists in the HDL design and *instance_name* denotes the instance-providing part or all of this functionality. An *instance_name* is a simple name, rooted name, or design-relative hierarchical name (see 5.3.3.4). If an empty string appears as an *instance_name*, this indicates that an instance was created and then optimized away. Such an instance shall not be re-inferred or reimplemented by subsequent tool runs.

In this case, the following also apply:

— Isolation enable signal(s) are automatically connected to one or more ports of an instance of a cell defined by the library command **define_isolation_cell** (see 7.4). If the strategy specifies multiple isolation enable signals, then the cell shall also be defined with both the **-enable** option and the **-aux_enables** option (see 7.4), the first isolation enable signal shall be connected to the port specified by the **-enable** option, and the rest of the signals shall be connected to the ports specified by the **-aux_enables** option in the same order.

— If the strategy specifies a single isolation supply set, the supply nets of the set shall be automatically connected to the primary supply ports of the isolation cell. If the strategy specifies multiple isolation supply sets, the isolation enable ports shall have related power, ground, and bias port attributes (see 6.47), and the supply nets of the isolation supply set corresponding to each isolation enable signal shall be automatically connected to the supply ports matching the related power, ground, and bias ports of the isolation enable port (see 7.4).

— If there are no supply ports on the instance, then the isolation supply set(s) specified in the strategy shall be implicitly connected to the instance.

— It shall be an error if there is a single isolation enable signal and there is more than one port on the library cell of the instance defined as isolation enable pin or aux enable pin (see 7.4).

-update adds information to the base command executed in the same scope. When specified with **-update**, **-elements** and **-exclude_elements** are additive: the set of instances or ports in the *aggregate_element_list* is the union of all **-elements** specifications given in the base command and any update of this command, and the *aggregate_exclude_list* is the union of all **-exclude_elements** specifications given in the base command and any update of this command.

Tools shall not use information about system power states to avoid inserting isolation as directed by these strategies. However, tools may optionally use information about system power states to issue a warning that certain strategies appear to be unnecessary.

The following also apply:

- This command never applies to inout ports.
- It is erroneous if an isolation strategy isolates its own control signal.
- It shall be an error if **-no_isolation** is specified along with any of the following: **-force_isolation**, **-isolation_signal**, **-isolation_sense**, **-instance**, **-location**, **-name_prefix**, **-name_suffix**, **-isolation_supply**, **-isolation_power_net**, or **-isolation_ground_net**.
- It shall be an error if the isolation supply set is explicity specified and that supply set is not available in the domain in which the isolation cell is inserted.

NOTE 1—To specify an isolation strategy for a port `P` on the lower boundary of a power domain `D` (see 4.4.2), a **set_isolation** command can specify `-domain D` and specify the port name `I/P`, where `I` is the hierarchical name of an instance that is instantiated in domain `D` but is not in the extent of domain `D`, and `P` is the simple name of the port of that instance. The combination of the **-domain** specification and the hierarchical port name makes it clear that this reference is to the HighConn of the specified port, which is part of the lower boundary of the domain `D`.

NOTE 2—The *exclude_list* in **-exclude_elements** can specify instances or ports that have not already been explicitly or implicitly specified via an explicit or implied *element_list*.

NOTE 3—If a **-diff_supply_only**, **-source**, or **-sink** argument is used and instances are included in designs with different power distribution or connectivity, the evaluation of the need for isolation may vary and cause a change in the logical function of a block.

NOTE 4—Isolation clamp value port properties can be annotated in HDL using the attributes shown in 5.6. The same attributes may be specified using the **set_port_attributes** command in 6.47.

NOTE 5—It is not an error if multiple isolation strategies apply to a connection from one domain to another domain.

Syntax example

```
set_isolation parent_strategy
  -domain pda
  -elements {a b c d}
  -isolation_supply {pda_isolation_supply}
    -clamp_value {1}
-applies_to outputs -sink pdb

set_isolation parent_strategy -update
    -domain pda
    -isolation_signal cpu_iso
    -isolation_sense low -location parent
```

6.45 set_level_shifter

Purpose	Specify a level-shifter strategy.		
Syntax	**set_level_shifter** *strategy_name*		
	-domain *domain_name*		
	[**-elements** *element_list*]		
	[**-exclude_elements** *exclude_list*]		
	[**-source** <*source_domain_name* | *source_supply_ref*>]		
	[**-sink** <*sink_domain_name* | *sink_supply_ref*>]		
	[**-use_equivalence** [<**TRUE** | **FALSE**>]]		
	[**-applies_to** <**inputs** | **outputs** | **both**>]		
	[**-applies_to_boundary** <**lower** | **upper** | **both**>]		
	[**-rule** <**low_to_high** | **high_to_low** | **both**>]		
	[**-threshold** <*value*>]		
	[**-no_shift**] [**-force_shift**]		
	[**-location** <**self** | **other** | **parent** | **fanout**>]		
	[**-input_supply** *supply_set_ref*] [**-output_supply** *supply_set_ref*]		
	[**-internal_supply** *supply_set_ref*]		
	[**-name_prefix** *pattern*] [**-name_suffix** *pattern*]		
	[**-instance** **{{***instance_name port_name***}***}]		
	[**-update**]		
Arguments	*strategy_name*	The name of the level-shifter strategy.	
	-domain *domain_name*	The domain for which this strategy is defined.	
	-elements *element_list*	A list of instances or ports to which the strategy potentially applies.	**R**
	-exclude_elements *exclude_list*	A list of instances or ports to which the strategy does not apply.	**R**
	-source <*source_domain_name* | *source_supply_ref*>	The name of a supply set or power domain. When a domain name is used, it represents the primary supply of that domain.	**R**
	-sink <*sink_domain_name* | *sink_supply_ref*>	The name of a supply set or power domain. When a domain name is used, it represents the primary supply of that domain.	**R**
	-use_equivalence [<**TRUE** | **FALSE**>]	Indicates whether to consider supply set equivalence. If **-use_equivalence** is not specified at all, the default is **-use_equivalence TRUE**; if **-use_equivalence** is specified without a value, the default value is **TRUE**.	**R**
	-applies_to <**inputs** | **outputs** | **both**>	A filter that restricts the strategy to apply only to ports of a given direction.	**R**
	-applies_to_boundary <**lower** | **upper** | **both**>	Restricts the application of filters to specified boundary. Default is both.	**R**
	-rule <**low_to_high** | **high_to_low** | **both**>	A filter that restricts the strategy to apply only to ports that require a given level-shifting direction. The default is **both**.	**R**
	-threshold <*value*>	A filter that restricts the strategy to apply only to ports that involve a voltage difference above a certain threshold. The default is `0`.	**R**
	-no_shift	Specifies that level-shifter cells shall not be inserted on the specified ports.	**R**
	-force_shift	Disables any implementation optimization involving level-shifter cells for a given strategy.	**R**

	-location <**self** | **other** | **parent** | **fanout** | >	The location in which inferred level-shifter cells are placed in the logic hierarchy, which determines the power domain in which they shall be inserted. The default is **self**.	**R**
	-input_supply *supply_set_ref*	The supply set used to power the input portion of the level-shifter.	**R**
	-output_supply *supply_set_ref*	The supply set used to power the output portion of the level-shifter.	**R**
	-internal_supply *supply_set_ref*	The supply set used to power internal circuits within the level-shifter.	**R**
	-name_prefix *pattern* **-name_suffix** *pattern*	The name format (prefix and suffix) for generated level-shifter instances or nets related to implementation of the level-shifting strategy.	**R**
	-instance **{**{*instance_name port_name*}*}	The name of a technology library leaf cell instance and the name of the logic port that it level-shifts.	**R**
	-update	Indicates that this command provides additional information for a previous command with the same *strategy_name* and *domain_name* and executed in the same scope.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

The **set_level_shifter** command defines a level-shifting strategy for ports on the interface of a power domain (see 6.20). A level-shifter strategy is applied at the domain boundary, as required to correct for voltage differences between driving and receiving supplies of a port.

-domain specifies the domain for which this strategy is defined.

-elements explicitly identifies a set of candidate ports to which this strategy potentially applies. The *element_list* may contain rooted names of instances or ports in the specified domain. If an instance name is specified in the *element_list*, it is equivalent to specifying all the ports of the instance in the *element_list* but with lower precedence (see 5.7). Any *element_list*s specified on the base command and any *elements_list*s specified in any updates (see **-update**) of the base command are all combined into a single elements list. If **-elements** is not specified in the base command or any update, every port on the interface of the domain is included in the *aggregate_element_list* (see 5.9).

-exclude_elements explicitly identifies a set of ports to which this strategy does not apply. The *exclude_list* may contain rooted names of instances or ports in the specified domain. If an instance name is specified in the *exclude_list*, it is equivalent to specifying all the ports of the instance in the *exclude_list*. Any *exclude_list*s specified on the base command or any updates of the base command are combined into the *aggregate_exclude_list* (see 5.9).

The arguments **-source**, **-sink**, **-applies_to**, **-rule**, and **-threshold** serve as filters that further restrict the set of ports to which a given **set_level_shifter** command applies. The command only applies to those ports that satisfy all of the specified filters.

The **-source** option specifies the simple name, rooted name, or design-relative hierarchical name (see 5.3.3.4) of a power domain or supply set. **-source** is satisfied by any port that is driven by logic powered by a supply set that matches (see **-use_equivalence**) the specified supply set, ignoring any isolation or level-shifting cells that have already been inferred or instantiated from an isolation or level-shifting strategy.

The **-sink** option specifies the simple name, rooted name, or design-relative hierarchical name (see 5.3.3.4) of a power domain or supply set. **-sink** is satisfied by any port that is received by logic powered by a supply set that matches (see **-use_equivalence**) the specified supply set, ignoring any isolation or level-shifting cells that have already been inferred or instantiated from an isolation or level-shifting strategy.

NOTE—A port that does not have a driver will never satisfy the **-source** filter. A port that does not have a receiver will never satisfy the **-sink** filter.

-use_equivalence specifies whether supply set equivalence is to be considered in determining when two supply sets match. If **-use_equivalence** is specified with the value *False*, the **-source** and **-sink** filters shall match only the named supply set. Otherwise, the **-source** and **-sink** filters shall match the named supply set or any supply set that is equivalent to the named supply set.

-applies_to is satisfied by any port that has the specified mode. For upper boundary ports, this filter is satisfied when the direction of the port matches. For lower boundary ports, this filter is satisfied when the inverse of the direction of the port matches. For example, a lower boundary port with a direction `OUT` would satisfy the **-applies_to IN** filter, because an output from a lower boundary port is an input to this domain. **-applies_to** is always relative to the specified domain.

-**applies_to_boundary** restricts the application of filters to specified boundary. The default value is **both**. It shall be an error if **-applies_to_boundary lower** is specified and there is no lower boundary associated with the power domain interface.

-rule is satisfied by any port for which the driving and receiving logic have the specified voltage relationship. If **low_to_high** is specified, a given port satisfies this filter if the voltage of its driver supply is less than the voltage of its receiver supply. If **high_to_low** is specified, a given port satisfies this filter if the voltage of its driver supply is greater than the voltage of its receiver supply. If **-rule both** is specified, a given port satisfies this filter if would satisfy either **-rule low_to_high** or **-rule high_to_low**.

-threshold is satisfied by any port for which the magnitude of the difference between the driver and receiver supply voltages can exceed a specified threshold value. The nominal power and ground of the port's driver supply are compared with the nominal power and ground of the port's receiver supply to determine if level-shifting is required. The variation ranges of respective power and ground supplies are also considered. This option requires tools to use information defined in power states of the supplies involved in a given interconnection between objects with different supplies. If **-threshold** is not specified, it defaults to `0`, which means that a level-shifter will be inserted for a given port if there is any voltage difference.

The following algorithm illustrates how level-shifter insertion is determined. The algorithm below considers only power rails and only does the analysis required for insertion when the input voltage is lower than the output voltage (**-rule low_to_high**). A complete implementation must consider the ground rails also and must also do the analysis for insertion when the input voltage is higher than the output voltage (**-rule high_to_low**).

This algorithm is presented in terms of the voltage requirements of the legal, most-refined power states of the supply sets providing power to the source and sink(s) of the port involved. The same algorithm could be used for analysis based on the actual combinations of power rail values possible in a given implementation, which must be sufficient to cover all the legal power states of the system, but may provide additional supply combinations as well.

In the following, the references to "low_factor" and "high_factor" for a given supply object refer to the variation factors specified by **set_variation** for a supply object S or for any supply object equivalent to S.

```
for each domain port P with a strategy R,
   a source whose supply is A, and sink whose supply is B,
 for each legal power state PSA of source supply A,
   for each legal power state PSB of sink supply B,
      if there is a legal power state containing {PSA, PSB} then
```

```
          /* Get the nominal values for power and
              ground for both supply sets */
          A_nom_pwr = nominal voltage specified for A.power
                      in the supply expression for PSA;
          B_nom_pwr = nominal voltage specified for B.power
                      in the supply expression for PSB;

          /* Check first that this strategy R is applicable */
          if (A_nom_pwr < B_nom_pwr) and (R.rule == high_to_low) then
            return (NOT_APPLICABLE);
          end if;

          /* Check whether level shifting is required based only
             on the nominal voltage */
          if (B_nom_pwr - A_nom_pwr > R.threshold) then
            return (REQUIRED);
          end if;

          /* Determine the min/max voltage values possible given
             the specified variation */
          A_var_min_pwr = A_nom_pwr * (low_factor for A.power);
          A_var_max_pwr = A_nom_pwr * (high_factor for A.power);
          B_var_min_pwr = B_nom_pwr * (low_factor for B.power);
          B_var_max_pwr = B_nom_pwr * (high_factor for B.power);

          /* Check whether level shifting is required based
             only on the voltage variation ranges */
          if correlated (A,B) then
            A_B_min_pwr_diff = B_var_min_pwr - A_var_min_pwr;
            A_B_max_pwr_diff = B_var_max_pwr - A_var_max_pwr;
            if (A_B_min_pwr_diff > R.threshold ||
                A_B_max_pwr_diff > R.threshold) then
               return (REQUIRED);
            end if;
          else /* uncorrelated */
            A_max_B_min_pwr_diff = B_var_max_pwr - A_var_min_pwr;
            A_min_B_max_pwr_diff = B_var_min_pwr - A_var_max_pwr;
            if (A_max_B_min_pwr_diff > R.threshold ||
                A_min_B_max_pwr_diff > R.threshold) then
               return (REQUIRED);
            end if;
          end if;   /* correlated(A,B) */

          return (NOT REQUIRED);
       end if;
     end for;   /* each PSB */
   end for;   /* each PSA */
  end for;  /* each P */
```

The *effective_element_list* (see 5.9) for this command consists of all the port names in the *aggregate_element_list* that are not also in the *aggregate_exclude_list* and that satisfy all of the filters specified in the command. If a port in the *effective_element_list* is not on the interface of the specified domain, it shall not be level-shifted.

If a given port name is referenced in the *effective_element_list* of more than one level-shifting strategy of a given domain, the precedence rules (see 5.7) determine which of those strategies actually apply to that port name. If the precedence rules identify multiple strategies that apply to the same port name, then those strategies shall each have a **-sink** filter that matches the receiving supply of a different sink domain for the specified port. It shall be an error if the precedence rules identify multiple strategies that apply to the same port name such that more than one strategy applies to the same sink domain for that port.

If **-no_shift** is specified, then level-shifting is not inferred for any port in the *effective_element_list*.

If **-force_shift** is specified, then level-shifting is inferred for each port in the *effective_element_list* and the inferred level-shifting cells are not to be optimized away, even if such optimization does not change the behavior of the design.

If neither **-no_shift** nor **-force_shift** is specified, then level-shifting is inferred for each port in the *effective_element_list*, and implementation tools are free to optimize away level-shifting cells that are redundant, provided that such optimization does not change the behavior of the design.

-**location** determines the location domain into which a level-shifter cell is to be inserted.

> **self**—the level-shifter cell shall be placed inside the self domain, i.e., the domain whose port is being shifted (the default).
>
> **parent**—the level-shifter cell shall be placed in the parent domain (see 3.1) of the port being shifted. It shall be an error if the port is a port of a design top module, or if the port is a lower boundary port.
>
> **other**—the level-shifter cell shall be placed in the parent domain (see 3.1) for an upper boundary port, and in the child domain (see 3.1) for a lower boundary port. It shall be an error if an upper boundary port is a port of a design top module, or if a lower boundary port is a port of a leaf cell.
>
> **fanout**—the level-shifter cell shall be placed in each fanout domain (see 3.1).

A level-shifter cell shall be inserted within the location domain at a port (or ports) on the location domain boundary. The level-shifter cell shall be inserted into the instance that contains the port at which the level-shifter cell is inserted.

If -**location fanout** is specified, the level-shifter cell shall be inserted at the port on the location domain boundary that is closest to the receiving logic. If the receiving logic is in a macro cell instance, the level-shifter cell shall be inserted at the input port of that macro cell instance, on the lower boundary of the location domain; otherwise the level-shifter cell shall be inserted at the location domain port that is driven by the port to which the strategy applies.

If -**location fanout** is not specified, and -**sink** *domain_name* is specified, then the sink domain determines whether the level-shifter cell is inserted at an input port or an output port of the location domain. If *domain_name* is the name of the location domain, then the level-shifter cell is inserted at the location domain input port; otherwise a level-shifter cell is inserted at each location domain output port that drives domain *domain_name*.

If neither **-location fanout** nor **-sink** *domain_name* are specified, then the level-shifter cell is inserted at the port of the location domain that is (for the self domain), or corresponds to (for the parent or child domain), the port to which the strategy applies.

If the port at which the level-shifter is inserted is connected to the input or output of an isolation cell, or is connected to the output of one isolation cell and the input of another isolation cell, the level-shifter is inserted either immediately before, or immediately after, or between the isolation cell(s), as appropriate, to achieve the best match between any explicitly specified input/output supplies of the strategy and the actual driver/receiver supplies at each location.

If multiple level-shifter strategies are defined that would insert a level-shifter at the same domain boundary, any of those level-shifter strategies can be applied in any of the preceding locations, in either domain, either singly or in combination. If two potential solutions match the driving and receiving supplies equally well, the solution that applies a level-shifting strategy contributed by a domain closer to the receiving domain shall be used.

-input_supply specifies the supply set connected to input supply ports of the level-shifter. The default is the supply of the logic driving the level-shifter input. The default is used if and only if that supply set is available in the domain in which the level-shifter will be located. It shall be an error if the default supply set is required but is not available.

-output_supply specifies the supply set connected to the output supply ports of the level-shifter. The default is the supply of the logic receiving the level-shifter output. The default is used if and only if that supply set is available in the domain in which the level-shifter will be located. It shall be an error if the default supply set is required but is not available.

Default input and output supply set definitions apply only if exactly one level-shifter strategy applies to a given port, all drivers of that port have equivalent supplies, and all receivers of that port have equivalent supplies. For more complex cases, the required supply sets should be explicitly specified.

If the level-shifter strategy is mapped to a library cell that requires only a single supply, then explicit specification of an input supply set is not required, any explicit input supply set specification is ignored, and the default input supply set does not apply; only the output supply set is used.

-internal_supply specifies the supply set that shall be used to provide power to supply ports that are not related to the inputs or outputs of the level-shifter. There is no default supply set defined for **-internal_supply.**

-name_prefix specifies the pattern to generate the substring to place at the beginning of any generated name implementing this strategy (see 6.37).

-name_suffix specifies the pattern to generate the substring to place at the end of any generated name implementing this strategy (see 6.37).

-instance specifies that the level-shifter functionality exists in the HDL design, and *instance_name* denotes the instance-providing part or all of this functionality. An *instance_name* is a simple name or hierarchical name rooted in the current scope. If an empty string appears as an *instance_name*, this indicates that an instance was created and then optimized away. Such an instance shall not be re-inferred or reimplemented by subsequent tool runs.

-update adds information to the base command executed in the same scope. When specified with **-update**, **-elements** and **-exclude_elements** are additive: the set of instances or ports in the *aggregate_element_list* is the union of all **-elements** specifications given in the base command and any update of this command, and the *aggregate_exclude_list* is the union of all **-exclude_elements** specifications given in the base command and any update of this command.

The following also apply:

— This command never applies to inout ports.

— The simstate semantics of all implicitly connected supply sets apply to the output of a level-shifter.

— It shall be an error if **-no_shift** is specified along with any of the following: **-force_shift**, **-instance**, **-location**, **-name_prefix**, **-name_suffix**, **-input_supply**, **-output_supply**, or **-internal_supply**.

— It shall be an error if there is a connection between a driver and receiver and all of the following apply:
  1) The supplies powering the driver and receiver are at different voltage levels.
  2) A level-shifter is not specified for the connection using a level-shifter strategy.
  3) A level-shifter cannot be inferred for the connection by analysis of the power states of the supplies to the driver and receiver.

— It shall be an error if the input supply set or output supply set is explicitly specified and that supply set is not available in the domain.

NOTE 1—To specify a level-shifting strategy for a port `P` on the lower boundary of a power domain `D`, a **set_level_shifter** command can specify `-domain D` and specify the port name `I/P`, where `I` is the hierarchical name of an instance that is instantiated in domain `D` but is not in the extent of domain `D`, and `P` is the simple name of the port of that instance. The combination of the **-domain** specification and the hierarchical port name makes it clear that this reference is to the HighConn of the specified port, which is part of the lower boundary of the domain `D`.

NOTE 2—The *exclude_list* in **-exclude_elements** can specify instances or ports that have not already been explicitly or implicitly specified via an explicit or implied *element_list*.

NOTE 3—It is not an error if multiple level-shifting strategies apply to a connection from one domain to another domain.

Syntax example

```
set_level_shifter shift_up
  -domain PowerDomainZ
  -applies_to inputs -source PowerDomainX.ss1
  -threshold 0.02
  -rule both
set_level_shifter TurnOffDefaultLS -domain PD -no_shift
//this turns off inference of a default level-shifter for ports on the
//upper boundary of domain PD
```

6.46 set_partial_on_translation

Purpose	Define the translation of **PARTIAL_ON**.
Syntax	**set_partial_on_translation** <**OFF** | **FULL_ON**>
Arguments	**OFF** | **FULL_ON** The value to use in place of **PARTIAL_ON**.
Return value	Return the setting of the translation if successful or raise a `TCL_ERROR` if not.

This command causes translation of **PARTIAL_ON** to **FULL_ON** or **OFF**, as specified by the command argument, for purposes of evaluating the power state of supply sets and power domains. If this command is executed in a given run, the state of a supply set is evaluated after **PARTIAL_ON** is translated to **FULL_ON** or **OFF** for each supply net in the set. If this command is not executed in a given run, no translation of **PARTIAL_ON** is performed.

It shall be an error if this command is invoked with different values in the same UPF description.

Syntax example

```
set_partial_on_translation FULL_ON
```

6.47 set_port_attributes

Purpose	Define information on ports.	
Syntax	**set_port_attributes** [**-model** *name*] [**-elements** *element_list*] [**-exclude_elements** *element_exclude_list*] [**-ports** *port_list*] [**-exclude_ports** *port_exclude_list*] [**-applies_to** <**inputs** | **outputs** | **inouts** | **{**<**inputs** | **outputs** | **inouts** >*}>] [**-attribute {***name value***}**]* [**-clamp_value** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-sink_off_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-source_off_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-driver_supply** *supply_set_ref*] [**-receiver_supply** *supply_set_ref*] [**-literal_supply** *supply_set_ref*] [**-pg_type** *pg_type_value*] [**-related_power_port** *supply_port_name*] [**-related_ground_port** *supply_port_name*] [**-related_bias_ports** *supply_port_name_list*] [**-feedthrough**] [**-unconnected**] [**-is_analog**] [**-is_isolated**]	
Arguments	**-model** *name*	A module or library cell whose ports are to be attributed.
	-elements *element_list*	A list of instances whose ports are to be attributed.
	-exclude_elements *element_exclude_list*	A list of instances whose ports are to be excluded from the command.
	-ports *port_list*	A list of simple names (if used with -model) or rooted names (otherwise) of ports to be attributed.
	-exclude_ports *port_exclude_list*	A list of ports to be excluded from the command.
	-applies_to <**inputs** | **outputs** | **inouts** | **{**<**inputs** | **outputs** | **inouts** >*}>	Indicates whether the specified input ports, output ports, inout ports, or any list of these three choices, are to be attributed.
	-attribute {*name value***}**	The attribute *name* and *value* pair to be associated with the specified ports.
	-clamp_value <**0** | **1** | **any** | **Z** | **latch** | *value*>	The clamp requirement. Equivalent to **-attribute {UPF_clamp_value** *value***}** (see 5.6).
	-sink_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	The clamp requirement when the sink domain's primary supply is not NORMAL. Equivalent to **-attribute {UPF_sink_off_clamp_value** *value***}** (see 5.6).
	-source_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	The clamp requirement when the source domain's primary supply is not NORMAL. Equivalent to **-attribute {UPF_source_off_clamp_value** *value***}** (see 5.6).
	-driver_supply *supply_set_ref*	The supply set used by drivers of the port. Equivalent to **-attribute {UPF_driver_supply** *supply_set_ref***}** (see 5.6).
	-receiver_supply *supply_set_ref*	The supply set used by receivers of the port. Equivalent to **-attribute {UPF_receiver_supply** *supply_set_ref***}** (see 5.6).

	-literal_supply *supply_set_ref*	The supply set used to model a literal value associated with an instance port. Equivalent to **-attribute {UPF_literal_supply** *supply_set_ref*} (see 5.6).
	-pg_type *pg_type_value*	The *pg_type* port. Equivalent to **-attribute {UPF_pg_type** *pg_type_value*} (see 5.6).
	-related_power_port *supply_port_name*	The power port for the attributed port. Equivalent to **-attribute {UPF_related_power_port** *supply_port_name*} (see 5.6).
	-related_ground_port *supply_port_name*	The ground port for the attributed port. Equivalent to **-attribute {UPF_related_ground_port** *supply_port_name*} (see 5.6).
	-related_bias_ports *supply_port_name_list*	The bias port(s) for the attributed port. Equivalent to **-attribute {UPF_related_bias_ports** *supply_port_name_list*} (see 5.6).
	-feedthrough	Indicates that the specified ports are connected together internally to form a feedthrough. Equivalent to **-attribute {UPF_feedthrough TRUE}** (see 5.6).
	-unconnected	Indicates that the specified ports are not connected at all internally. Equivalent to **-attribute {UPF_unconnected TRUE}** (see 5.6).
	-is_isolated	Indicates that the specified ports are internally isolated and do not require external isolation. Equivalent to **-attribute {UPF_is_isolated TRUE}** (see 5.6).
	-is_analog	Indicates that the specified ports are analog ports. Equivalent to **-attribute {UPF_is_analog TRUE}** (see 5.6).
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **set_port_attributes** command specifies information associated with ports of models or instances. Model ports are referenced using -**model**; instance ports are referenced using either -**elements** or -**ports** without -**model**. If -**model** is specified and the model *name* is **.** (a dot), the command applies to the model corresponding to the current scope.

Certain predefined attributes identify a port's related supplies and in doing so may define the lower boundary of a power domain; other predefined attributes provide information relevant to isolation and level-shifting insertion. Predefined attribute **UPF_literal_supply** can only be specified for instance ports; all other predefined attributes can only be specified for model ports.

User-defined attributes may also be associated with a port. The meaning of a user-defined attribute is not specified by this standard. User-defined attributes can be specified for either model ports or instance ports, or both.

The set of ports attributed is determined as follows:

a) A set of candidate ports is first identified. This set includes the following:

   1) If **-elements** is specified, all ports of each instance named in the elements list are included in the candidate set, including any logic ports inferred from **create_logic_port** (see 6.19), but excluding any supply ports.

   2) If **-ports** is specified, each port named in the ports list is included in the candidate set.

   3) If **-model** and **-ports** are specified, each port of the named module or library cell named in the ports list is included in the candidate set.

b) The candidate set is then restricted to those ports that satisfy any filters specified. A port is removed from the candidate set if:

   1) The port name appears in the **-exclude_ports** list.

   2) The port is a port on an instance named in the **-exclude_elements** list.

   3) The port direction is not consistent with any of the directions identified by the **-applies_to** option.

c) The resulting restricted set is the set of ports to be attributed.

If a given port is included in the final candidate set of ports of more than one **set_port_attributes** command, the precedence rules (see 5.7) determine which of those **set_port_attributes** commands actually apply to that port.

If **-model** is specified, the port attributes are applied to the selected ports of the model. In this case, only simple names that are declared in the model may be referenced in arguments to this command and all names are interpreted relative to the topmost scope of the model. If **-model** is not specified, the port attributes are applied to the selected instance ports. In this case, only rooted names of instance ports may be referenced in this command, and all such names are interpreted relative to the current scope.

-model and **-ports** can be used to specify attributes for ports of a hard macro or soft macro. For example, if ports of the macro are connected to each other by the same metal wire, i.e., a feedthrough connection, they should have the **UPF_feedthrough** attribute set to **TRUE**. If a port is not connected to any logic inside the macro, it should have the **UPF_unconnected** attribute set to **TRUE**.

set_port_attributes -model *name* can be specified in the topmost scope of the named model to define attributes of ports of that model. In this case, the specification applies to all instances of the model in any design or soft macro in which it is instantiated.

set_port_attributes -model *name* can also be specified in a scope that is outside the named model to define an attribute of a port of the model if that attribute is not already defined for that port within the model. In this case, the specification applies to all instances of the model that are instantiated in the design or soft macro in which the attribute is specified, from the design top scope down to but not including the leaf cell instances of the design or soft macro. It shall be an error if an attribute of a given port of a given model is defined more than once with different values within a design or a soft macro.

-clamp_value defines the **UPF_clamp_value** attribute, which specifies the clamp value to be used if this port has an isolation strategy applied to it.

-sink_off_clamp defines the **UPF_sink_off_clamp_value** attribute, which specifies the clamp requirement when the supply set connected to the sink is in a power state with a corresponding simstate other than NORMAL.

-source_off_clamp defines the **UPF_source_off_clamp_value** attribute, which specifies the clamp requirement when the supply set connected to the source is in a power state with a corresponding simstate other than NORMAL.

When a user-defined clamp *value* is specified for **UPF_clamp_value** or **UPF_sink_off_clamp_value** or **UPF_source_off_clamp_value**, it shall be a legal value for the type of the port. A clamp value of **any** specifies any clamp value legal for the port type is allowed. If the port needs to be isolated in a given context, the specific clamp value to use shall be specified in a **set_isolation** command (see 6.44).

-driver_supply and **-receiver_supply** define the attributes **UPF_driver_supply** or **UPF_receiver_supply**, respectively. These attributes can be used to specify the driver supply of a macro cell output port or the receiver supply of a macro cell input port. They can also be used to specify the assumed driver supply of external logic driving a primary input or to specify the assumed receiver supply of external logic receiving a primary output, when the macro is implemented separately from the context in which it will be instantiated. These attributes are ignored if applied to a port that is not on a macro boundary.

When the **UPF_driver_supply** attribute is defined for an output port, it specifies the driver supply of the logic driving the port. If the driving logic is not present within the model or instance whose port is being attributed, it is presumed the specified driver supply is the supply for the driver logic; therefore, the port is corrupted when the driver supply is in a simstate other than **NORMAL**. For an output port with the attribute **UPF_driver_supply**, when that port has a single source and the driving logic is present within the model or instance whose port is being attributed, it shall be an error if the actual supply of the driving logic is not the same as, or equivalent to, the specified driver supply. The actual supply of the driving logic is the supply of the logic element driving this port after applying all strategies in the power intent, and therefore may be the supply of a retention cell, a repeater cell, an isolation cell, or a level-shifter cell inserted by such a strategy.

When the **UPF_receiver_supply** attribute is defined for an input port, it specifies the receiver supply of the logic receiving the port. If the receiving logic is not present within the model or instance whose port is being attributed, it is presumed the specified receiver supply is the supply for the receiving logic. For an input port with the attribute **UPF_receiver_supply**, when that port has a single receiver supply and the receiving logic is present within the model or instance whose port is being attributed, it shall be an error if the actual supply of the receiving logic is not the same as, or equivalent to, the specified receiver supply. The actual supply of the receiving logic is the supply of the logic element driven by this port after applying all strategies in the power intent, and therefore may be the supply of a retention cell, a repeater cell, an isolation cell, or a level-shifter cell inserted by such a strategy.

If **UPF_driver_supply** is not defined for a primary input port or **UPF_receiver_supply** is not defined for a primary output port, the default driver supply or receiver supply, respectively, assumed to be the external supply of that port for verification and implementation of this design, is an anonymous supply set that is not equivalent to any other supply set.

The ports of the top-level module are always considered to be on a macro boundary with regards to evaluating **UPF_driver_supply** and **UPF_receiver_supply** attributes.

NOTE—The scope in find_objects can be set to any scope that **set_scope** in a given UPF can reach. However, **find_objects** is prohibited from initiating a search that starts in a lower scope that is a leaf cell or below a leaf cell with respect to the current scope.

-pg_type defines the **UPF_pg_type** attribute on a supply port for use with automatic connection semantics. *pg_type_value* is a string denoting the supply port type.

NOTE—**UPF_pg_type** only applies to supply ports and is the only predefined attribute that applies to supply ports. All other attributes apply to logic ports.

If any of **-related_power_port**, **-related_ground_port**, or **-related_bias_ports** is specified, an implicit supply set is created consisting of the supply nets connected to the specified ports. If **-related_power_port** *supply_port_name* and **-related_ground_port** *supply_port_name* are specified, the specified *supply_port_name*s shall be used as the power and ground functions, respectively, of the implicit supply set. If **-related_bias_ports** *supply_port_name_list* is specified, each port in the *supply_port_list* shall have a *pg_type* of `nwell`, `pwell`, `deepnwell`, or `deeppwell`, and each port shall be used as the appropriate bias function of the implicit supply set, as indicated by the value of the associated attribute.

If the port being attributed is `in` mode, the related ports specify the **UPF_receiver_supply** attribute of the port being attributed, as if the implicitly created supply set were specified as the **-receiver_supply** argument. If the port being attributed is `out` mode, the related ports specify the **UPF_driver_supply** attribute of the port being attributed, as if the implicitly created supply set were specified as the

-driver_supply argument. If the port being attributed is `inout` mode, the related ports specify both the **UPF_receiver_supply** and the **UPF_driver_supply** attributes of the port being attributed, as if the implicitly created supply set were specified as both the **-receiver_supply** and the **-driver_supply** arguments.

By the previous definition, related supplies always refer to the driver and receiver supplies of the logic inside a module.

-**literal_supply** defines the **UPF_literal_supply** attribute, which identifies the supply set to be used to implement a literal constant value associated with an input port of an instance. It shall be an error if this attribute is specified for an instance port that is not driven by a literal constant.

-feedthrough defines the **UPF_feedthrough** attribute, which identifies a set of ports on the interface of a module or cell that are directly connected to each other inside the module or cell and therefore create a feedthrough through the module or cell.

-unconnected defines the **UPF_unconnected** attribute, which identifies a port on the interface of a module or cell that is not connected to either a source or sink within the module or cell and is not connected to any other port on the interface of the module or cell.

-is_isolated defines the **UPF_is_isolated** attribute, which identifies a port on the interface of a module or cell that is internally isolated and does not require external isolation.

-is_analog defines the **UPF_is_analog** attribute, which identifies a signal port on the interface of a module or cell that is an analog port.

The following also apply:

- It shall be an error if **-model** is specified and **-elements** is also specified.
- It shall be an error if any predefined attribute other than attribute **UPF_literal_supply** is specified without **-model.**
- It shall be an error if one of the attributes **UPF_related_power_port** and **UPF_related_ground_port** is specified for a port, but not both.
- It shall be an error if attribute **UPF_related_bias_ports** is specified for a port, but either attribute **UPF_related_port_power** or attribute **UPF_related_ground_port** is not specified for that port.
- It shall be an error if a supply port is included in **-ports** and that port has no **UPF_pg_type** attribute.
- It shall be an error if **UPF_pg_type** is specified for a port that is not a supply port.
- It shall be an error if no argument is used.
- It shall be an error if **-ports** is specified and **-elements** is also specified.
- It shall be an error if attribute **UPF_driver_supply** or **UPF_receiver_supply** is specifed for a macro port that also has the attribute **UPF_unconnected** associated with it.
- It shall be an error if an analog port appears in the element_list of a strategy.
- It shall be an error if an analog port is connected to a port that is not an analog port.

Examples

Specifying clamp value constraints:

```
set_port_attributes -model M -ports {outP} -clamp_value 1
```

or

```
set_port_attributes -model M -ports {outP} -attribute {UPF_clamp_value "1"}
```

Specifying the driver supply for a model's output port:

```
set_port_attributes -model M -ports {outP}
   -attribute {UPF_related_power_port "my_VDD"}

set_port_attributes -model M -ports {outP}
   -attribute {UPF_related_ground_port "my_VSS"}

set_port_attributes -model M -ports {outP}
   -attribute {UPF_related_bias_ports "my_VNWELL my_VPWELL"}
```

or

```
set_port_attributes -model M -ports {outP}
   -driver_supply localSS
```

Specifying the assumed driver supply for an model's input port:

```
set_port_attributes -model M -ports {inP}
   -driver_supply aonSS
```

Specifying the literal supply for an instance's input port:

```
set_port_attributes -ports {i1/inP}
   -literal_supply /top/aonSS
```

6.48 set_repeater

Purpose	Specify a repeater (buffer) strategy.		
Syntax	**set_repeater** *strategy_name* **-domain** *domain_name* [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-source** <*source_domain_name* | *source_supply_ref*>] [**-sink** <*sink_domain_name* | *sink_supply_ref*>] [**-use_equivalence** [<**TRUE** | **FALSE**>]] [**-applies_to** <**inputs** | **outputs** | **both**>] [**-applies_to_boundary** <**lower** | **upper** | **both**>] [**-repeater_supply** *supply_set_ref*] [**-name_prefix** *string*] [**-name_suffix** *string*] [**-instance** **{**{*instance_name port_name*}***}**] [**-update**]		
Arguments	*strategy_name*	The name of the repeater strategy.	
	-domain *domain_name*	The domain for which this strategy is defined.	
	-elements *element_list*	A list of instances or ports to which the strategy potentially applies.	**R**
	-exclude_elements *exclude_list*	A list of instances or ports to which the strategy does not apply.	**R**
	-source <*source_domain_name* | *source_supply_ref*>	The name of a supply set or power domain. When a domain name is used, it represents the primary supply of the specified domain.	**R**
	-sink <*sink_domain_name* | *sink_supply_ref*>	The name of a supply set or power domain. When a domain name is used, it represents the primary supply of the specified domain.	**R**
	-use_equivalence [<**TRUE** | **FALSE**>]	Indicates whether to consider supply set equivalence. If **-use_equivalence** is not specified at all, the default is **-use_equivalence TRUE**; if **-use_equivalence** is specified without a value, the default value is **TRUE**.	**R**
	-applies_to <**inputs** | **outputs** | **both**>	A filter that restricts the strategy to apply only to ports of a given direction.	**R**
	-applies_to_boundary <**lower** | **upper** | **both**>	Restricts the application of filters to specified boundary. Default is both.	**R**
	-repeater_supply *supply_set_ref*]	The supply set that powers the inserted buffer.	**R**
	-name_prefix *string*] [**-name_suffix** *string*]	The name format (prefix and suffix) for inserted buffer cell instances or nets related to implementation of the strategy.	**R**
	-instance **{**{*instance_name port_name*}***}**	The name of a technology library leaf cell instance and the name of the logic port that it buffers.	**R**
	-update	Indicates that this command provides additional information for a previous command with the same *strategy_name* and *domain_name* and executed in the same scope.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

The **set_repeater** command defines a strategy for inserting repeater cells (buffers) for ports on the interface of a power domain (see 6.20). Repeaters are placed within the domain, driven by input ports of the domain, and driving output ports of the domain.

-domain specifies the domain for which this strategy is defined.

-elements explicitly identifies a set of candidate ports to which this strategy potentially applies. The *element_list* may contain rooted names of instances or ports in the specified domain. If an instance name is specified in the *element_list*, it is equivalent to specifying all the ports of the instance in the *element_list*. Any *element_list*s specified on the base command or any updates (see **-update**) of the base command are combined. If **-elements** is not specified in the base command or any update, every port on the interface of the domain is included in the *aggregate_element_list* (see 5.9).

-exclude_elements explicitly identifies a set of ports to which this strategy does not apply. The *exclude_list* may contain rooted names of instances or ports in the specified domain. If an instance name is specified in the *exclude_list*, it is equivalent to specifying all the ports of the instance in the *exclude_list*. Any *exclude_list*s specified on the base command or any updates of the base command are combined into the *aggregate_exclude_list* (see 5.9).

The arguments **-source**, **-sink**, and **-applies_to** serve as filters that further restrict the set of ports to which a given **set_repeater** command applies. The command only applies to those ports that satisfy all of the specified filters.

The **-source** option specifies the simple name, rooted name, or design-relative hierarchical name (see 5.3.3.4) of a power domain or supply set. **-source** is satisfied by any port that is driven by logic powered by a supply set that matches (see **-use_equivalence**) the specified supply set, ignoring any isolation or level-shifting cells that have already been inferred or instantiated from an isolation or level-shifting strategy.

The **-sink** option specifies the simple name, rooted name, or design-relative hierarchical name (see 5.3.3.4) of a power domain or supply set. **-sink** is satisfied by any port that is received by logic powered by a supply set that matches (see **-use_equivalence**) the specified supply set, ignoring any isolation or level-shifting cells that have already been inferred or instantiated from an isolation or level-shifting strategy.

NOTE—A port that does not have a driver will never satisfy the **-source** filter. A port that does not have a receiver will never satisfy the **-sink** filter.

-use_equivalence specifies whether supply set equivalence is to be considered in determining when two supply sets match. If **-use_equivalence** is specified with the value *False*, the **-source** and **-sink** filters shall match only the named supply set. Otherwise, the **-source** and **-sink** filters shall match the named supply set or any supply set that is equivalent to the named supply set.

-applies_to is satisfied by any port that has the specified mode. For upper boundary ports, this filter is satisfied when the direction of the port matches. For lower boundary ports, this filter is satisfied when the inverse of the direction of the port matches. For example, a lower boundary port with a direction `OUT` would satisfy the `-applies_to IN` filter, because an output from a lower boundary port is an input to this domain. **-applies_to** is always relative to the specified domain.

-**applies_to_boundary** restricts the application of filters to specified boundary. The default value is **both**. It shall be an error if **-applies_to_boundary lower** is specified and there is no lower boundary associated with the power domain interface.

The *effective_element_list* (see 5.9) for this command consists of all the port names in the *aggregate_element_list* that are not also in the *aggregate_exclude_list* and that satisfy all of the filters specified in the command. If a port in the *effective_element_list* is not on the interface of the specified domain, it shall not be buffered.

If a given port name is referenced in the *effective_element_list* of more than one repeater strategy of a given domain, the precedence rules (see 5.7) determine which of those strategies actually apply to that port name. If the precedence rules identify multiple strategies that apply to the same port name, then the port name shall be the name of an input port to the domain, and each of those strategies shall each have a **-sink** filter that matches the receiving supply of a different sink domain for the specified input port. It shall be an error if the precedence rules identify multiple strategies that apply to the same port name and that port is an output port of the domain, or more than one strategy applies to the same sink domain for that port.

-repeater_supply is implicitly connected to the primary or backup supply ports of the buffer cell. If **-repeater_supply** is not specified, then if the primary supply set of the domain containing the driver of the repeater is available in the power domain where the repeater will be located, that supply is used as the default supply. It shall be an error if **-repeater_supply** is not specified and the default supply is not available in the domain.

-name_prefix specifies the substring to place at the beginning of any generated name implementing this strategy.

-name_suffix specifies the substring to place at the end of any generated name implementing this strategy.

-instance specifies that the repeater functionality exists in the HDL design and *instance_name* denotes the instance-providing part or all of this functionality. An *instance_name* is a simple name or a hierarchical name rooted in the current scope. If an empty string appears as an *instance_name*, this indicates that an instance was created and then optimized away. Such an instance shall not be re-inferred or reimplemented by subsequent tool runs.

-**update** adds information to the base command executed in the same scope. When specified with -**update**, -**elements** and **-exclude_elements** are additive: the set of instances or ports in the *aggregate_element_list* is the union of all **-elements** specifications given in the base command and any update of this command, and the *aggregate_exclude_list* is the union of all **-exclude_elements** specifications given in the base command and any update of this command.

The following also apply:

— This command never applies to inout ports.

— The simstate semantics of the repeater supply set apply to the output of a repeater.

NOTE 1—To specify a repeater strategy for a port `P` on the lower boundary of a power domain `D` (see 4.4.2), a **set_repeater** command can specify `-domain D` and specify the port name `I/P`, where `I` is the hierarchical name of an instance that is instantiated in domain `D` but is not in the extent of domain `D`, and `P` is the simple name of the port of that instance. The combination of the **-domain** specification and the hierarchical port name makes it clear that this reference is to the HighConn of the specified port, which is part of the lower boundary of the domain `D`.

NOTE 2—Insertion of a repeater can change the driver supply and receiver supply of ports that are sinks or sources, respectively, of the inserted repeater. Such changes could affect the interpretation of -**source** or -**sink** filters of **set_isolation** (see 6.44) or **set_level_shifter** (see 6.45) strategies that apply to those ports. These changes could also affect the default for the input supply set or the output supply set of **set_level_shifter** strategies that apply to those ports.

NOTE 3—The *exclude_list* in -**exclude_elements** can specify instances or ports that have not already been explicitly or implicitly specified via an explicit or implied *element_list*.

Syntax example

```
set_repeater feedthrough_buffer1
-domain PD3 -applies_to outputs
```

6.49 set_retention

Purpose	Specify a retention strategy.		
Syntax	**set_retention** *retention_name* **-domain** *domain_name* [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-retention_supply** *ret_supply_set*] [**-no_retention**] [**-save_signal** {*logic_net* <**high** | **low** | **posedge** | **negedge**>} **-restore_signal** {*logic_net* <**high** | **low** | **posedge** | **negedge**>}] [**-save_condition** {*boolean_expression*}] [**-restore_condition** {*boolean_expression*}] [**-retention_condition** {*boolean_expression*}] [**-use_retention_as_primary**] [**-parameters** {<**RET_SUP_COR** | **NO_RET_SUP_COR** | **SAV_RES_COR** | **NO_SAV_RES_COR**> *}] [**-instance** {{*instance_name* [*signal_name*]}*}] [**-update**] [**-retention_power_net** *net_name*] [**-retention_ground_net** *net_name*]		
Arguments	*retention_name*	Retention strategy name.	
	-domain *domain_name*	The domain for which this strategy is applied.	
	-elements *element_list*	The **-elements** option specifies a list of objects: instances, *retention_list_name* of elements lists (see 6.50), named processes, or state elements or signal names to which this strategy is applied.	**R**
	-exclude_elements *exclude_list*	The **-exclude_elements** option specifies a list of objects: instances, named processes, or state elements or signal names that are not included in this strategy.	**R**
	-no_retention	Prevents the inference of retention cells on the specified elements regardless of any other specifications.	**R**
	-retention_supply *ret_supply_set*	This option specifies the supply set used to power the logic inferred by the *retention_name* strategy.	**R**
	-save_signal {*logic_net* <**high** | **low** | **posedge** | **negedge**>} **-restore_signal** {*logic_net* <**high** | **low** | **posedge** | **negedge**>}	The **-save_signal** and **-restore_signal** options specify a rooted name of a logic net or port and its active level or edge.	**R**
	-save_condition {*boolean_expression*}	The **-save_condition** option specifies a Boolean expression (see 5.4). The default is *True* if the **-save_signal**/**-restore_signal**s are specified, else the **-save_condition** is a don't care.	**R**
	-restore_condition {*boolean_expression*}	The **-restore_condition** option specifies a Boolean expression. The default is *True* if the **-save_signal**/**-restore_signal**s are specified, else the **-restore_condition** is a don't care.	**R**
	-retention_condition {*boolean_expression*}	The **-retention_condition** option specifies a Boolean expression.	**R**
	-use_retention_as_primary	The **-use_retention_as_primary** option specifies that the storage element and its output are powered by the retention supply.	**R**
	-parameters {<**RET_SUP_COR** | **NO_RET_SUP_COR** | **SAV_RES_COR** | **NO_SAV_RES_COR**> *}	The **-parameters** option provides control over retention register corruption semantics.	**R**

	-instance {{*instance_name* [*signal_name*]}*}	The name of a technology library leaf cell instance and the optional name of the signal that it retains. If this instance has any unconnected supply ports or save and restore control ports, then these ports need to have identifying attributes in the cell model, and the ports shall be connected in accordance with this **set_retention** command.	**R**
	-update	Use **-update** if the *retention_name* has already been defined.	**R**
Legacy arguments	**-retention_power_net** *net_name*	This option defines the supply net used as the power for the retention logic inferred by this strategy. This is a legacy option; see also 6.2 and Annex D.	**R**
	-retention_ground_net *net_name*	This option defines the supply net used as the ground for the retention logic inferred by this strategy. This is a legacy option; see also 6.2 and Annex D.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

The **set_retention** command specifies a set of objects in the domain that need to be retention registers and identifies the save and restore behavior. If an instance is specified, all registers within the instance acquire the specified retention strategy. If a process is specified, all registers inferred by the process acquire the specified retention strategy. If a `reg`, signal, or variable is specified and that object is a sequential element, the implied register acquires the specified retention strategy. Any specified `reg`, signal, or variable that does not infer a sequential element shall not be changed by this command.

If **-elements** is specified, only elements in the element list that are also a part of the *domain_name* are included. Any element names outside the extent of *domain_name* are excluded. When **-elements** is not specified, this is equivalent to using the elements list that defines the power domain. When used with **-update**, **-elements** is additive such that the set of elements or signals is the union of all calls of this command for a given strategy specifying any of these parameters.

-exclude_elements can also be used to define a list of storage elements that are not included in this strategy. When used with -**update**, **-exclude_elements** is additive such that the set of elements or signals excluded is the union of all calls of this command for a given strategy.

-retention_supply specifies the supply set that shall be used to power the state element holding the retained value, as well as the control logic, if any, that evaluates the **-save_condition**, **-restore_condition**, and **-retention_condition**. The supply set specified by **-retention_supply** is implicitly connected to the retention logic inferred by this command.

In verification, if **-retention_supply** is not specified, an anonymous always-on supply set shall be assumed to power the state element holding the retained value and any associated control logic. In implementation, it shall be an error if the supply required for the correct operation of inferred retention cells is not specified explicitly in the UPF power intent.

For a balloon-style retention register (see 4.4.5), the retained value is transferred to the register on the restore event when **-restore_condition** evaluates to *True*. The restore event is the rising or falling edge of an edge-triggered restore event or the trailing edge of a level-sensitive restore event. A level-sensitive restore event has priority over any other register operation.

-restore_condition gates the restore event, defining the restore behavior of the register. If the **-save_signal**/**restore_signal**s are not specified, the **-restore_condition** becomes a don't care. The register is restored when the restore event occurs and the **-restore_condition** is *True*.

For a balloon-style retention register, the retained value shall be the register's value at the time of the save event when **-save_condition** evaluates to *True*. The save event is the rising or falling edge of an edge-triggered save event or the trailing edge of a level-sensitive save event.

-save_condition gates the save event, defining the save behavior of the register. If the **-save_signal**/ **restore_signal** options are not specified, the **-save_condition** becomes a don't care. The register contents are saved when the save event occurs and the **-save_condition** is *True*.

-retention_condition defines the retention behavior of the retention element. If the **-retention_condition** is specified, it must evaluate to TRUE for the value of the state element to be retained. If the retention condition evaluates to FALSE and the primary supply is not NORMAL, the retained value of the state element is corrupted. The receiving supply of any pin listed in the **-retention_condition** shall be at least as on as the retention supply of the retention strategy.

-save_condition, **-restore_condition**, and **-retention_condition** shall only reference logic nets or ports rooted in the current scope. The **-save_signal**/**-restore_signal**/**-save_condition**/**-restore_condition** apply only to balloon-style retention registers. For master-/slave-alive implementations (see 4.4.5), the **-save_signal**/**-restore_signal** should not be specified. The retention behavior of this style is specified through the **-retention_condition**. It shall be an error if **-save_signal**/**-restore_signal** is not specified and the **-retention_condition** is also not specified.

The normal mode storage element of the retention register is powered by the primary supply of the domain, therefore the receiver supply of the retention register's data input is the primary supply. By default, the output driver of the retention register is also powered by the primary supply of the domain, in which case the driver supply of the retention register output is the primary supply. However, if **-use_retention_as_primary** is specified, the retention supply powers the output driver of the register instead, and the driver supply of the data output of the retention register is therefore the retention supply. In the latter case, the simstate for the retention supply set is applied to the register's output. Inferred state elements shall be consistent with the **-use_retention_as_primary** constraint.

NOTE 1—UPF only supports the output pins' driving supply being different from the primary supply (with **-use_retention_supply_as_primary**); the input pins' receiving supply can only be assumed to be the primary supply of the domain.

NOTE 2—The **-use_retention_as_primary** changes the driver supply of ports that are sinks of the inserted retention register. Such changes could affect the interpretation of the **-source** filters of the **set_repeater** (see 6.48), **set_isolation** (see 6.44), or **set_level_shifter** (see 6.45) strategies that apply to those ports.

The **-parameters** option provides control over retention register corruption semantics. For a retention strategy, it shall be an error to specify:

— both **RET_SUP_COR** and **NO_RET_SUP_COR**; or
— both **SAV_RES_COR** and **NO_SAV_RES_COR**.

RET_SUP_COR activates and **NO_RET_SUP_COR** deactivates corruption of the normal mode register when retention supplies are **CORRUPT**. When neither value is specified for a retention strategy, **RET_SUP_COR** is the default value.

SAV_RES_COR activates and **NO_SAV_RES_COR** deactivates corruption of the normal mode register during concurrent assertion of level-sensitive **save**, **save_condition**, **restore**, and **restore_condition**. When neither value is specified for a retention strategy, **SAV_RES_COR** is the default value.

-instance specifies that the retention functionality exists in the HDL design and *instance_name* denotes the instance-providing part or all of this functionality. An *instance_name* is a hierarchical name rooted in the current scope. If an empty string appears in an *instance_name*, this indicates that an instance was created and then optimized away. Such an instance shall not be re-inferred or reimplemented by subsequent tool runs.

-update adds information to the base command executed in the same scope of the power domain for which the inferred cells are defined.

The elements requiring retention can be attributed in HDL as shown in 5.6.

For details on the simulation semantics of this command, please refer to 9.7.

Examples

Some examples of the **set_retention** command are shown as follows:

a) Save-restore balloon-type RFF:

   Has an explicit save and restore pin, which perform save/restore functions.

```
set_retention my_ret \
-save_signal {save high} \
-restore_signal {restore high} \
 ...
```

b) Single retention pin balloon-type RFF:

   1) Has a single pin that performs save/restore functions.

   2) To remain in a retention state, the retention pin shall be kept at a certain value.

```
set_retention my_ret \
-save_signal {ret posedge} \
-restore_signal {ret negedge} \
-retention_condition {ret} \
...
```

c) Single retention pin slave-alive type RFF:

   1) Has a single retention control pin, but no save/restore function is involved as the slave latch (or storage element) is powered by the retention supply.

   2) Requires the retention pin to remain at a certain value to be in retention mode.

```
set_retention my_ret \
-retention_condition {ret} \
...
```

NOTE—No save/restore signals/conditions are specified in this case. Here, the retention condition is explicitly specified, meaning the retention condition has to be true during retention mode.

d) No retention pin slave alive type RFF with output powered by retention supply:

   1) Has no retention control pin, and no save/restore function is involved as the slave latch (or storage element) is powered by the retention supply.

   2) Requires the clocks/async sets/resets to be related to retention supply and parked at a certain value during retention mode.

   3) The **-use_retention_as_primary** is specified as the output is expected to be powered by the retention supply.

```
set_retention my_ret \
-retention_condition {!clock && !reset} \
-use_retention_as_primary \
...
```

6.50 set_retention_elements

Purpose	Create a named list of elements whose collective state shall be maintained if retention is applied to any of the elements in the list.	
Syntax	**set_retention_elements** *retention_list_name* **-elements** *element_list* [**-applies_to** <**required** | **not_optional** | **not_required** | **optional**>] [**-exclude_elements** *exclude_list*] [**-retention_purpose** <**required** | **optional**>] [**-transitive** [<**TRUE** | **FALSE**>]]	
Arguments	*retention_list_name*	A simple name; this shall be unique within the current *scope*.
	-elements *element_list*	A list of rooted names: instances, named processes, state elements, or signal names.
	-applies_to <**required** | **not_optional** | **not_required** | **optional**>	Filter elements based on the **UPF_retention** attribute value.
	-exclude_elements *exclude_list*	A list of rooted names: instances, named processes, state elements, or signal names.
	-retention_purpose <**required** | **optional**>	The intended retention use of *retention_list_name*. The default is **required**.
	-transitive [<**TRUE** | **FALSE**>]	If **-transitive** is not specified at all, the default is **-transitive TRUE**. If **-transitive** is specified without a value, the default value is **TRUE**.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **set_retention_elements** command defines a list of state elements whose collective state shall be maintained coherently if retention is applied to any of these elements. (see 6.49 and 6.35).

It shall be an error if the collective state of the elements is not maintained while any one element is in a retention state.

-applies_to filters the *effective_element_list,* removing any elements that are not consistent with the selected filter choice: **required**, **optional**, **not_required**, or **not_optional**, as follows:

— Filter choice **required** removes all elements that do not have the **UPF_retention** attribute value `required`.

— Filter choice **optional** removes all elements that do not have the **UPF_retention** attribute value `optional`.

— Filter choice **not_required** removes all elements that do have the **UPF_retention** attribute value `required`.

— Filter choice **not_optional** removes all elements that do have the **UPF_retention** attribute value `optional`.

When **-retention_purpose** is **required**, retention shall only be necessary if elements in the *retention_element_list* are in the extent of a power domain that has retained elements.

It shall be an error if **retention_purpose** is **required** and an element belonging to *retention_element_list* is not retained when any element in the same power domain extent is retained.

Syntax example

```
set_retention_elements ret_chk_list
    -elements {proc_1 sig_a}
```

6.51 set_scope

Purpose	Specify the current scope.	
Syntax	**set_scope** *instance*	
Arguments	*instance*	The instance that becomes the current scope upon completion of the command.
Return value	Return the current scope prior to execution of the command as a design-relative hierarchical name (see 5.3.3.4) if successful or raise a `TCL_ERROR` if it fails (e.g., if the instance does not exist).	

The **set_scope** command changes the current scope to the specified scope and returns the name of the previous scope as a design-relative hierarchical name.

The following also apply:

- The instance name may be a simple name, a scope-relative hierarchical name, a design-relative hierarchical name, the symbol `/`, the symbol `.`, or the symbol `..`.
- If the instance name is `/`, the current scope is set equal to the current design top instance.
- If the instance name is `.`, the current scope is unchanged.
- If the instance name is `..`, and the current scope is not equal to the current design top instance, the current scope is changed to the parent scope.
- It shall be an error if the instance name is `..` and the current scope is equal to the current design top instance.
- It shall be an error if any prefix of the instance name is the name of a leaf cell instance in the logic hierarchy.

Examples

Given the hierarchy:

```
top/
    mid/
        bot/
```

if the current design top instance is `/top`, and the current scope is `/top/mid`, then:

```
set_scope bot ;# changes current scope to /top/mid/bot (child of current
    scope)
set_scope . ;# leaves current scope unchanged as /top/mid (current scope)
set_scope .. ;# changes current scope to /top (parent of current scope)
set_scope / ;# changes current scope to /top (current design top instance)
```

If the current design top instance is `/top/mid` and the current scope is `/top/mid`, then:

```
set_scope bot ;# changes current scope to /top/mid/bot
set_scope . ;# leaves current scope unchanged as /top/mid
set_scope .. ;# results in an error
set_scope / ;# changes current scope to /top/mid (current design top
    instance)
```

If the current design top instance is `/top` and the current scope is `/top`, then:

```
set_scope mid/bot ;# changes current scope to /top/mid/bot
set_scope . ;# leaves current scope unchanged as /top
set_scope .. ;# results in an error
set_scope / ;# changes current scope to /top (current design top instance)
```

6.52 set_simstate_behavior

Purpose	Specify the simulation simstate behavior for a model or library.	
Syntax	**set_simstate_behavior <ENABLE | DISABLE>** [**-lib** *name*] [**-models** *model_list*] [**-elements** *element_list*] [**-exclude_elements** *exclude_list*]	
Arguments	<**ENABLE | DISABLE**>	Define if the UPF simstate behavior shall be enabled for the specified model(s).
	-lib *name*	The library name.
	-models *model_list*	One or more model names.
	-elements *element_list*	A list of instances.
	-exclude_elements *exclude_list*	A list of instances to exclude from the *effective_element_list* (see 5.9).
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

This command specifies the simstate behavior for models or instances.

If **ENABLE** is specified, the simstate simulation semantics are applied for every supply set automatically connected to an instance of the model. See also 9.5.

a) If there is a single supply set connected, the simstates for that supply set are applied.

b) When no supply set is connected, and each port to which a supply net is connected is of a different *`pg_type`*, an anonymous supply set is created containing the supply nets connected to each port, with each supply net associated with the function appropriate for the *`pg_type`* of that port, and the default simstates for that supply set are applied for the model.

c) When there are multiple supply sets connected, the simstates of all supply sets are applied.

d) For a hard macro instance in which there are multiple supply pins of the same *`pg_type`*, an anonymous supply set is created for each unique combination of supply pins identified as related supplies of a logic pin of the macro instance, with each supply pin associated with the function appropriate for the *`pg_type`* of that pin. The default simstates of each supply set are applied during simulation for any logic pin related to that supply set.

e) For an instance of a hard macro behavioral model, each logic pin of the instance is corrupted according to the applicable simstate of the supply set associated with the logic pin.

If **-models** is not defined and **-lib** is specified, the simstate behavior is defined for all models in *name*.

It shall be an error if:

— **-models** is specified and any of the model(s) cannot be found.

— **-elements** is specified and any of the element(s) cannot be found.

— **-exclude_elements** is specified and any of the *exclude_elements*(s) cannot be found.

— **-exclude_elements** is specified and **-model**, **-elements**, or **-lib** is not specified.

— A given model has its simstate behavior both enabled and disabled, by **set_simstate_behavior** commands, **UPF_simstate_behavior** attributes, or a combination thereof.

— *effective_element_list* is empty.

Simstate behavior of a module can be enabled or disabled in HDL using the following attributes:

Attribute name: **UPF_simstate_behavior**

Attribute value: <**ENABLE** | **DISABLE**>

SystemVerilog or Verilog-2005 example

```
(* UPF_simstate_behavior = "ENABLE" *) module my_adder;
```

VHDL example

```
attribute UPF_simstate_behavior of my_adder : entity is "ENABLE";
```

Syntax example

```
set_simstate_behavior ENABLE -lib library1 -models ANDX7_non_power_aware
```

6.53 set_variation

Purpose	Specify the variation range for a supply source.	
Syntax	**set_variation** **-supply** *supply_name_list* **-range {** *low_factor high_factor* **}**	
Arguments	**-supply** *supply_name_list*	A list of the names of the supply port, supply net, or supply set functions for which variation is being specified.
	-range { *low_factor high_factor* }	Variation factors with respect to the nominal voltage. Variation is expressed as multipliers that, when applied to the nominal voltage, give the low and high bounds of the variation range.
Return value	Return an empty string if successful or raise a TCL_ERROR if not.	

The **set_variation** command specifies how much a supply source may vary below and above its nominal voltage.

Nominal voltage values for supply set functions, supply ports, and supply nets may be specified in the definition of named power states for a supply set (see 6.5). Nominal voltage values of a supply port may be specified in the definition of named port states (see 6.4). The **set_variation** command defines variation factors with respect to nominal voltage for any supply object. Taken together, the nominal voltage and variation percentages define a voltage variation range for the specified supply. For example, {0.9 1.1} applied to a nominal voltage of 0.9 gives a variation range of 0.81 to 0.99.

Variation specified for a given supply object also applies to any electrically equivalent supply object. It shall be an error if different variation specifications are given for electrically equivalent supplies.

If variation is not specified for a given supply object or any electrically equivalent supply object, then the supply is assumed to have no variation, as if it were specified as **-range** {1.0 1.0}.

Syntax example

```
set_variation -supply { vss1 vss2 ground } -range { 0.95 1.05 }
```

6.54 upf_version

Purpose	Retrieves the version of UPF being used to interpret UPF commands and documents the UPF version for which subsequent commands are written.
Syntax	**upf_version** [*string*]
Arguments	*string* The UPF version for which subsequent commands are written.
Return value	Returns the version of UPF currently being used to interpret UPF commands.

The **upf_version** command returns a string value representing the UPF version currently being used by the tool reading the UPF file. When the UPF version defined by this standard is being used, the returned value shall be the string `"3.0"`. **upf_version** may also include an argument that documents the UPF version for which the UPF commands that follow were written. For UPF commands intended to be interpreted according to the UPF version defined by this standard, the argument shall be the string `"3.0"`.

This standard does not define any other value for the returned value of the **upf_version** command or for the *string* argument. This standard also does not define how a tool uses the specified UPF version argument; in particular, this standard does not define the meaning of a description consisting of UPF commands intended to be interpreted according to different UPF versions.

Syntax example

```
upf_version 3.0
```

6.55 use_interface_cell

Purpose	Specify the functional model and a list of implementation targets for isolation and level-shifting.	
Syntax	**use_interface_cell** *interface_implementation_name* **-strategy** *list_of_isolation_level_shifter_strategies* **-domain** *domain_name* **-lib_cells** *lib_cell_list* [**-port_map** **{{***port net_ref***}** ***}**] [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-applies_to_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-update_any** <**0** | **1** | **known** | **Z** | **latch** | *value*>] [**-force_function**] [**-inverter_supply_set** *list*]	
Arguments	*interface_ implementation_name*	The interface cell implementation strategy.
	-strategy *list_of_isolation_level_ shifter_strategies*	The isolation or level-shifter strategy, or a pair of isolation and level-shifter strategies, as defined by **set_isolation** and **set_level_shifter**.
	-domain *domain_name*	The domain in which the strategies are defined.
	-lib_cells *lib_cell_list*	A list of library cell names.
	-port_map **{{***port net_ref***}** ***}**	The *port* and the net (*net_ref)* connections.
	-elements *element_list*	A list of ports from the *list_of_isolation_level_shifter_strategies* to which the command applies.
	-exclude_elements *exclude_list*	A list of ports from the *list_of_isolation_level_shifter_strategies* to which this command does not apply.
	-applies_to_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	Only ports that have the specified clamp value are mapped.
	-update_any <**0** | **1** | **known** | **Z** | **latch** | *value*>	What is now the clamp value when **-applies_to_clamp** is **any**.
	-force_function	The first model in *lib_cell_list* is used as the functional specification of isolation behavior.
	-inverter_supply_set *list*	The supply set implicitly connected to any inversion logic required by an isolation signal connection.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **use_interface_cell** command provides user control for the integration of isolation and level-shifting. The command specifies the implementation choices through **-lib_cells** and the functional isolation behavior to be used if **-force_function** is specified.

Each cell specified in **-lib_cells** shall be defined by a **define_isolation_cell** (see 7.4) or **define_level_shifter_cell** (see 7.5) command or defined in the Liberty file with required attributes.

NOTE—Unlike **map_isolation_cell** and **map_level_shifter_cell, use_interface_cell** can be used to manually map any isolation, level-shifting, or combined isolation level-shifting cells. It may apply to an isolation strategy, a level-shifting strategy, or one of each.

When **-force_function** is specified the first model in *lib_cell_list* shall be used as the functional model. The isolation sense specification for the isolation strategy is ignored when **-force_function** is specified. It is erroneous if the functional model clamps to a value that is different to the previously specified port clamp value.

-elements selects the ports from the specified list of strategies to which the mapping command is applied. If **-elements** is not specified, all ports inferred from the list of strategies shall have the mapping applied. When **-applies_to_clamp** is specified, this command is applied only to the ports with that clamp value.

When **-applies_to_clamp** is **any**, **-update_any** shall be used to specify the clamp value after mapping. An **-update_any** value of **known** specifies that the isolation function is more complex than can be specified by a single value.

-port_map connects the specified *net_ref* to a *port* of the model. A *net_ref* may be one of the following:

a) A logic net name

b) A supply net name

c) One of the following symbolic references

   1) **isolation_supply.***function_name*

      function_name refers to the supply net corresponding to the function it provides to the **isolation_supply**.

   2) **isolation_supply**[*index*]**.***function_name*

      i) *index* is a non-negative integer corresponding to the position in the **isolation_supply** list specified for the isolation strategy.

      ii) The **isolation_supply** *index* shall be specified if the isolation strategy specified more than one **isolation_supply**.

   3) **isolation_signal**

      i) Refers to the isolation signal specified in the corresponding isolation strategy.

      ii) To invert the sense of the isolation signal, the SystemVerilog bit-wise negation operator ~ can be specified before the isolation_signal. The logic inferred by the negation shall be implicitly connected to the **inverter_supply_set** if specified, otherwise the **isolation_supply** shall be used.

   4) **isolation_signal**[*index*]

      i) *index* is a non-negative integer corresponding to the position in the **isolation_signal** list specified for the isolation strategy.

      ii) The **isolation_signal** *index* shall be specified if the isolation strategy specified more than one **isolation_signal**.

      iii) To invert the sense of the isolation signal, the SystemVerilog bit-wise negation operator ~ can be specified before the isolation_signal. If the **isolation_signal** is being inverted then the **inverter_supply_set[***index***]** if specified shall be implicitly connected to the inferred inverter, otherwise the **isolation_supply**[*index*] shall be used.

   5) **input_supply.***function_name*

      function_name refers to the supply net corresponding to the function it provides to the level-shifter **input_supply**.

6) **output_supply.**_function_name_

   _function_name_ refers to the supply net corresponding to the function it provides to the level-shifter **output_supply**.

7) **internal_supply.**_function_name_

   _function_name_ refers to the supply net corresponding to the function it provides to the level-shifter **internal_supply**.

The **-port_map** option shall not reference the data input port or the data output port. The input port shall be connected to the data input for the interface cell and the output port connected to the data output for the interface cell.

It shall be an error if:

— _domain_name_ does not indicate a previously created power domain.
— _list_of_isolation_level_shifter_strategies_ is an empty list.
— **-force_function** is not specified and none of the specified models in _lib_cell_list_ implements the functionality specified by the corresponding _isolation_strategy_ and port attributes.
— **-update_any** is specified and **-applies_to_clamp** is not **any**.
— After completing the _port_ and _net_ref_ connections and the data input and output connections, any port is unconnected.
— Ports specified by **-elements** are not included in all specified strategies.
— More than one isolation strategy is specified.
— More than one level-shifter strategy is specified.

Syntax example

```
use_interface_cell my_interface -strategy {ISO1 LS1} -domain PD1 \
  -lib_cells {combo1 combo2} \
  -elements {top/moduleA/port1 top/moduleA/port2 top/moduleA/port3}
```

7. Power-management cell definition commands

7.1 Introduction

Clause 7 documents the syntax for each UPF power-management cell command. A power-management cell is one of the following:

— "Always-on" cell
— Diode clamp
— Isolation cell
— Level-shifter cell
— Power-switch cell
— Retention cell

Power-management cell commands define characteristics of the instances of power-management cells used to implement and verify the power intent for a given design. These commands do not alter the existing library cell definitions and only have semantics when they are used with design power intent commands (see Clause 6).

Similar to how libraries are processed in a design flow, UPF power-management cell commands need to be processed before any other power intent commands and after the relevant cell libraries have been loaded.

It shall be an error if conflicting information is specified in multiple commands (of any type).

To understand the relationship between each UPF power-management cell command and its library cell definition in Liberty format, see Annex F.

7.2 define_always_on_cell

Purpose	Identify always-on cells.	
Syntax	**define_always_on_cell** **-cells** *cell_list* **-power** *power_pin* **-ground** *ground_pin* [**-power_switchable** *pin*] [**-ground_switchable** *pin*] [**-isolated_pins** *list_of_pin_lists* [**-enable** *expression_list*]]	
Arguments	**-cells** *cell_list*	Identifies the specified cells as always-on cells.
	-power *power_pin*	Identifies the power pin of the cell. If this option is specified with the **-power_switchable** option, it indicates this is a non-switchable power pin.
	-ground *power_pin*	Identifies the ground pin of the cell. If this option is specified with the **-ground_switchable** option, it indicates this is a non-switchable ground pin.
	-power_switchable *pin*	Specifies the power pin to be connected via a rail connection to the switchable power supply.
	-ground_switchable *pin*	Specifies the ground pin to be connected via a rail connection to the switchable ground supply.
	-isolated_pins *list_of_pin_lists*	Specifies a list of pin lists. Each pin list groups pins that are isolated internally with the same isolation control signal. These pin lists can only contain input pins.
	-enable *expression_list*	Specifies a list of simple expressions. Each simple expression describes the isolation control condition for the corresponding isolated pin list in the **-isolated_pins** option. If the internal isolation does not require a control signal, use an empty string for that pin list. The number of elements in this list shall correspond to the number of lists specified for the **-isolated_pins** option.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **define_always_on_cell** library command identifies the library cells having more than one set of power and ground pins that can remain functional even when the supply to the switchable power or ground pin is switched off.

NOTE—A cell called *always-on* does not mean the cell can never be powered off. When the supply to non-switchable power or ground of such cell is switched off, the cell becomes non-functional. In other words, the term always-on actually means *relatively always-on*.

By default, all input and output pins of this cell are related to the non-switchable power and ground pins.

Examples

The following example defines cell `aon_cell` as an always-on cell. The cell had three isolated pins: `pin1`, `pin2`, and `pin3`. Pins `pin1` and `pin2` have the same isolation control signal `iso1`, but `pin3` has no isolation control signal.

```
define_always_on_cell -cells aon_cell
   -isolated_pins { {pin1 pin2} {pin3}} -enable {!iso1 ""}
```

The following example defines cell `AND2_AON` as an always-on cell. The cell has two power pins and performs the `AND` function (as defined in the library) as long as the supply connected to power pin `VDD` is not switched off.

```
define_always_on_cell -cells AND2_AON -power_switchable VDDSW
   -power VDD -ground VSS
```

7.3 define_diode_clamp

Purpose	Identify diode cells or cells pins with diode protection.	
Syntax	**define_diode_clamp** **-cells** *cell_list* **-data_pins** *pin_list* [**-type** <**power** | **ground** | **both**>] [**-power** *pin*] [**-ground** *pin*]	
Arguments	**-cells** *cell_list*	Identifies cells as diode clamp cells or pins of the specified cells as diode clamp pins.
	-data_pins *pin_list*	Specifies a list of cell input pins that have built-in clamp diodes.
	-type <**power** | **ground** | **both**>	Specifies the type of clamp diode associated with the data pins. The type determines whether to use the power pin, ground pin, or both. Possible values are as follows: **both** indicates a power and ground clamp diode **ground** indicates a ground clamp diode **power** indicates a power clamp diode (the default)
	-power *pin*	Specifies the cell pin that connects to the power net.
	-ground *pin*	Specifies the cell pin that connects to the ground net.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **define_diode_clamp** library command identifies a list of library cells that are power cells, ground cells, or power and ground diode clamp cells, or complex cells that have input pins with built-in clamp diodes.

When **-type** is **ground**, then **-power** is optional. When **-type** is power, then **-ground** is optional. When **-type** is **both**, then both **-power** and **-ground** need to be specified as well.

It shall be an error if neither **-power** nor **-ground** is specified.

NOTE—The **define_diode_clamp** command is typically used for pins that have antenna protection diodes. Hence, this command may apply to regular non–power-managed cells.

Examples

The following command defines a cell `cellA` with diode protection at the pin `in1` where the diode is connected to the power pin `VDD1` of the cell.

```
define_diode_clamp -cells cellA -data_pins in1 -type power -power VDD1
```

7.4 define_isolation_cell

Purpose	Identify isolation cells.	
Syntax	**define_isolation_cell** **-cells** *cell_list* [**-power** *power_pin*] [**-ground** *ground_pin*] {**-enable** *pin* [**-clamp_cell** <**high** | **low**>] | **-pin_groups** **{{***input_pin output_pin* [*enable_pin*]**}***} | **-no_enable** <**high** | **low** | **hold**>} [**-always_on_pins** *pin_list*] [**-aux_enables** *ordered_pin_list*] [**-power_switchable** *power_pin*] [**-ground_switchable** *ground_pin*] [**-valid_location** <**source** | **sink** | **on** | **off** | **any**>] [**-non_dedicated**]	
Arguments	**-cells** *cell_list*	Identifies the specified cells as isolation cells.
	-power *power_pin*	Identifies the power pin of the cell. If this option is specified with the **-power_switchable** option for a multi-rail isolation cell, it indicates this is a non-switchable power pin.
	-ground *ground_pin*	Identifies the ground pin of the cell. If this option is specified with the **-ground_switchable** option for a multi-rail isolation cell, it indicates this is a non-switchable ground pin.
	-enable *pin*	Identifies the specified cell pin as the isolation enable pin. For non–clamp-type isolation cells, the enable pin polarity is determined by the cell function defined in the library files. For the special clamp-type cell identified by the **-clamp_cell** option, the enable polarity is active high if the clamp output is low and the enable polarity is active low if the clamp output is high. For a multi-rail isolation cell, the enable pin is related to the non-switchable power and ground pins of the cells.
	-clamp_cell <**high** | **low**>	Indicates the specified cells are isolation clamp cells. Such a cell, which consists of a single PMOS or NMOS transistor, does not perform any logic function and is only used to clamp a net to **high** or **low** when the enable pin is activated.
	-pin_groups **{{***input_pin output_pin* [*enable_pin*]**}***}	Specifies a list of input-output paths for multi-bit isolation cells. Each group in the list specifies one cell input pin, one cell output pin, and one optional enable pin that applies to the specified path. An enable pin may appear in more than one group. It shall be an error if the same input or output pin appears in more than one group.

	-no_enable <**high** | **low** | **hold**>	Specifies the following: The isolation cell does not have an enable pin. The output of the cell when the supply for the switchable power (or ground) pin is powered down. Possible values are as follows: **high** indicates the cell output is logic value `1` **low** indicates the cell output is logic value `0` **hold** indicates the cell output is the same as the logic value before the supply for the switchable power or ground is powered down
	-always_on_pins *pin_list*	Specifies a list of cell pins related to the nonswitchable power and nonswitchable ground pins of the cell.
	-aux_enables *ordered_pin_list*	Specifies additional or auxiliary enable pins for the isolation cell.
	-power_switchable *power_pin*	Identifies the switchable power pin of a multi-rail isolation cell.
	-ground_switchable *ground_pin*	Identifies the switchable ground pin of a multi-rail isolation cell.
	-valid_location <**source** | **sink** | **on** | **off** | **any**>	Specifies the valid location of the isolation cell. The default value is **sink**.
	-non_dedicated	Allows the specified cells to be used as normal cells, not for power management purposes.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **define_isolation_cell** library command identifies the library cells that can be used for isolation in a design with power gating.

By default, the output pin of a multi-rail isolation cell is related to the non-switchable power and ground pins. The non-enable input pin is related to the switchable power and ground pins. A *multi-rail isolation cell* is a cell with two power or ground pins.

If **-clamp_cell** is specified with value **high**, the only supply pin that can be specified is **-power**. If **-clamp_cell** is specified with **low**, the only supply pin that can be specified is **-ground**. For all other isolation cells, both **-power** and **-ground** shall be specified.

The **-aux_enables** option specifies additional or auxiliary enable pins for the isolation cell. By default, all pins specified in this option are related to the switchable power or ground pin. The list is an ordered list and each element can be accessed by using index starting at `1`, where the isolation enable pin specified in the **-enable** option is assumed to be index `0`.

If an auxiliary enable pin is related to the non-switchable power or ground, that pin shall also be specified using the **-always_on_pins** option. The logic that drives this pin shall be on when the isolation enable is asserted at pin specified by the **-enable** option.

The **-valid_location** option specifies the valid location of the isolation cell, as follows:

a) **source**—indicates the cell shall be inserted in a location where the primary supply set is equivalent to the driving supply set for a net requiring isolation. Such cells are typically multi-rail isolation cells and used for off-to-on isolation. It typically relies on its switchable power and ground supply for its normal function and on its non-switchable power or ground supply to provide the isolation function. See item d) for **off** value for special cases.

b) **sink**—indicates the cell shall be inserted in a location where the primary supply set is equivalent to the receiving supply set for a net requiring isolation. Such cells are typically single-rail isolation cells and used for off-to-on isolation.

c) **on**—indicates the cell can only be inserted in the location where the primary supply set is equivalent to either the driving supply set or the receiving supply for a net requiring isolation and the primary supply set is not switched off when the isolation function is needed. When used for off-to-on isolation, it is equivalent to **sink**. Such cells are typically single-rail isolation cells.

d) **off—**indicates the cell can be inserted in a location where the primary supply set is equivalent to either the driving supply set or the receiving supply for a net requiring isolation and the primary supply set may be switched off when the isolation function is needed. When used for off-to-on isolation, it is equivalent to **source**. Such cells are typically multi-rail isolation cells.

NOTE—Some single-rail isolation cells with special circuit structure can also be used in the switched-off domain. For example, a single-rail `NOR` gate can be placed in a power-switched-off domain for off-to-on isolation with an output value `low`; a single-rail `NAND` gate can be placed in the ground switched-off domain for off-to-on isolation with an output value `high`.

e) **any**—indicates the cell can be placed in any location. Such cells are typically multi-rail isolation cells. In addition, this cell is designed in a way that neither its normal function nor its isolation function relies on the primary supply of the domain in which it is located. Therefore, this type of cell can be used for off-to-on or on-to-off isolation.

Examples

The following isolation cell can be placed in any location for a design that uses ground switches for shutoff. `VDD` is the rail pin for power connection and `GVSS` is the ground pin for non-switchable ground connection. This cell does not have a rail pin for ground connection.

```
define_isolation_cell -cells iso_cell1 -power VDD -ground GVSS
     -enable iso_en -valid_location any
```

The following examples illustrate the use of the **-pin_groups** option to specify multi-bit isolation cells with two paths:

```
define_isolation_cell -cells mbit_iso1 -pin_groups { { datain1 dataout1
    iso1 } { datain2 dataout2 iso2 } }
    -power VDD -ground VSS -valid_location sink
define_isolation_cell -cells mbit_iso2 -pin_groups { { datain1 dataout1 }
    { datain2 dataout2} }
    -power VDD -ground VSS -valid_location sink
```

For cell `mbit_iso1`, there are two isolation paths. The first is from data input `datain1` to output `dataout1` with `iso1` as the isolation enabler. The second is from data input `datain2` to output `dataout2` with `iso2` as the isolation enabler.

For cell `mbit_iso2`, there are also two isolation paths. However, this special isolation cell has no isolation enabler to control each path. As a result, there is no isolation enable signal defined in each group.

7.5 define_level_shifter_cell

Purpose	Identify level-shifter cells.	
Syntax	**define_level_shifter_cell** **-cells** *cell_list* [**-input_voltage_range** {{*lower_bound upper_bound*}*}] [**-output_voltage_range** {{*lower_bound upper_bound*}*}] [**-ground_input_voltage_range** {{*lower_bound upper_bound*}*}] [**-ground_output_voltage_range** {{*lower_bound upper_bound*}*}] [**-direction** <**low_to_high** | **high_to_low** | **both**>] [**-input_power_pin** *power_pin*] [**-output_power_pin** *power_pin*] [**-input_ground_pin** *ground_pin*] [**-output_ground_pin** *ground_pin*] [**-ground** *ground_pin*] [**-power** *power_pin*] [**-enable** *pin* | **-pin_groups** {{*input_pin output_pin* [*enable_pin*]}*}] [**-valid_location** <**source** | **sink** | **either** | **any**>] [**-bypass_enable** *expression*] [**-multi_stage** *integer*]	
Arguments	**-cells** *cell_list*	Identifies the specified cells as level-shifter cells.
	input_voltage_range {{*lower_bound upper_bound*}*}	Identifies a list of voltage ranges for the input (source) supply voltage that can be handled by this level-shifter. The voltage range shall be specified as follows: `{lower_bound upper_bound}` This option shall only be specified for power-shifting cells.
	-output_voltage_range {{*lower_bound upper_bound*}*}	Identifies a list of voltage ranges for the output (destination) power supply voltage that can be handled by this level-shifter. The voltage range shall be specified as follows: `{lower_bound upper_bound}` This option shall only be specified for power-shifting cells.
	-ground_input_voltage_range {{*lower_bound upper_bound*}*}	Identifies a list of voltage ranges for the input (source) ground supply voltage that can be handled by this level-shifter. The voltage range shall be specified as follows: `{lower_bound upper_bound}` This option should only be specified for ground-shifting cells.
	-ground_output_voltage_range {{*lower_bound upper_bound*}*}	Identifies a list of voltage ranges for the output (destination) ground supply voltage that can be handled by this level-shifter. The voltage range shall be specified as follows: `{lower_bound upper_bound}` This option shall only be specified for ground-shifting cells.
	-direction <**low_to_high** | **high_to_low** | **both**>	Specifies whether the level-shifter can be used between a driver with lower voltage swing and a receiver with higher voltage swing (**low_to_high**), or vice versa (**high_to_low**), or both (**both**). The *voltage swing* is simply the difference between the power voltage and ground voltage. The default is **low_to_high**.
	-input_power_pin *power_pin*	Identifies the input power pin. This option is usually specified for power shifting and used with **-output_power_pin**.
	-output_power_pin *power_pin*	Identifies the output power pin. This option is usually specified for ground shifting and used with **-input_power_pin**.
	-input_ground_pin *ground_pin*	Identifies the input ground pin. This option is usually specified for ground shifting and used with **-output_ground_pin**.

	-output_ground_pin *ground_pin*	Identifies the output ground pin. This option is usually specified for ground shifting and used with **-input_ground_pin**.
	-ground *ground_pin*	Identifies the ground pin of the cell. This option can only be specified for level-shifters that only perform power shifting. In other words, it shall be an error to use this option with **-input_ground_pin** and **-output_ground_pin**.
	-power *power_pin*	Identifies the power pin of the cell. This option can only be specified for level-shifters that only perform ground shifting. In other words, it shall be an error to use this option with **-input_power_pin** and **-output_power_pin**.
	-enable *pin*	Identifies the pin that prevents internal floating when the power supply of the originating power domain is powered down, but the output voltage level power pin remains on. The related power and ground of this pin is the output power and ground pins defined for this cell.
	-pin_groups **{{***input_pin output_pin* [*enable_pin*]**}*}**	Specifies a list of input-output paths for multi-bit level-shifter cells. Each group in the list specifies one cell input pin, one cell output pin, and one optional enable pin that applies to the specified path. An enable pin may appear in more than one group. It shall be an error if the same input or output pin appears in more than one group.
	-valid_location <**source** | **sink** | **either** | **any**>	Specifies the valid location of the level-shifter cell. The default value is **sink**.
	-bypass_enable *expression*	Specifies when to bypass the voltage shifting functionality. When the expression evaluates to *True*, the cell behaves like a buffer. The expression shall be a simple expression of the bypass enable input pin. By default, the related power and ground of this pin is the output power and ground pin defined for this cell.
	-multi_stage *integer*	Identifies the stage of a multi-stage level-shifter to which this definition (command) applies. For a level-shifter cell with *N* stages, *N* definitions shall be specified for the same cell. Each definition needs to associate a number from `1` to *N* for this option. For more information, see Annex G.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **define_level_shifter_cell** library command identifies the library cells to use as level-shifter cells, as follows:

— If **-input_voltage_range** is specified, the **-output_voltage_range** shall also be specified.

— If **-ground_input_range** is specified, the **-ground_output_range** shall also be specified.

— It shall be an error if neither **-input_voltage_range** nor **-ground_input_voltage_range** is specified.

If a list of voltage ranges is specified for the input supply voltage, a list of voltage ranges for the output supply voltage with the same number of elements shall also be specified., i.e., each member in the list of input voltage ranges needs to have a corresponding member in the list of output voltage ranges.

By default, the enable and output pins of this cell are related to the output power and output ground pins (specified through the **-output_power_pin** and **-output_ground_pin** options). And the non-enable input pin is related to the input power and input ground pins (specified through the **-input_power_pin** and **-input_ground_pin** options).

The **-valid_location** option specifies the valid location of the level-shifter cell, as follows:

a) **source**—indicates the cell shall be inserted in a location where the primary supply set is equivalent to the driving supply set for a net requiring level-shifting.

b) **sink**—indicates the cell shall be inserted in a location where the primary supply set is equivalent to the receiving supply set for a net requiring level-shifting.

c) **either**—indicates the cell shall be inserted in a location where the primary supply set is equivalent to the driving supply set or the receiving supply set for a net requiring level-shifting.

d) **any**—indicates the cell can be placed in any location.

    1) If the cell contains pins for rail connection, these pins shall not be specified through the **-input_power_pin**, **-output_power_pin**, **-input_ground_pin**, or **-output_ground_pin** options.

    2) A power level-shifter with this setting can be placed in any location as long as its primary ground net is equivalent to the driving and receiving primary ground net of the net requiring level-shifting.

    3) A ground level-shifter with this setting can be placed in any location as long as its primary power net is equivalent to the driving and receiving primary power net of the net requiring level-shifting.

    4) For a power and ground level-shifter, which requires two definitions of the command—one for the power part and one for the ground part of the cell—the **-valid_location** can be different in the two definitions:

Power part	**Ground part**
`any`	`source|sink|either`
`source|sink|either`	`any`
`any`	`any`

i) In the first case, the ground-shifting part of the level-shifter definition determines the location.

ii) In the second case, the power-shifting part of the level-shifter definition determines the location.

iii) In the third case, the cell can be placed in a domain whose power and ground supplies are neither driving the logic power and ground supplies nor receiving the logic power and ground supplies.

Examples

The following example identifies level-shifter cells with one power pin and one ground pin that perform power shifting from 1.0 V to 0.8 V.

```
define_level_shifter_cell
-cells LSHL
-input_voltage_range {{1.0 1.0}} -output_voltage_range {{0.8 0.8}}
-direction high_to_low
-input_power_pin VH -ground G
```

The following example identifies level-shifter cells that perform power shifting from 0.8 V to 1.0 V. In this case, the level-shifter cells need to have two power pins and one ground pin.

```
define_level_shifter_cell
-cells LSLH
-input_voltage_range {{0.8 0.8}} -output_voltage_range {{1.0 1.0}}
-direction low_to_high
-input_power_pin VL -output_power_pin VH -ground G
```

The following example identifies level-shifter cells with valid location `any` to perform voltage shifting from 0.8 V to 1.0 V. The cells have three power pins and one ground pin.

`VDD`—This is the standard cell rail; this pin is not used by the cell.

`VDDL`—This is the power pin to which the input signal is related.

`VDDH`—This is the power pin to which the output signal is related.

`VSS`—This is the ground pin of the cell.

```
define_level_shifter_cell
-cells LSLH
-direction low_to_high
-input_voltage_range {{0.8 0.8}} -output_voltage_range {{1.0 1.0}}
-input_power_pin VDDL -output_power_pin VDDH -ground VSS
-valid_location any
```

The following example identifies level-shifter cells that perform both power shifting from 0.8 V to 1.0 V and ground shifting from 0.2 V to 0 V. In this case, the level-shifter cells need to have two power pins and two ground pins. In addition, since the input voltage swing is 0.6 V (0.8 V to 0.2 V), which is smaller than the output voltage swing of 1.0 V (1.0 V to 0 V), the direction of the cell is `low_to_high`.

```
define_level_shifter_cell
-cells LSLH
-input_voltage_range {{0.8 0.8}} -output_voltage_range {{1.0 1.0}}
-ground_input_voltage_range {{0.2 0.2}} -ground_output_voltage_range {{0.0
 0.0}}
-direction low_to_high
-input_power_pin VL -output_power_pin VH
-input_ground_pin GH -output_ground_pin GL
```

The following example indicates the level-shifter can shift from 0.8 V to 1.0 V or from 1.0 V to 1.2 V. However, the cell cannot shift power voltage from 0.8 V to 1.2 V.

```
define_level_shifter_cell
-cells LSLH
-input_voltage_range {{0.8 1.0}} -output_voltage_range {{1.0 1.2}}
-input_power_pin VL -output_power_pin VH -ground_pin VSS
-direction low_to_high
```

The following example indicates the level-shifter can shift from input range 0.8 V to 0.9 V to output range 1.0 V to 1.1 V, or from input range 0.9 V to 1.0 V to output range 1.1 V to 1.2 V. Note that the cell cannot shift input voltages between 0.8 V to 0.9 V to output voltages 1.1 V to 1.2 V.

```
define_level_shifter_cell
-cells LSLH -input_power_pin VL -output_power_pin VH -ground_pin VSS
-input_voltage_range {{0.8 0.9} {0.9 1.0}}
-output_voltage_range {{1.0 1.1} {1.1 1.2}}
-direction low_to_high
```

The following examples illustrate the use of the **-pin_groups** option to specify multi-bit level-shifter cells with and without `enable`:

```
define_level_shifter_cell -cells mbit_en_ls -pin_groups { { datain1
   els_dataout1 en1 } {datain2 els_dataout2 en2 } }
define_level_shifter_cell -cells mbit_ls -pin_groups { { datain1
   ls_dataout1 } { datain2 ls_dataout2 } }
```

7.6 define_power_switch_cell

Purpose	Identify a power switch or ground-switch cell.	
Syntax	**define_power_switch_cell** **-cells** *cell_list* **-type** <**footer** | **header**> **-stage_1_enable** *expression* [**-stage_1_output** *expression*] {**-power_switchable** *power_pin* **-power** *power_pin* | **-ground_switchable** *ground_pin* **-ground** *ground_pin*]} [**-stage_2_enable** *expression* [**-stage_2_output** *expression*]] [**-always_on_pins** *ordered_pin_list*] [**-gate_bias_pin** *power_pin*]	
Arguments	**-cells** *cell_list*	Identifies the specified cells as power-switch cells.
	-type <**footer** | **header**>	Specifies whether the power-switch cell is a **header** or **footer** cell.
	-stage_1_enable (**-stage_2_enable**) *expression*	Specifies when the switch cell driven by this input pin is turned on (enabled) or off. If only stage 1 is specified, the switch is turned on when the expression for the **-stage_1_enable** option evaluates to *True* and the switch is turned off when the expression for the **-stage_1_enable** option evaluates to *False*. If both stages are specified, the switch is turned on when the expression for both enable options evaluates to *True* and the switch is turned off when the expression for both enable options evaluates to *False*. The Boolean expression is a simple expression of the input pin.
	-stage_1_output (**-stage_2_output**) *expression*	Specifies whether the output pin in the expression is the buffered or inverted output of the input pin specified through the corresponding **-stage_*x*_enable** option. In a design, this pin is used to connect another switch cell in series to form a power-switch chain.
	-power_switchable *power_pin*	Identifies the output power pin in the corresponding cell. This option can only be used if the power gating cell is used to cut off the path from power to ground on the power side (i.e., for a header cell). This pin shall be connected to a switchable power net.
	-power *power_pin*	Identifies the input power pin of the cell.
	-ground_switchable *ground_pin*	Identifies the output ground pin in the corresponding cell. This option can only be used if the power gating cell is used to cut off the path from power to ground on the ground side (i.e., for a footer cell). This pin shall be connected to a switchable ground net.
	-ground *power_pin*	Identifies the input ground pin of the cell.
	-always_on_pins *ordered_pin_list*	Specifies a list of cell pins related to the input power and ground pins of the cell.
	-gate_bias_pin *power_pin*]	Identifies a power pin that provides the supply used to drive the gate input of the switch cell.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **define_power_switch_cell** library command identifies the library cells to use as power-switch cells. The input enable and output enable pins of these cells are related to the non-switchable power and ground pins.

Examples

The following example defines a header power switch. The power switch has two stages. The power switch is completely on if the transistors of both stages are on. The stage 1 transistor is turned on by applying a low value to input `I1`. The output of the stage 1 transistor, `O1`, is a buffered output of input `I1`. The stage 2 transistor is turned on by applying a high value to input `I2`. The output of stage 2 transistor, `O2`, is the inverted value of input `I2`.

```
define_power_switch_cell -cells 2stage_switch -stage_1_enable !I1
-stage_1_output O1 -stage_2_enable I2 -stage_2_output !O2 -type header
```

7.7 define_retention_cell

Purpose	Identify state retention cells.	
Syntax	**define_retention_cell** **-cells** *cell_list* **-power** *power_pin* **-ground** *ground_pin* [**-cell_type** *string*] [**-always_on_pins** *pin_list*] [**-restore_function** {{**pin** <**high** | **low** | **posedge** | **negedge**}}] [**-save_function** {{**pin** <**high** | **low** | **posedge** | **negedge**}}] [**-restore_check** *expression*] [**-save_check** *expression*] [**-retention_check** *expression*] [**-hold_check** *pin_list*] [**-always_on_components** *component_list*] [**-power_switchable** *power_pin*] [**-ground_switchable** *ground_pin*]	
Arguments	**-cells** *cell_list*	Identifies the specified cells as state retention cells.
	-power *power_pin*	Identifies the power pin of the cell. If this option is specified with the **-power_switchable** option, it indicates this is a non-switchable power pin.
	-ground *ground_pin*	Identifies the ground pin of the cell. If this option is specified with the **-ground_switchable** option, it indicates this is a non-switchable ground pin.
	-cell_type *string*	Specifies a user-defined name grouping the specified cells into a class of retention cells that all have the same retention behavior. This specification limits the group of cells that can be used to those requested through the **-lib_cell_type** option of the **map_retention_cell** command (see 6.35).
	-always_on_pins *pin_list*	Specifies a list of cell pins that are related to the nonswitchable power and ground pins of the cells.
	-restore_function {{**pin** <**high** | **low** | **posedge** | **negedge**}}	Specifies the polarity or the edge sensitivity of the restore pin that enables the retention cell to restore the saved value after exiting power shut-off mode. By default, the restore pin relates to the non-switchable power and ground pin of the cell. If not specified, the restore event is triggered when the primary power is restored, or the power-up event. When neither **-save_function** nor **-restore_function** is specified, the current value is always saved before entering retention mode and the saved value is restored when the primary power is restored.
	-save_function {{**pin** <**high** | **low** | **posedge** | **negedge**}}	Specifies the polarity or the edge sensitivity of the save pin that enables the retention cell to save the current value before entering retention mode. By default, the save pin relates to the non-switchable power and ground pin of the cell. If not specified, the save event is triggered by the negation of the restore function when it is specified. When neither **-save_function** nor **-restore_function** is specified, the current value is always saved before entering retention mode and the saved value is restored when the primary power is restored.
	-restore_check *expression*	Specifies the additional condition when the states of the sequential elements can be restored. The expression shall be a function of the cell input pins. The expression shall be *True* when the restore event occurs.
	-save_check *expression*	Specifies the additional condition when the states of the sequential elements can be saved. The expression shall be a function of the cell input pins. The expression shall be *True* when the save event occurs.

	-retention_check *expression*	Specifies an additional condition to meet (after the primary supply of the retention cell is switched off and before the supply is powered on again) for the retention operation to be successful. The *expression* can be a Boolean function of cell input pins. The *expression* shall be *True* when the primary supply set of the power domain in which the retention logic is located, is shut off and the retention supply set is on.
	-hold_check *pin_list*	Specifies a list of pins that maintain the same logic value during the retention period, from the time when the save event occurs to the time when the restore event occurs. The pin may be the clock pin or any other control pin.
	-always_on_ components *component_list*	Specifies a list of component names: instances, named processes, state elements, or signal names, in the corresponding simulation model that are powered by the nonswitchable power and ground pins. The logic values of the specified components are corrupted if the state value of the non-switchable power and group pin is `OFF`. NOTE—The option has only an impact on tools that use the gate-level simulation models of state retention cells.
	-power_switchable *power_pin*	Identifies the switchable ground pin. This cell can be used for retention purpose in a power domain that can be shutoff using power switches (i.e., using a header cell).
	-ground_switchable *ground_pin*	Identifies the switchable ground pin. This cell can be used for retention purpose in a power domain that can be shutoff using ground switches (i.e., using a footer cell).
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **define_retention_cell** library command identifies the library cells to use as retention cells. The following also apply:

— By default, all pins of this cell are related to the switchable power and ground pins, unless otherwise specified.

— It shall be an error if the save and restore functions both identify the same pin, and the polarity or edge sensitivity are the same for that pin. For example, the following two commands are incorrect:

```
define_retention_cell -cells My_Ret_Cell1
  -restore_function {pg high} -save_function {pg high}
define_retention_cell -cells My_Ret_Cell2
  -restore_function {pg posedge} -save_function {pg posedge}
```

— It shall be an error if the conditions specified in **-save_check**, **-restore_check**, or **-retention_check** conflict with **-hold_check**. For example, the specification:

```
-hold_check clk -save_check !clk -restore_check clk
```

shall be an error since the `-hold_check` requires the `clk` signal to hold the same value from the time when the save event occurs to the time when the restore event occurs, but the other two options require the signal `clk` have different values.

NOTE—If the cell data output pin is listed in the **-always_on_pins** list, then this retention cell may be used for retention strategies that specify **-use_retention_as_primary**.

Example

In the following example, the cell design requires clock `clk` be held to `0` to save or restore the state of the sequential element. If retention control pin `save` is set to `0`, the state will be saved and saved data will be restored when the primary power `VDD` is restored. The retention power `VDDC` shall be on to enable the retention while `VDD` is switched off.

```
define_retention_cell -cells My_Ret_Cell -power VDDC
-ground VSS -power_switchable VDD
-save_check {!clk} -restore_check {!clk}
-save_function {save negedge}
```

8. UPF processing

8.1 Overview

All UPF commands have an immediate effect when they are executed by a Tcl interpreter. For the following commands, the immediate effect is the only effect:

— **add_parameter** (see 6.3)

— **apply_power_model** (see 6.9)

— **begin_power_model** (see 6.11)

— **create_hdl2upf_vct** (see 6.17)

— **create_upf2hdl_vct** (see 6.27)

— **end_power_model** (see 6.29)

— **find_objects** (see 6.30)

— **load_simstate_behavior** (see 6.31)

— **load_upf** (see 6.32)

— **set_correlated** (see 6.39)

— **set_design_attributes** (see 6.40)

— **set_design_top** (see 6.41)

— **set_equivalent** (see 6.43)

— **set_partial_on_translation** (see 6.46)

— **set_port_attributes** (see 6.47)

— **set_scope** (see 6.51)

— **set_simstate_behavior** (see 6.52)

— **set_variation** (see 6.53)

— **upf_version** (see 6.54)

All other UPF commands have both an immediate and a deferred effect. For these commands, the immediate effect is to add the command syntax to an internal structure for further processing. The deferred effect varies with the command, but typically contributes to construction of a power intent model reflecting the specification. This model is then applied to the design as appropriate for the tool involved.

One exception is the **save_upf** command (see 6.38), for which the deferred effect is generation of a UPF file describing the power intent for a given scope. This generation occurs after the power intent model has been fully constructed, so the generated UPF file is complete.

NOTE—This algorithm defines a reference model for UPF command processing, to illustrate how the interdependencies between design data and the UPF specification, and among UPF commands themselves, can be satisfied. A given tool may use a different algorithm as long as the overall effect is the same as this algorithm would present.

8.2 Data requirements

In addition to the UPF file(s) involved, UPF processing requires access to the following data:

- Elaborated design hierarchy
- UPF attribute specifications in HDL (if any)
- Library cell definitions

These data need to be available when UPF processing begins.

8.3 Processing phases

8.3.1 Overview

Before UPF processing begins, information from Liberty models is imported into the UPF context as follows:

For each instance in the design for which a corresponding Liberty model is available:

- For each pin of the Liberty model with attribute pgtype, if the instance does not have a port of the same name, then such a port is implicitly created for that instance in the HDL design.
- For each attribute of the Liberty model with a corresponding UPF predefined attribute, an equivalent UPF **set_design_attributes** or **set_port_attributes** command is prepended to the top-level UPF file.

The following algorithm describes the detailed sequence of operations to process a UPF description, extract the power intent it specifies, and apply the power intent to a design for use in a verification or implementation tool.

The current context initially consists of the top-level (design top model, design top instance, current scope), and the top-level UPF file (prepended with imported Liberty attributes).

Phase 1 (and conditionally phase 2) is executed by reading and executing UPF commands, as follows:

a) If the command is **load_upf** or **apply_power_model** then
   1) the design top model, design top instance, and current scope variables are changed to the new context, and

2) Phase 1 is applied recursively to process the new context, and then

3) the context reverts to the parent context.

Otherwise:

4) the command is interpreted in the current context.

b) If the current design top model has attribute {**UPF_is_soft_macro** TRUE} defined on it, then

   1) Phase 2 is executed to build the power intent model for the current design top instance, down to but not including any leaf instances below the current design top.

Phase 3 is executed for the whole design.

Phase 4 is executed for the whole design.

8.3.2 Phase 1—read power intent specification

In this phase, the UPF commands are parsed and further processed to create a normalized representation of the UPF specification. This involves the following operations:

a) Read UPF commands and execute the immediate effect of each UPF command as it is read in:

   1) For the **create_supply_port** and **create_logic_port** commands, which create named objects in the design: if the port name is not already defined in the current scope in the HDL design hierarchy, then define the port in the current scope in the HDL design hierarchy.

   2) For **set_port_attributes**/**set_design_attributes** commands, which define attributes of objects in the HDL design: associate such attribute definitions immediately with those objects in the HDL design hierarchy.

   3) For commands that refer to objects in the design by name: resolve references to the design relative to the current scope in the HDL design hierarchy.

   4) For the **find_objects** command, which searches for objects in the design hierarchy based on search criteria: execute the **find_objects** command in the current scope of the HDL design hierarchy, taking into account names defined in 1) above and attributes defined in 2) above.

   5) For all other commands: execute their immediate effect as appropriate.

   6) For any command that has a deferred effect: add the command to the syntactic model of the UPF specification.

b) Collapse **-update** commands in the syntactic model of the UPF specification and check for conflicts.

c) Apply defaults for defaultable options.

In general, names shall be defined before being referenced. In this phase, name-defining UPF commands are associated with the scope in which the object is defined, or with the parent object for which a subordinate object is defined, as appropriate, so that subsequent name references can be resolved at this stage.

Names of design objects referenced in UPF commands shall be defined in the design hierarchy before they are referenced in UPF. Names of the library cells referenced in UPF commands shall be defined for the design before they are referenced in UPF. Names of UPF-defined objects shall be defined and associated with the appropriate design hierarchy scope before they are referenced in UPF. Names of objects that are associated with other objects (supply set handles of power domains; functions of supply sets or supply set handles; port states of ports; power states of supply sets, power domains, or modules; simstates of power states) shall be defined and associated with the relevant parent object before they are referenced in UPF. Names of VCTs shall be defined in UPF and associated with the global VCT scope before they are referenced in UPF.

Any command that updates a previous command that defined a simple name in a design hierarchy scope shall be processed in the scope in which the original command was processed and be associated with that same scope. Any command that updates a previous command that defined an object associated with a parent object shall also be processed in the scope in which the original command was processed and be associated with that same parent object.

8.3.3 Phase 2—build power intent model

In this phase, the normalized UPF specification is executed to construct a model of the power intent expressed by the specification. This involves the following operations:

a) Construct power domains:
    1) As specified by **create_power_domain** commands (see 6.20).
    2) Using the effective element list algorithm in 5.9.
    3) Including constructing required supply sets and functions.
    4) Atomic power domains shall be constructed first, followed by non-atomic power domains.
b) Construct control logic for isolation, retention, and switch instances as specified by **create_logic_*** (see 6.18 and 6.19) and **connect_logic_net** (see 6.13) commands.
c) Construct supply networks and connections to power domains/strategies:
    1) As specified by **create_supply_*** (see 6.24, 6.25, and 6.26) and **create_power_switch** (see 6.21) commands.
    2) **connect_supply_*** (see 6.14 and 6.15), **create_*_vct** (see 6.17 and 6.27), and **associate_supply_set** (see 6.10) commands, including:
        i) Equivalent supply declarations
        ii) Error checks related to supply set/function association
        iii) Implicit associations of supply nets and logic nets with power switch ports, as defined in **create_power_switch** commands
d) Construct explicit, implicit, and automatic supply connections as specified by **connect_supply_*** commands (see 6.14 and 6.15), **associate_supply_set** (see 6.7), etc.
e) Apply the power model of a hard IP cell as specified by **apply_power_model** command (see 6.9).
f) Construct composite domains:
    1) As specified by **create_composite_domain** (see 6.16) commands
    2) Including propagation of primary supply to/among subdomains
    3) Including error checks related to domain composition
g) Identify power-domain boundary ports and their supplies by analyzing the elaborated design and **create_power_domain** (see 6.20) commands.
h) Apply retention strategies for each domain as specified by **set_retention** (see 6.49 and 4.6.7).

i) Apply repeater strategies for each domain as specified by **set_repeater** (see 6.48 and 4.6.7).

j) Apply isolation strategies for each domain boundary port as specified by **set_isolation** (see 6.44 and 4.6.7).

k) Apply level-shifting strategies for each domain boundary port as specified by **set_level_shifter** (see 6.45 and 4.6.7).

l) Identify cells to use for isolation, level-shifting, retention, and switch elements as specified by **map_*** (see 6.34 and 6.35) and **use_interface_cell** (see 6.55) commands.

m) Construct power states as specified by **add_power_state** (see 6.5) commands.

n) Construct power state transitions as specified by **add_state_transition** (see 6.7) commands.

8.3.4 Phase 3—recognize implemented power intent

In this phase, the **-instance** options of all commands are processed to identify instances of cells that implement the power intent. If a given command has a **-instance** option, this indicates that the command has been implemented by some preceding step in the flow. The implementation may or may not be complete. In particular, new logic added to the design by some tool step (e.g., for test insertion) may trigger further implementation through another application of the same command.

If a given command has a **-instance** option that specifies an empty string as the instance name, this indicates the instance resulting from applying the command in this particular context has been optimized away. In this case, tools shall not infer a cell for this application of the command. In particular, verification tools shall not infer a cell for purposes of verification, and implementation tools shall not re-implement the command by inserting a cell again.

If a given command has a **-instance** option that specifies a hierarchical name as the instance name, the specified instance shall exist in the design. It shall be an error if that hierarchical name does not identify a cell instance of the appropriate type for the command. Attributes specified in library cells, in HDL models, or in UPF may be used to determine whether a given cell instance is appropriate for the command whose **-instance** option identifies it as resulting from the implementation of that command. In this case also, tools shall not infer a cell for this application of the command. Instead, the existing cell shall be used.

In addition to the preceding, commands that create supply or logic ports or nets are processed to identify any ports or nets that already exist in the HDL hierarchy. If a **create_supply_port** (see 6.25), **create_supply_net** (see 6.24), **create_logic_port** (see 6.19), or **create_logic_net** (see 6.18) command specifies a port or net name that already exists in the current scope of the HDL hierarchy, it shall be an error if that port or net name does not identify a port or net, respectively, of the appropriate type for the command. A supply port or net is appropriate for a **create_supply_port** or **create_supply_net** command, respectively, if it is declared to be of type `supply_net_type` defined in the package UPF. A logic port or net is appropriate if it is declared with the standard logic type in the relevant HDL. In this case also, tools shall not create a new port or net for this application of the command. Instead, the existing port or net shall be used.

8.3.5 Phase 4—apply power intent model to design

In this phase, some or all of the power intent model is applied to the HDL design. A given tool shall add the power intent elements required for that tool's operation to the design model. Power intent model elements that are already present in the design shall not be added again. This includes implementation of any checkers introduced by the **bind_checker** command (see 6.12).

NOTE—It may be appropriate for a given tool to update existing elements in the design to more completely reflect the power intent model. For example, a tool may choose to change the data type of a net in the design used as a supply net, from a single-bit type to the appropriate (SystemVerilog or VHDL) `supply_net_type`.

8.3.6 Phase 5—query power intent model

In this phase, power intent model data can be queried via the information model API (see Clause 10). This API consists of Tcl-based (see 11.1) and HDL-based (see 11.2) UPF query commands. Any checkers resulting from new bind_checker commands (see 6.12) introduced in this phase shall be implemented in this phase.

8.4 Error checking

Error checking is done in various UPF-processing stages. Error checks include the following classes of checks, which would be performed in Phases 1, 2, and 3 of UPF processing:

a) Phase 1—Read and resolve UPF specification (see 8.3.2)
   1) UPF syntax checks (including semantic restrictions)
   2) Update conflict checks
   3) Design scope/object reference checks (scope/object not found)
b) Phase 2—Build power intent model (see 8.3.3)
   1) Conflicts between two commands applying to same object
   2) Completeness checks (e.g., all instances are in a power domain)
c) Phase 3—Identify implemented power intent (see 8.3.4)
   1) Name conflicts (an existing design object conflicts with a UPF name)

If a tool detects and reports an error in any of the preceding UPF-processing phases, the tool may continue processing if possible, in order to identify any additional errors that might exist in the UPF specification or its interpretation with the design hierarchy, but processing should terminate before phase 4, where the power intent model is applied to the design hierarchy.

9. Simulation semantics

9.1 Supply network creation

UPF supply network creation commands define the power supply network that connects power supplies to the instances in a design. After these commands are applied, every instance in a design is connected to the power supply network. The *supply network* is a set of supply nets, supply ports, switches, and potentially, regulators and generators. Supply sets are defined in terms of supply nets and conveniently define a complete power circuit for instances. Supply sets simplify the management of related supply nets and facilitate connections based on the role the supply set provides for a power domain and the functions the supply nets provide within the set (see 9.2.2). The supply network defines how power sources are distributed to the instances and how that distribution is controlled.

A supply port that propagates but does not originate a supply state and voltage value defines a *supply source*. At any given time, a supply source can be traced through the supply network connectivity to a single root supply driver. The output port of a switch is a root supply driver; the value of this driver is computed according to the algorithm given in 6.21. A power switch, voltage regulator, or bias generator modeled in HDL should be modeled as a separate component with an output port that acts as a root supply driver to provide power to other components.

Determination of the root supply driver is required for certain supply network resolution functions (see 6.24).

NOTE—Since the supply net type is defined in the package UPF, it is possible to create the supply network entirely in HDL source.

A supply net can be connected to a port declared in the HDL description. In this case, the supply net state is connected to the port; the voltage is not used. VCTs define the conversion from supply net state values to values of an HDL type and vice versa to facilitate more complex modeling consistent with an organization's logic value interpretations of UPF supply port states.

If a supply net is connected to a HDL port of a single bit type, a default VCT that maps the **FULL_ON** state to logic 1 and the **OFF** state to logic 0 shall be inserted automatically. The default VCT facilitates building simple functional models. If this mapping is not the one desired for a particular connection, a user-defined VCT implementing the desired mapping can be specified explicitly for the connection (see also Annex B).

Supply port/net interconnections create a supply network that may span multiple instances at potentially multiple levels in the logic hierarchy. Evaluation of supply networks during simulation requires consideration of the whole collection of electrically equivalent supply ports/nets (see 4.5.5) making up each supply network.

a) A group of electrically equivalent ports/nets (see 4.5.5) constitutes a supply network, including ports/nets that are both equivalent by connection and declared electrically equivalent.
    1) The source(s) of the group are the top-level and leaf-level sources.
    2) The load(s) of the group are the top-level and leaf-level loads.
    3) Internal ports act only as connections within the group.
b) If there are no resolved nets in the group, then the group is unresolved.
c) For an unresolved group, it shall be an error if there is more than one supply source in the group.
d) If there is at least one resolved net in the group, then the group is resolved.
e) For a resolved group, it shall be an error if:
    1) The group contains two resolved nets with different resolution types.
    2) Any two resolved nets in the group are separated by a unidirectional internal port.
f) In general, it shall be an error if a unidirectional supply port (an input port or an output port) in the group:
    1) has a supply source on the load side, and
    2) has a load on the supply source side.
g) For an unresolved group of electrically equivalent supply ports/nets (see 4.5.5), the single source drives all the loads directly.
h) For a resolved group of electrically equivalent supply ports/nets:

1) All electrically equivalent resolved nets in a group are collapsed into a single resolved net.
2) Supply sources provide inputs to the resolved net.
3) The resolution type of the resolved net determines how inputs are resolved.
4) The resolved value is distributed to all loads.

9.2 Supply network simulation

9.2.1 Supply network initialization

Simulation initialization semantics are defined by each HDL. Existing models rely on the HDL initialization semantics for operations such as initializing read-only memories (ROMs), etc. To ensure that initialization of the design occurs correctly during power-aware simulation, model initialization code and design code should be cleanly separated. In Verilog-2005 or SystemVerilog, initial blocks can be used for model initialization code, since these are not affected by power-aware simulation semantics. In VHDL, model initialization code should be placed in processes that will not be synthesized and these processes should be included in an "always-on" power domain during power-aware simulation.

The initial state of supply ports and supply nets is **OFF** with an unspecified voltage value. The initial state of a supply set is determined by the initial state of each supply function of the supply set. The initial state of a supply set function is determined by the initial state of the corresponding supply net with which it has been associated or else the initial state of the root supply driver of that function.

NOTE—Implicitly created supply nets are initialized the same as explicitly created supply nets.

9.2.2 Power-switch evaluation

During simulation, a power switch created with **create_power_switch** corresponds to a process that is sensitive to changes in its input port (net state and voltage value), as well as the signals referenced in the Boolean expressions that define its control inputs. Whenever the input supply ports or control signals change, the corresponding on-state, on-partial-state, off-state, and error-state Boolean functions are evaluated and the value of the power switch output port is recomputed. See 6.21 **create_power_switch** for the algorithm used to determine the output value of the switch.

9.2.3 Supply network evaluation

During simulation, each supply object maintains two pieces of information: a supply state and a voltage value. The supply state itself consists of two pieces of information: an on/off state and a full/partial state. The supply state values are **FULL_ON**, **OFF**, **PARTIAL_ON**, and **UNDETERMINED**. **PARTIAL_ON** typically represents a resolved supply net state when some, but not all, switches are **FULL_ON** or any switch is **PARTIAL_ON** (see also 6.24.3).

During simulation, the supply network is evaluated repeatedly whenever the value of a root supply driver or a switch input or a resolved supply net input changes. Supply network evaluation consists of the following:

a) Evaluation and resolution of supply nets (see 6.24.3)
b) Evaluation of power switches (see 6.21)
c) Evaluation of supply set power states (see 9.3)

d) Evaluation and application of simstates (see 9.5 and 9.6).

The supply network is evaluated in the same step of the simulation cycle as the logic network. New root supply driver values are propagated along the connected supply nets in the same manner that logic values are propagated along the logic network.

NOTE—As no material distinction between **PARTIAL_ON** and **PARTIAL_OFF** exists, only **PARTIAL_ON** is defined.

9.3 Power state simulation

9.3.1 Supply state and power state control

The supply state of a supply port, supply net, or supply set function may be changed from an HDL testbench in simulation using the `set_supply_state` function defined in package UPF (see Annex B). The `set_supply_state` function assigns one of the enumeration values **OFF**, **PARTIAL_ON**, **FULL_ON**, or **UNDETERMINED** to the state field of the supply object to which it is applied, as indicated by its parameter. The voltage field of the supply object is left unchanged.

The supply state of a supply port, supply net, or supply set function may also be changed using the `supply_on` and `supply_off` functions defined in package UPF. The `supply_on` function sets the supply state to **FULL_ON** and the supply voltage as indicated by its parameter. The `supply_off` function sets the supply state to **OFF** and leaves the voltage unchanged.

The power state of a supply set may be changed from an HDL testbench in simulation using the `set_power_state` function defined in the package UPF (see 11.2). The `set_power_state` function activates the specified power state of the specified supply set (or supply set handle). This function can be used to control the power states of supply sets, before supply distribution networks have been implemented or completed.

When `set_power_state` is used to activate a supply set's power state, the functions of the supply set (e.g., primary.power) shall have their supply state set as follows:

a) If the specified supply set power state has a supply expression, then:
   1) For any term in the supply expression of the form <function>==<value>, the supply state of that function shall be set to the specified supply state value (e.g., through an invocation of set_supply_state).
   2) For any term in the supply expression of the form <function>!=<value>, the supply state of that function shall be set to the value **UNDETERMINED**.

b) If the specified supply set power state has no supply expression, then:
   1) If the simstate of the specified supply set power state is CORRUPT: the state shall be set to **OFF** and the voltage value is unspecified.
   2) For any other simstate: the state shall be set to **FULL_ON** and the voltage value is unspecified.

It shall be an error if an invocation of `set_power_state` results in the assignment of two different values to the same supply port, net, or supply set function.

The power state of a power domain, composite domain, group, or module may also be changed from an HDL testbench in simulation using the `set_power_state` function. In each case, the `set_power_state` function traverses the power state dependencies starting from a specified power state of the object and sets the power states or supply states of leaf-level objects as required to reach the specified power state of the top-level object.

For these object types, `set_power_state` recursively calls itself for each term of the form <object>==<state> in the defining expression of the specified power state. This recursion terminates when any of the following conditions hold:

— The power state is a deferred power state (has no defining expression). In this case, the specified state of the object is made active, and the function returns.

— The object is a supply set. In this case, the supply expression is processed instead of the logic expression, as described above, and the function returns.

The `set_power_state` function shall only be invoked to activate a definite or deferred power state. It shall be an error if it is invoked to activate an indefinite power state.

The `set_power_state` function does not attempt to set control signals to the values indicated in the defining expression. It only sets the leaf-level object power states and/or supply states as required by the power state specified in the top-level invocation. A testbench may also need to drive control signals to the values required to make a given state active.

It is possible for a call to `set_power_state` to cause activation of two different power states for the same object. It shall be an error if the two power states are not related by refinement.

For example, suppose `set_power_state` is invoked to set domain PD1 to a state S. If state S of domain PD1 requires domain PD2 to be in state S2 and domain PD3 to be in state S3, and these states S2 and S3 require domain PD4 to be in two different states S4a, S4b, and both of these latter states are deferred power states, the recursive calls of `set_power_state` will first activate state S4a of domain PD4 and then later activate state S4b of domain PD4. If these two states are related by refinement, the current power state of PD4 will be the most refined power state of these two. Otherwise, the current power state of PD4 will be the predefined **ERROR** power state.

NOTE—Tools may provide other mechanisms to change the power state of the supply set or power domain. Such mechanisms are outside the scope of this standard.

9.3.2 Supply state and power state determination

The supply state of a supply port, supply net, or supply set function is determined by the supply state of the supply subnet (see 4.5) containing it. The supply state of any given supply subnet is determined as follows:

— At the beginning of simulation:

  — The initial supply state is the supply state **OFF**.

— During simulation:

  — If the supply subnet contains a switch output port, then the current supply state is determined by evaluating the power switch whenever there is a change in the value of any of its inputs.

  — If the supply subnet contains a resolved supply net, then the current supply state is determined by evaluating the resolution function of the resolved supply net whenever there is a change in the value of any of its inputs.

  - Otherwise, the current supply state is the state most recently assigned to the root supply driver of the supply subnet by the set_supply_state command or the supply_on() function or the supply_off() function

The set of active power states of a supply set, power domain, composite domain, group, model, or instance is determined as follows:

- At the beginning of simulation, the set of active power states is computed, as required by the defining expressions of the power states of the object, based on:
  - the initial values of control signals,
  - the initial power states of subordinate objects, and
  - the initial supply states of supply objects
- During simulation, the set of active power states is re-computed whenever there is a change in any of the following characteristics of any subordinate object referenced in the defining expression of any power state of this object:
  - the current value of a control signal, or
  - the set of active power states of an object, or
  - the current supply state of a supply object.

A power state of a supply set is determined to be active as follows:

- If the power state is a deferred power state (see 4.7.3), then:
  - if the power state has a supply expression, the power state is active if the supply expression evaluates to True;
  - otherwise the power state is active if it was made active by `set_power_state`.
- Otherwise:
  - the power state is active if its defining expression evaluates to True,
  - and it shall be an error if it has a supply expression that does not evaluate to True.
- The predefined power state **OFF** is active if no other power state of the supply set is active.

A power state of a power domain, composite domain, group, model, or instance is determined to be active as follows:

- If the power state is a deferred power state, then the power state is active if it was made active by `set_power_state.`
- Otherwise, the power state is active if its defining expression evaluates to True.
- The predefined power state **UNDEFINED** is active if no other power state of the object is active.

The current power state of a supply set, power domain, composite domain, group, model, or instance is determined as follows:

- If exactly one power state of the object is active, then the current power state is that power state; else,

- If more than one definite or deferred power state of the object is active, and all active states are related by refinement, then the current power state is the most refined power state in that set; else,
- The current power state is the predefined power state **ERROR**.

It shall be an error if the current power state of a supply set or power domain is **UNDEFINED** and add_power_state for states of that object was specified with **-complete**.

9.4 Power state transition detection

Each object for which power states can be defined may have an associated set of named power state transitions. Each named power state transition is defined in terms of one or more pairs of states: a starting state (or *from state*), and an ending state (or *to state*). Named power state transitions occur when the *to state* becomes active after the *from state* is active, and certain other conditions are satisfied.

A transition from a *from state* to a *to state* may include one or more intermediate states. The set of intermediate states allowed for a given transition include any abstraction of the *from state* and any abstraction of the *to state*. If the power states of the object have been defined without being specified as complete, then the set of allowed intermediate states also includes the predefined power state UNDEFINED.

In the following,

- *active(S)* means that power state S is active
- *current(S)* means that power state S is the current power state
- *intermediate* means an intermediate state that is allowed in a given transition
- (…)* means a sequence of zero or more repetitions of the parenthesized item

A transition from state S1 to state S2, where S1 is an abstraction of S2 occurs when the following sequence of conditions occurs:

{active(S1) && not active(S2); active(S2)}

A transition from state S2 to state S1, where S2 is a refinement of S1 occurs when the following sequence of conditions occurs:

{active(S2); (not active(S2) && active(S1)}

A transition from state S1 to state S2, where S1 and S2 are not related by refinement, occurs when the following sequence of conditions occurs:

{active(S1); (current(intermediate))*; active(S2)}

9.5 Simstate simulation

9.5.1 Overview

The current simstate of a supply set (or supply set handle) is reevaluated whenever there is a change in the current power state of the supply set. If the current power state defines a simstate, then that simstate becomes the current simstate; otherwise, the current simstate remains unchanged. Each simstate has well-defined simulation semantics, as specified in the following subclauses. Multiple power states may be defined with the same simstate specification. The simstate semantics are applied to all elements that have the supply set connected to it (including no supply net connections except those implied by the supply set connection to the element) and that have the simstate semantics implicitly or explicitly enabled.

Elements implicitly connected to a particular supply set have simstate semantics enabled by default. Elements automatically or explicitly connected to a particular supply set have simstate semantics disabled by default. Use **set_simstate_behavior** to override the default enablement of simstate semantics (see 6.52).

The supply set powering a state element or the driver for a net can be in a state in which the supply is not adequate to support normal operational behavior. Under specified circumstances while in these states, the logic value of the state element or net becomes unknown. A corrupt value for a state element or net indicates the logic state of the state element or net is unknown due to the state of the supply powering the state element or driver of the net. The corrupt value of a state element or net shall be the HDL's default initial value for that object's type, except for VHDL `std_ulogic` and `std_logic` typed-objects, which shall use `X` as the corruption value (not `U`).

NOTE—An object can be declared with an explicit initial value. This explicit initial value has no relationship to the corrupt value for the object. For example, in VHDL, the objects of `Integer` type have the default initial value of `Integer'Left` (`-2147483648` for a system using 32 bits to represent `Integer` types). A process variable inferring a state element may be declared to be of type `Integer` with an initial value of `0`. The corrupt value for the variable is `Integer'Left`, not `0`.

The following subclauses define the simulation semantics for simstates. These semantics are applied to the elements connected to the supply set with simstate behavior **ENABLED**.

9.5.2 NORMAL

This state is a normal, power-on functional state. The simulator executes the design behavior of the elements consistent with the HDL or UPF specification that defines the element.

9.5.3 CORRUPT

This state is a non-functional state. For example, this state can be used to represent a power-gated/power-off supply set state. In this power state, state elements powered by the supply set and the logic nets driven by elements powered by the supply set are corrupted. The element is disabled from evaluation while this state applies.

As long as the supply set remains in a **CORRUPT** simstate, no additional activity shall take place within the elements, i.e., all processes modeling the behavior of the element become inactive, regardless of their original sensitivity list. Events that were scheduled for elements supplied by the supply set before entering this simstate shall have no effect.

9.5.4 CORRUPT_ON_ACTIVITY

This state is a power-on state that is not dynamically functional. For example, this state can be used to represent a high-voltage threshold, (body-bias) state that does not have characterized (defined) switching performance. In this simstate, the logic state of the elements is maintained unless there is activity on any of the element's inputs. Upon activity on any input, then all state elements and logic nets driven by the element are corrupted.

9.5.5 CORRUPT_ON_CHANGE

This state is a power-on state that is not dynamically functional. For example, this state can be used to represent a high-voltage threshold, (body-bias) state that does not have characterized (defined) switching performance. In this simstate, the logic state of the elements is maintained unless there is a change on any of the element's outputs. Upon change of any output, then all logic nets driven by that element output are corrupted.

9.5.6 CORRUPT_STATE_ON_CHANGE

This state is a power-on state that represents a power level sufficient to power normal functionality for combinational functionality, but insufficient for powering the normal operation of a state element if the state element is written with a new value. The simulator executes the design behavior of the elements consistent with the HDL or UPF specification that defines the element, except that any change to the stored value in a state element results in the writing of a corrupt value to the state element.

9.5.7 CORRUPT_STATE_ON_ACTIVITY

This state is a power-on state that represents a power level sufficient to power normal functionality for combinational functionality but insufficient for powering the normal operation of a state element if there is any write activity on the state element. The simulator executes the design behavior of the elements consistent with the HDL or UPF specification that defines the element, except that any activity inside state elements, whether that activity would result in any state change or not, results in the writing of a corrupt value to the state element.

9.5.8 NOT_NORMAL

This is a special, placeholder state. It allows early specification of a non-operational power state while deferring the detail of whether the supply set is in the **CORRUPT**, **CORRUPT_ON_ACTIVITY**, **CORRUPT_ON_CHANGE**, **CORRUPT_STATE_ON_CHANGE**, or **CORRUPT_STATE_ON_ACTIVITY** simstate. If the supply set matches a power state specified with simstate **NOT_NORMAL**, the semantics of **CORRUPT** shall be applied, unless overridden by a tool-specific option. **NOT_NORMAL** semantics shall never be interpreted as **NORMAL**.

The functions defined in package UPF (see 11.2) that query the simstate for a state that was originally **NOT_NORMAL** shall return the simstate to be applied in simulation for that state; e.g., **CORRUPT** for the default interpretation of **NOT_NORMAL**.

The query functions (see 11.1.2) that query the simstate for a state having a **NOT_NORMAL** simstate shall return **NOT_NORMAL** when it was not updated with any other simstate.

NOTE 1—Using the default interpretation of **CORRUPT** for **NOT_NORMAL** provides a conservative interpretation—the broadest corruption semantics—for simulation of the design for functional verification. However, a conservative interpretation of **NOT_NORMAL** for other tools, such as power estimation tools, might be to use a bias or lowered voltage level interpretation such as **CORRUPT_ON_ACTIVITY**.

NOTE 2—As it is possible for two or more power states of a supply set to match the state of the supply set's nets and for multiple simstate specifications to apply simultaneously, the effective result is that the simstate with the broadest corruption semantics shall apply. For example, a supply set that matches power states with simstates of **CORRUPT_STATE_ON_CHANGE** and **CORRUPT_STATE_ON_ACTIVITY** shall result in the application of **CORRUPT_STATE_ON_ACTIVITY** simstate semantics being applied.

9.6 Transitioning from one simstate state to another

9.6.1 Introduction

The following subclauses define the simulation semantics for transitions from one simstate to another. These semantics are applied to the elements connected to the supply set with simstate behavior **ENABLED**.

9.6.2 Any state transition to CORRUPT

In this case, the nets and state elements driven by the elements connected the supply set in this simstate shall be corrupted. The elements connected to this supply set are inactive as long as the supply set is in the **CORRUPT** simstate.

9.6.3 Any state transition to CORRUPT_ON_ACTIVITY

In this case, the current state of nets and state elements driven by the element shall remain unchanged at the transition. The processes modeling the behavior of the element shall remain enabled for activation (evaluation). Any net or state element that is actively driven after transitioning to this state shall be corrupted.

Any attempt to restore a retention register's retained value while in the **CORRUPT_ON_ACTIVITY** state shall result in corruption of the register's value.

9.6.4 Any state transition to CORRUPT_ON_CHANGE

In this case, the current state of nets and state elements driven by the element shall remain unchanged at the transition. The processes modeling the behavior of the element shall remain enabled for activation (evaluation).

9.6.5 Any state transition to CORRUPT_STATE_ON_CHANGE

In this case, the current state of nets and state elements driven by the element shall remain unchanged at the transition. The processes modeling the behavior of the element shall be enabled for activation (evaluation).

9.6.6 Any state transition to CORRUPT_STATE_ON_ACTIVITY

In this case, the current state of nets and state elements driven by the element shall remain unchanged at the transition. The processes modeling the behavior of the element shall be enabled for activation (evaluation).

9.6.7 Any state transition to NORMAL

In this case, the processes modeling the behavior of the element shall be enabled for activation (evaluation), and the combinational and level-sensitive sequential logic functionality in each process shall be re-evaluated to restore and properly propagate constant values and current input values. Edge-sensitive sequential logic functionality within the element shall not be evaluated at this transition.

9.6.8 Any state transition to NOT_NORMAL

NOT_NORMAL is simulated according to the interpretation of this placeholder simstate (see 9.5.8).

9.7 Simulation of retention

9.7.1 Introduction

Subclause 9.7 covers some of the basics of retention register operation and modeling, which are useful in describing the simulation semantics for the **set_retention** command (see 6.49). The following abbreviations are used in various figures and tables herein:

`VDD`	primary supply port of the register
`VDDRET`	retention supply port of the register
`SS`	save signal is asserted
`SC`	save condition
`RS`	restore signal is asserted
`RC`	restore condition
`RTC`	retention condition

NOTE—In verification, if no retention supply is specified in a retention strategy, then for any inferred retention cell instance, retention supply port `VDDRET` will be connected to an anonymous always-on supply (see 6.49).

9.7.2 Retention corruption summary

A retention register has the same simulation behavior as a regular register when both supplies `VDD` and `VDDRET` are `ON`, the save/restore signals are inactive, and the retention condition is *False*. The main simulation difference between a non-retention register and a retention register comes when the corruption behavior is modeled during various power state transitions. The retention register is composed of at least three components (see 4.4.5), as follows:

— *Register value* is the data held in the storage element of the register. In functional mode, this value gets updated on the rising/falling edge of clock or gets set or cleared by set/reset signals, respectively.

— *Retained value* is the data in the retention element of retention register. The retention element is powered by the retention supply.

— *Output value* is the value on the output of the register.

The retained value of the retention register can be corrupted in the following ways:

a) If `VDDRET==OFF`

   Corrupt if `RET_SUP_COR` is set

b) Else If `VDDRET==ON`

   1) If `VDD==ON`

      `(SS && SC) && (RS && RC)` (both save/restore functions are true) and `SAV_RES_COR` is set

2) Else If `VDD==OFF`

   i) `(SS && SC)`—trying to save when domain off

   ii) `(RS && RC)`—trying to restore when domain off

   iii) `!RTC`

The output value of the retention register can be corrupted in the following ways:

c) If **-use_retention_as_primary** is specified

   Output is corrupted whenever retained value (described above) is corrupted

d) If **-use_retention_as_primary** is not specified

   1) If `VDD==OFF`

      Corrupt always

   2) Else If `VDDRET==OFF`

      Corrupt if `RET_SUP_COR` is set

In summary, the preceding algorithm covers all the conditions by which a retention register (i.e., retained value/output value) can be corrupted. A corrupted retention register can then be restored to a valid state by a combination of one or more of the following:

— Restore (power up) the corrupting supplies.
— Deassert save/restore signals if the corruption is due to the condition when both are true simultaneously.
— Deassert retention condition.
— Apply reset/set and/or clock.

9.7.3 Retention modeling for different retention styles

Depending on the type of retention, the controlling inputs of the retention register like the save/restore signals may or may not exist on the register boundary. Thus, it is important to understand the modeling of the different flavors of retention, namely balloon-style retention and master/slave-alive style retention (see 4.4.5).

When the **set_retention** (see 6.49) is specified with **-save_signal** and (or) **-restore_signal**, balloon-style retention semantics are applied to it. The process of saving/restoring is unique to balloon-style retention. When the **set_retention** is not specified with both **-save_signal** and **-restore_signal** and it is specified only with a **-retention_condition**, the master/slave-alive retention semantics are applied instead. In this type of retention, the restore happens during power-up, as the master/slave latch is kept on the retention supply. However, whether to be in a retention state or not may be controlled by the value of one or more ports on the retention register. In the case of master/slave-alive retention, when the retention condition is true, the **retention_condition** shall take precedence over other signals such as clocks and async sets/resets.

A retention register may be in one of the following states:

— **NORMAL**—Functional/active mode, all supplies expected to be ON.
— **SAVE**—The time snapshot where the save action occurs (for balloon-latch style registers).
— **RESTORE**—The time snapshot where the restore action occurs (for balloon-latch style registers).

— **RETAIN_ON**—The time snapshot where the primary supply is `ON` and the register is in retention state (`retention_condition == True`).

— **RETAIN_OFF**—The time snapshot where the primary supply is `OFF` and the register is in retention state (`retention_condition == True`).

— **PARTIAL_CORRUPT**—The retained value is corrupted, but the register value is not corrupted.

— **CORRUPT**—The register value and retained value are both corrupted.

Table 8 summarizes the power state of a balloon-style retention register with respect to the states of the signals.

Table 9 summarizes the power state of a master/slave alive retention register with respect to the states of the signals.

Table 10 shows the output values of the retention register depending on the state of retention register.

Table 8—Retention power state table for balloon-style retention[a]

VDD	VDD RET	SS && SC	RS && RC	RTC	Retained value	Register value	Register state	Valid next states	Comments
ON	ON	FALSE	FALSE	FALSE	Previous saved data	Previous state value	NORMAL	SAVE, RESTORE	—
ON	ON	FALSE	FALSE	TRUE	Previous saved data	Previous state value	RETAIN_ON	NORMAL, RETAIN_OFF, RESTORE	—
ON	ON	FALSE	TRUE	X	Previous saved data	Retention value	RESTORE	NORMAL, RETAIN_ON	—
ON	ON	TRUE	FALSE	X	Register value	Previous state value	SAVE	RETAIN_ON, NORMAL	—
ON	ON	TRUE	TRUE	X	CORRUPT	CORRUPT	CORRUPT	NA	SAV_RES_COR is set
ON	OFF	X	X	TRUE	CORRUPT	CORRUPT	CORRUPT	NA	—
ON	OFF	X	TRUE	FALSE	CORRUPT	CORRUPT	CORRUPT	NA	RET_SUP_COR is set
ON	OFF	X	FALSE	FALSE	CORRUPT	Previous state value	PARTIAL_ CORRUPT	NORMAL	RET_SUP_COR is set
OFF	OFF	X	X	X	CORRUPT	CORRUPT	CORRUPT	NA	RET_SUP_COR is set
OFF	ON	FALSE	FALSE	FALSE	CORRUPT	CORRUPT	CORRUPT	NA	!RTC
OFF	ON	FALSE	FALSE	TRUE	Previous saved data	CORRUPT	RETAIN_OFF	RETAIN_ON	—
OFF	ON	FALSE	TRUE	X	CORRUPT	CORRUPT	CORRUPT	NA	Restore during power-down
OFF	ON	TRUE	X	X	CORRUPT	CORRUPT	CORRUPT	NA	Save during power-down

[a]The X in this table denotes a "don’t care" condition. Valid next states are non-corrupting next states.

Table 9—Retention state table for master/slave-alive retention

VDD	VDD RET	RTC	Retained/ register value	Register state	Valid next states	Comments
ON	ON	FALSE	Previous state value	NORMAL	RETAIN_ON	—
ON	ON	TRUE	Previous state value	RETAIN_ON	NORMAL, RETAIN_OFF	—
ON	OFF	TRUE	CORRUPT	CORRUPT	NA	RET_SUP_COR is set
ON	OFF	FALSE	CORRUPT	CORRUPT	NA	RET_SUP_COR is set
OFF	OFF	X	CORRUPT	CORRUPT	NA	—
OFF	ON	FALSE	CORRUPT	CORRUPT	NA	!RTC
OFF	ON	TRUE	Retention value	RETAIN_OFF	RETAIN_ON	—

Table 10 —Retention output value table[a]

use_retention_ as_primary	State	Register value	Output value
TRUE	NORMAL	DATA	DATA
TRUE	RETAIN-ON/RETAIN-OFF	DATA	DATA
TRUE	SAVE	DATA	DATA
TRUE	RESTORE	DATA	DATA
TRUE	CORRUPT	X	X
FALSE	NORMAL	DATA	DATA
FALSE	RETAIN-ON/RETAIN-OFF	DATA	VDD==ON?DATA:X
FALSE	SAVE	DATA	VDD==ON?DATA:X
FALSE	RESTORE	DATA	VDD==ON?DATA:X
FALSE	CORRUPT	X	X

[a]DATA in Table 10 stands for a valid data, and X stands for corrupt data.

Figure 6 describes the sequence of transitions in balloon-style retention register. In this case, the state transitions are not synchronous, i.e., they are not caused due by clock transitions.

Figure 7 describes the sequence of transitions in a master/slave-alive register. In this case, the state transitions are not synchronous, i.e., they are not caused due by clock transitions.

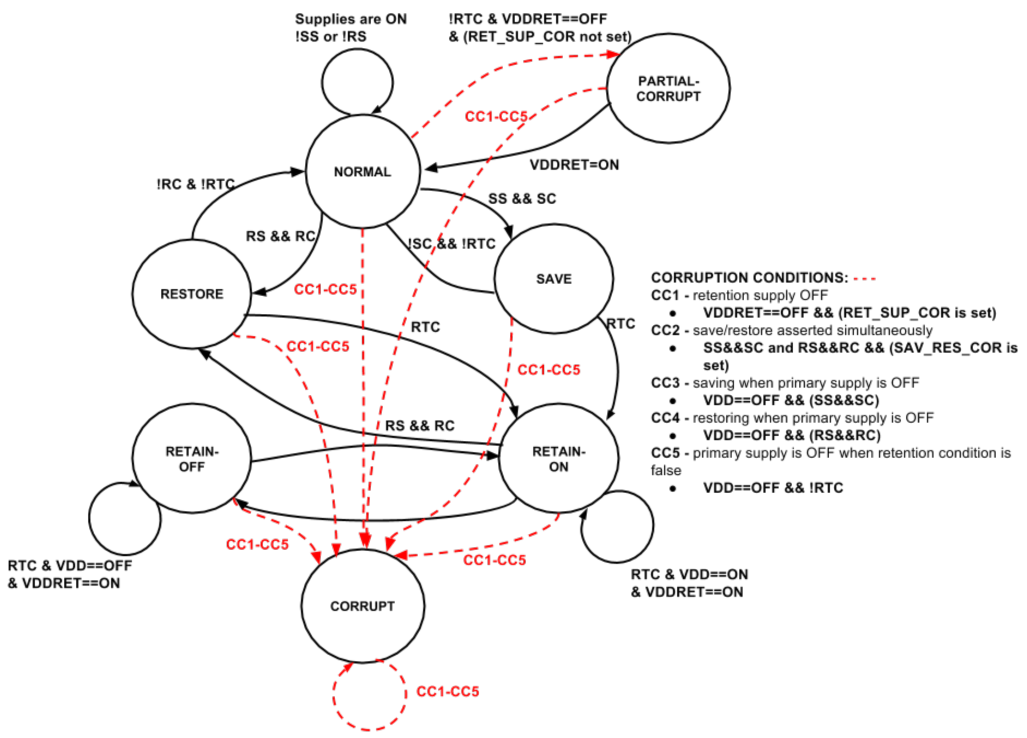


Figure 6—Retention station transition diagram for balloon-style retention

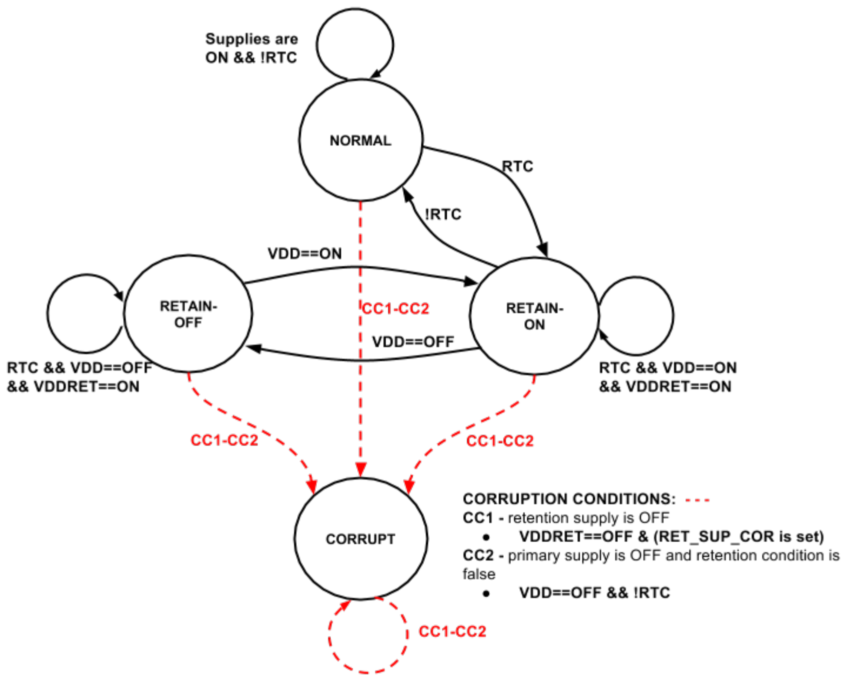


Figure 7—Retention state transition diagram for master/slave-alive style retention

9.8 Simulation of isolation

The simulation semantics for isolation are defined by the following algorithm, unless a specific simulation model is specified for a given instance by a **use_interface_cell** command (see 6.55). The algorithm is for a single-stage isolation cell with an explicitly specified **-isolation_supply**, **isolation_sense high**, and a **clamp_value** of **0**, **1**, **Z** from a predefined logic type (see 5.4), or any value of a user-defined datatype:

```
on any input change,
   if the current simstate of the isolation supply set is NORMAL, then
      if isolation_signal == 0 then
         data_output = data_input;
      else if isolation_signal == 1 then
          data_output = clamp_value;
      else /* isolation_signal has an unknown value */
          data_output = corrupted value;
      end
   else /* the isolation supply set is in a non-NORMAL state */
      data_output = corrupted value;
   end;
```

Where the corrupted value is X from the predefined logic type for a 1-bit port of that type, an array of X values for a port that is an array with elements of that type, and the leftmost value of the relevant data type for any port that is of a user-defined datatype. For an isolation cell with **-isolation_sense** low, the isolation signal values 0 and 1 would be interchanged.

For a single-stage isolation cell with an explicitly specified **-isolation_supply**, **isolation_sense high**, and a **-clamp_value** of **latch**:

```
on any input change,
   if the current simstate of the isolation supply set is NORMAL, then
      if isolation_signal == 0 then
         data_output = data_input;
         latched_value = data_input;
      else if isolation_signal == 1 then
         data_output = latched_value;
      else /* isolation_signal has an unknown value */
         data_output = corrupted value;
         latched_value = corrupted value;
      end
   else /* the isolation supply set is in a non-NORMAL state */
      data_output = corrupted value;
      latched_value = corrupted value;
   end;
```

NOTE—For an isolation cell inferred from a strategy specified with **-isolation_supply** {}, the above algorithms would test whether the supply function of the primary supply of the location domain that corresponds to the clamp value is FULL_ON, rather than testing whether the current simstate of the isolation supply set is NORMAL (see 6.44).

For a multi-stage isolation cell with N stages, each stage is simulated as given above, and the multiple stages are composed as follows:

```
isolation_stage[1].input = data_input;
isolation_stage[1].isolation_supply = isolation_supply[1];
isolation_stage[1].isolation_signal = isolation_signal[1];
for each stage K in 2 to N,
  isolation_stage[K].input = isolation_stage[K-1].output;
  isolation_stage[K].isolation_supply = isolation_supply[K];
  isolation_stage[K].isolation_signal = isolation_signal[K];
end;
data_output = isolation_stage[N].output;
```

9.9 Simulation of level-shifting

The simulation semantics for level-shifting are defined by the following algorithm, unless a specific simulation model is specified for a given instance by a **use_interface_cell** command (see 6.55):

```
on any input change,
   if the current simstate of any level shifter supply set \
         is not NORMAL, then
      data_output = corrupted value;
   else
      data_output = data_input;
   end;
```

Where the corrupted value is as defined in 9.8.

9.10 Simulation of repeaters

The simulation semantics for repeaters are defined by the following algorithm:

```
on any input change,
   if the current simstate of the repeater_supply \
         is not NORMAL, then
      data_output = corrupted value;
   else
      data_output = data_input;
   end;
```

Where the corrupted value is as defined in 9.7.

10. UPF information model

10.1 Overview

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 motivation for providing the information model is to provide a standard model which forms the underlying structure which shall be used by query commands and HDL package functions to access information related to power management. The objective is to provide a standard interface which can be used to access power-management information resulting from UPF and also provide a back-annotation to original UPF source.

The objects in the information model shall be constructed after all the UPF-processing steps have been completed, more specifically after the phase 4 of UPF processing (see 8.3.5). This implies that the APIs that query the information model (i.e., UPF queries and UPF HDL package functions) will only work after phase 4 of UPF processing. The information model does not capture the intermediate steps involved in reaching the phase 4.

It shall be an error if a query function:

— Appears in a power model
— Is followed by a UPF command that would affect the power intent

NOTE—Since the information model will only be complete after phase 4, therefore the UPF query commands cannot be used to construct the power intent which they are querying.

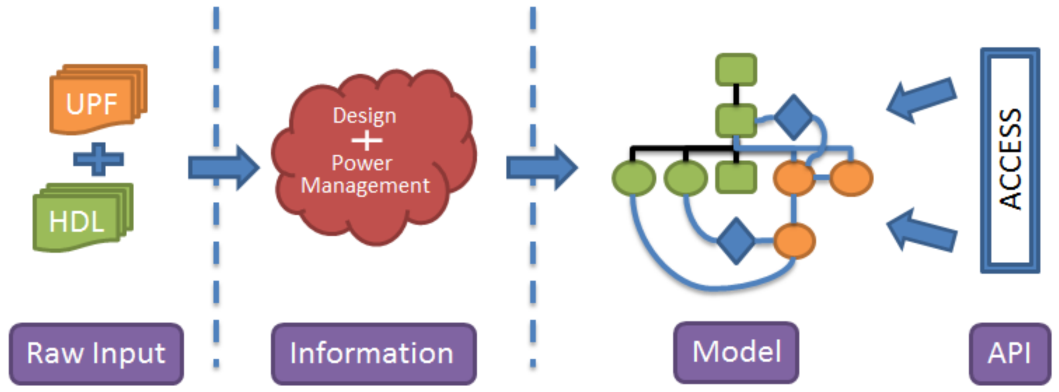


Figure 8—UPF information model flow

10.2 Components of UPF information model

10.2.1 Overview

The UPF information model consists of a collection of objects and the properties present on those objects. These objects belong to one of the various classes defined in the information model.

10.2.2 Objects

The objects in the information model are the primary holders of information. They are instances of the classes which belong to the UPF information model. They represent information about UPF and HDL and the relationship between UPF and HDL. The information is present on these objects in the form of properties, which can be accessed via APIs. Each object shall be denoted by a unique identifier called a UPF handle which shall be used by the APIs to access information present on it. They are broadly classified into three groups:

a) UPF objects
b) HDL objects
c) Relationship objects

10.2.3 Properties

10.2.3.1 Overview

Properties are pieces of information present on an object. They can be of the following types:

Table 11 —Kinds of properties

Property type	Property value	Property type name
Basic	String	upfStringT
	Integer	upfIntegerT
	Boolean	upfBooleanT
	Float	upfRealT
	Enumerated	Type names with suffix E
Complex	Handle to objects/properties	upfHandleT
	List of handles to other objects	upfIteratorT

Similar to objects, the properties are also referred by unique IDs which is constructed from a property name. Each property value of basic types is represented in a string when returned from query commands. The complex property values will be represented as a UPF handle or a list of UPF handles.

10.2.3.2 Dynamic properties

Some objects in the information model also maintain certain additional properties that are applicable during simulation environment. These are called "dynamic properties" and are only accessed by HDL package interface (see 11.2). Some of the dynamic properties also support write access under specific circumstances during the simulation (see 11.2.3.3). They enable the user to build abstract testbenches and checker/coverage models based on objects defined in the information model.

10.3 Identifiers in information model (IDs)

10.3.1 Overview

The various components in the information model are assigned unique strings which act as identifiers or IDs. These IDs are categorized into the following formats.

10.3.2 Handle ID or UPF handle

10.3.2.1 Overview

All objects in the information model are represented by a unique ID termed a *UPF handle*. This ID is used to query the properties in the information model. The definition of UPF handle varies from class to class but is broadly categorized into the following two kinds:

a) Hierarchical path ID
b) Tool-generated ID

10.3.2.2 Hierarchical path ID

10.3.2.2.1 Overview

The hierarchical path ID is the absolute hierarchical pathname from UPF root scope. It always starts with the hierarchical path separator "/" and can have names separated by "/", "." and "@" characters. These IDs are used for UPF and HDL group of objects. Since UPF ensures that there is no name clash with the design hierarchy, it is ensured that there is no conflict between the HDL objects and UPF objects, except in cases of UPF objects that get implemented in HDL and UPF objects representing port and net with the same name.

Examples

```
/top/dut_i/mid_i         # Handle to a scope in HDL
/top/dut_i/PD.iso_strat # Handle to an isolation strategy
/top/dut_i/port@1       # Handle to an HDL bit of a multi-bit port
```

10.3.2.2.2 Implemented UPF objects

Implemented UPF objects are UPF objects that get implemented and become part of HDL description. In such case, there will be two objects with the same name in a given scope. One will be a HDL object and another will be a UPF object. In such case, there will be a property that links from the UPF object to the HDL object. In case of implemented UPF objects, a search in the scope by name will always return the matching UPF object. This is typically the case with supply or logic network, where in earlier stages it may completely reside in UPF and later gets implemented and present in both UPF and HDL.

10.3.2.2.3 UPF ports and nets with same name

UPF also allows creation of supply port and supply nets with the same name in a given scope. In such cases, both the supply port and the supply net will result in the same hierarchical path ID. In order to avoid ID clash, the hierarchical path ID of supply net is suffixed with class ID separated by an "@" character. A search by name in that scope will result in the handle to supply port. However, if a search of the supply net is required, then the name of the supply net needs to be suffixed with an "@upfSupplyNetT" string.

Examples

```
UPF:
create_supply_port VDD
```

```
create_supply_net VDD
Handle:
/top/VDD                 #Handle to a supply port
/top/VDD@upfSupplyNetT  #Handle to a supply net
```

NOTE—The suffixing of class name in the handle ID is only required in the special case when there is a name conflict.

10.3.2.2.4 Hierarchical path IDs and relative pathnames

The relative pathnames are hierarchical pathnames that do not start with a "/" character. They provide reference to objects within the current scope. The UPF query commands that accept UPF handle can also accept relative pathnames. In that case, the UPF handle will be constructed internally by the query command by prefixing the UPF handle of the active scope.

The query commands will also accept "." where object handle is required. In that case, the "." will be expanded to handle ID of the current scope.

Examples

```
set_scope /top/dut_i
upf_query_object_properties mid/PD #Handle: /top/dut_i/mid/PD
upf_query_object_properties .      #Handle: /top/dut_i
```

NOTE—The automatic prefixing of the UPF handle of the active scope will not happen to tool-assigned IDs (starting with "#").

10.3.2.3 Tool-generated ID

The tool-generated IDs are special IDs generated by the tool constructing the power intent. They have a specific pattern with a mandatory # prefix and an integer counter in the end. They are created for relationship objects. The specific pattern of the tool-generated IDs for different classes of relationship objects is discussed in the respective subclauses.

Examples

```
#UPFEXTENT1#
```

10.3.3 Class ID

The class IDs are unique strings that represent class names in the information model. The classnames are represented in capitalized words with a "upf" prefix and "T" suffix.

Examples

```
upfPowerDomainT, upfHdlScopeT
```

10.3.4 Property ID

The property IDs are unique strings that represent the properties present on the object. The property names are denoted by all lowercase words separated by an "_" (underscore character) and a "upf" prefix.

Examples

```
upf_name, upf_parent
```

10.3.5 Enumerated ID

The enumerated IDs are strings that represent the enumerated values of a particular enumeration type. Some of these IDs are reused from the values already defined in the respective UPF commands except in cases where they result in a name conflict. All enumerated IDs are represented as uppercase strings separated by an "_" (underscore character). In case of name conflicts, the IDs are prefixed by "UPF_" followed by appropriate keywords.

Examples

```
UPF_SENSE_HIGH, FULL_ON, UPF_CELL_ISOLATION
```

10.4 Classification of objects

10.4.1 Overview

The objects in the information model are classified into three major groups: UPF objects, HDL objects, and relationship objects.

10.4.2 UPF objects

The UPF objects represent the group of objects that are created in UPF via UPF commands e.g., power domains, power states, etc. They represent the abstract objects that are created by UPF and have a valid name in the design hierarchy. They contain information coming from UPF commands and also the effect of application of those commands on HDL.

10.4.3 HDL objects

10.4.3.1 Overview

The HDL objects is a group of objects that are created to represent HDL information in the UPF information model. These objects capture the abstracted HDL information which is independent of the language in which the design is written. The objects are required to capture certain relationships needed for maintaining the power-management information. The relationships could be the following (but not limited to):

a) Creation scope of UPF objects
b) Extent and effective elements list of UPF objects
c) Control signals
d) Cells and their power-management information
e) Power-management cells that are already present in design
f) Supply/logic network which is already present in design

10.4.3.2 UPF information model and other HDL information models (e.g., VPI, VHPI, etc.)

The HDL objects in the UPF information model only represent the subset of information present in HDL information models. The idea is to have an abstraction of design information, coming from HDL, necessary to capture the power management. These objects are not designed to substitute the HDL information model which contains detailed knowledge of HDL, e.g., SystemVerilog, VHDL, etc.

If it is required to extract additional information about the HDL objects, then the user can either depend on the find_objects UPF command or rely on HDL information models (VPI, VHPI, etc.) to extract any information. The information model will provide the RTL style pathname through an API which can be used to get respective handles of the HDL information model (see 11.1.2.4).

If a particular HDL object is not present in the UPF information model, it does not imply that it is not present in the actual design. However, if the HDL object is present in the UPF information model, then it must be present in the user design at some stage in the design flow.

NOTE—There are some HDL objects which are inferred from UPF at RTL stage but not present in the original HDL, e.g., an isolation cell inserted for a strategy. The instance of such special cells will be represented as an HDL object at the target location determined by strategies.

10.4.3.3 Complex HDL objects

10.4.3.3.1 Overview

The HDL object also represents signals of complex types, like record, structure, arrays, as a multi-bit (upfHdlPortMultiBitT or upfHdlNetMultiBitT) kind of object. The signals of a complex type get transformed into a normalized vector of bits determined by a normalization algorithm. The tools can choose any normalization algorithm as long as it maintains some basic properties and provides an API to extract a RTL style name. This helps in providing a consistent and simple representation across all HDLs.

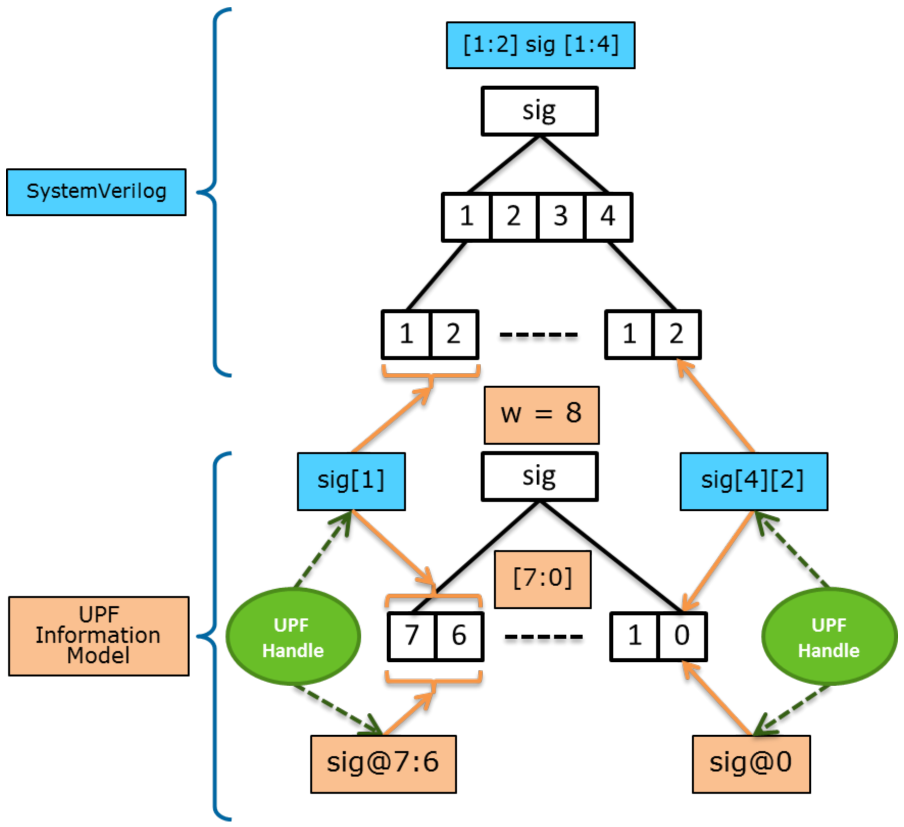


Figure 9 —Multi-bit type HDL objects

10.4.3.3.2 Multi-bit representation of complex HDL objects

Any HDL object of complex type that requires multiple bits to represent the value can be represented as a multi-bit (upfHdlPortMultiBitT or upfHdlNetMultiBitT) object in the information model. This representation provides a common, simple representation of any HDL object in information model without duplicating the type information from HDLs. The tools can do the translation from HDL object to multi-bit object on the fly using a specific normalization algorithm. The following are some of the properties of the multi-bit type object:

a) Any multi-bit type object represents a vector of bits of size "width". The equivalent bit representation in SystemVerilog for this object is "bit [width – 1: 0] <name of object>".

b) A bit of the multi-bit (upfHdlPortBitT or upfHdlNetBitT) object represents the normalized bit presentation of the complex type.

   1) The handleID of a bit object consists of either a valid RTL representation or a normalized representation of the form <object name>@<normalized index>. For example, sig[4][2], sig@0.

c) A slice of a multi-bit (upfHdlMultiBitSliceT) object is a subset of consecutive bits of the multi-bit object that represents a part of the complex type.

   1) The handleID of a slice object consists of either a valid RTL representation or a normalized representation of the form <object name>@<normalized msb>:<normalized lsb>. For example, sig[1], sig@7:6.

NOTE 1—The normalized handle ID of bit or slice objects is only returned by the APIs and users are not required to construct it on their own. The valid RTL name can be extracted from the normalized representation using upf_query_object_pathname API. See 11.1.2.4 for more details.

NOTE 2—In certain cases, only the normalized handle ID of bit is available. This is especially applicable for VHDL where there are scalar objects that require multiple bits to represent its value but cannot be split into bits at RTL (e.g., integer, enumerated types). The upf_query_object_pathname API will return null when queried on such bit objects, as there is no valid RTL representation. In such cases, the bit object also contains a special property "upf_smallest_atomic_slice" which can be used to get a handle of the smallest slice that represents the atomic object in HDL that has a valid RTL name.

Table 12—Multi-bit representation of various types in SystemVerilog

SystemVerilog	Multi-bit representation	Bit handle
bit [0:7] sig;	[7:0] sig	sig[3], sig@3
wire [7:0] sig;	[7:0] sig	sig[3], sig@3
int sig[1:0];	[63:0] sig	sig[34], sig@34 Smallest atomic slice is not populated as bit has valid RTL name

Table 13—Multibit representation of various types in VHDL

VHDL	Multi-bit representation	Bit handle
signal sig: bit_vector(0 to 7);	[7:0] sig	sig[3], sig@3
signal sig: std_logic_vector(7 downto 0);	[7:0] sig	sig[3], sig@3

type sig_arr is array(1 downto 0) of integer;
signal sig: sig_arr;
	[63:0] sig	sig@34
		Smallest atomic slice of sig@34 is sig@63:32 or sig(1).

10.4.4 Relationship objects

The relationship objects belong to a group of objects that are present for a special purpose in the information model. They capture certain relationships between other objects, e.g., relationship between UPF object and HDL object. The relationship objects are present only in UPF information model and do not exist in the user design. The handle of a relationship object consists of a tool-specific generated ID that may vary from one tool run to another.

10.4.5 Base classes

A variety of abstract base classes exist that share some common properties for a set of UPF objects. The various base classes in the UPF information model are shown in Table 14.

Table 14—Base classes

S. no.	Class name	Properties	Description
1	upfBaseT	—	Root class
2	upfBaseNamedT	upf_name, upf_parent	Base class for named objects
3	upfBaseRelationshipT	—	Base class for relationship objects
4	upfBaseHdlT	upf_cell_info upf_hdl_attributes upf_extents	Base class for HDL objects
5	upfBaseUpfT	upf_file, upf_line upf_creation_scope	Base class for UPF objects
6	upfExtentClassT	upf_effective_extents upf_supply_set_handles	Base class for objects having extents
7	upfHdlDeclT	—	Base class for HDL declarations
8	upfNetworkClassT	upf_hdl_implementation upf_root_driver upf_network_attributes	Base class for network UPF objects
9	upfStateClassT	upf_is_illegal	Base class for state objects
10	upfHdlNetClassT	—	Base class for HDL net objects
11	upfHdlPortClassT	upf_port_dir	Base class for HDL port objects
12	upfNetClassT	upf_fanin_conn upf_fanout_conn	Base class for UPF net objects
13	upfPortClassT	upf_hiconn, upf_loconn upf_port_dir	Base class for UPF port objects
14	upfStrategyT	upf_logic_refs	Base class for UPF strategies
15	upfBoundaryStrategyT	upf_location upf_applies_to upf_source_filter upf_sink_filter upf_name_prefix upf_name_suffix upf_is_use_equivalence	Base class for UPF boundary strategies

The different base classes can be used for checking and restricting Tcl procs to the class of objects that have the common properties. They provide a shorthand of selecting those objects. The class IDs can be used by the query "upf_object_in_class".

NOTE—Some of the derived classes may not contain some of the common properties present in the base classes. In such cases the property will not be populated for that object. For example, upf_is_illegal is not present in upfSupplyPortStateT even though it is derived from upfStateClassT. In that case, the upf_is_illegal is not populated for any object of upfSupplyPortStateT class. For list of all the properties present in different classes refer to 10.6.

10.4.6 Class hierarchy

In this standard, some of the diagrams (labeled as UML) are described using the UML notation. UML is described in ISO/IEC 19501:2005.

Figure 10 shows the class hierarchy for UPF-related classes.

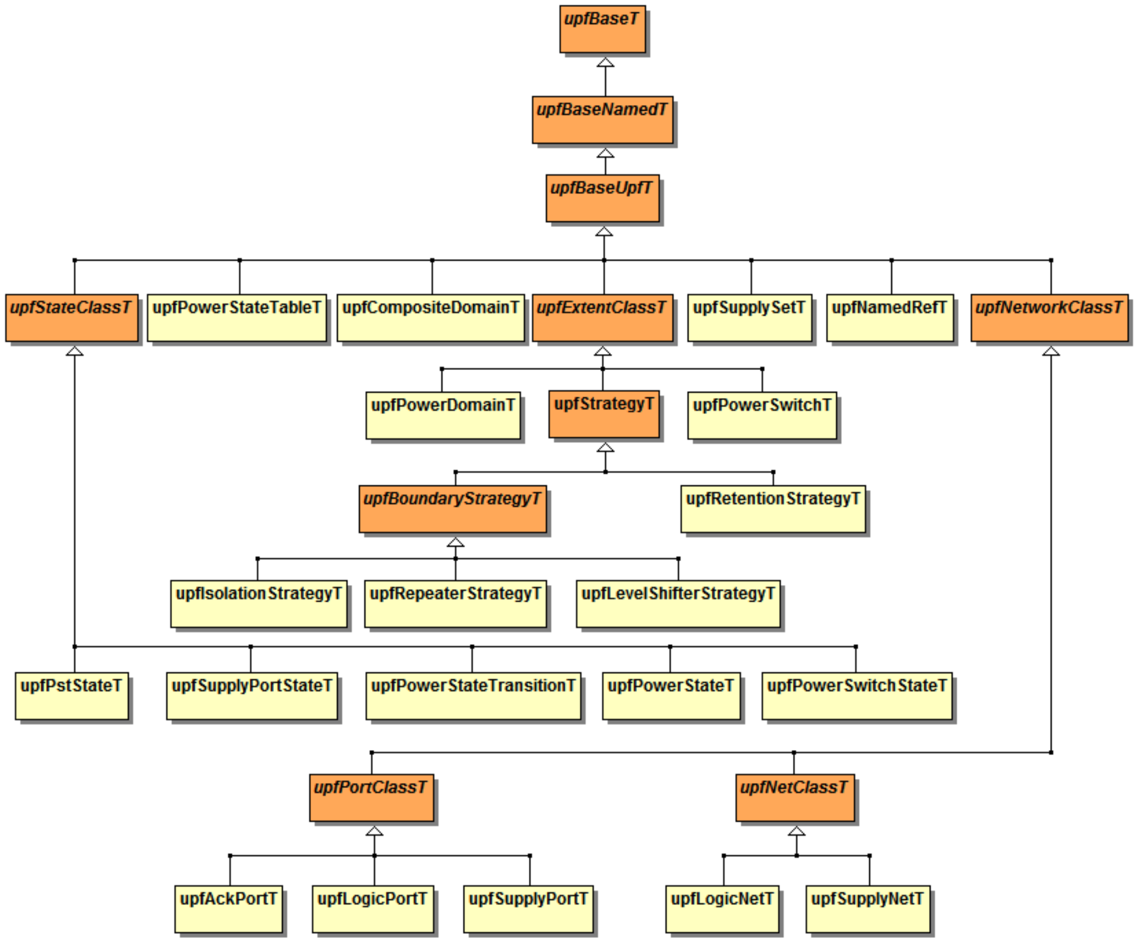


Figure 10 —UML class diagram showing class hierarchy of UPF objects

Figure 11 shows the class hierarchy for HDL-related classes.

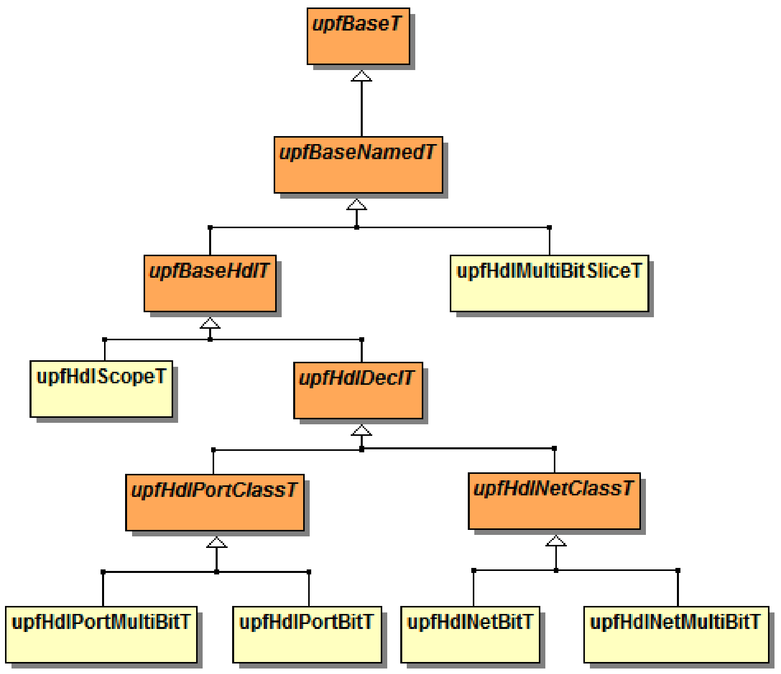


Figure 11 —UML class diagram showing class hierarchy of HDL objects

Figure 12 shows the class hierarchy of relationship objects.

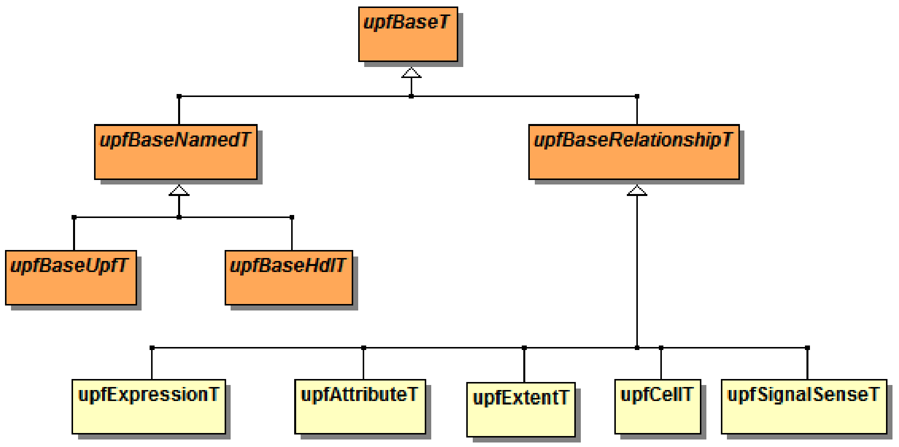


Figure 12 —UML class diagram showing class hierarchy of relationship objects

10.5 Example of design hierarchy

In order to emphasize the various concepts related to the UPF information model, an example design of the structure shown in Figure 13 was used. The UPF design root for all the commands start at /top/dut_i level. The root of the design starts at TB level. This root is also known as *root instance*. All hierarchical path IDs start from below the root; excluding the name of the root (i.e., TB in this case). Hence the hierarchical path ID for /TB/top/dut_i instance is represented as "/top/dut_i".

NOTE—It is the responsibility of the tool to define the root instance for a given design. The tools may provide mechanism to reset the root instance to some other level in the design hierarchy. However, this will affect the return value of query commands and representation of hierarchical path IDs. It shall be an error if root instance is defined as the hierarchy below the UPF design root.

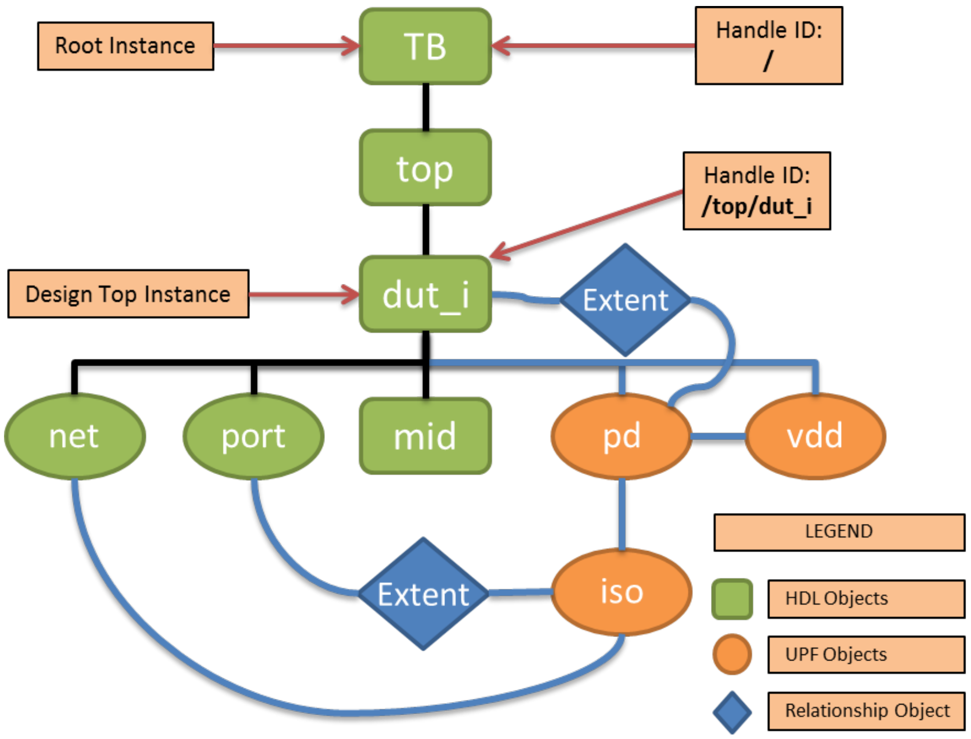


Figure 13—Example of design hierarchy

10.6 Object definitions

10.6.1 UPF objects

10.6.1.1 Power domain

Class name	upfPowerDomainT	
Class membership	upfPowerDomainT, upfExtentClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>/<upf_name of Object>`	
Handle ID examples	`/top/dut_i/PD`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfHdlScopeT	The HDL scope in which the object was created
upf_effective_extents	upfExtentT	The upfExtentT object that points to the first element in the effective_element_list of corresponding UPF command (see 10.6.3.1 for more details)
upf_supply_set_handles	List of upfSupplySetT	The list of supply set handles defined on the object
upf_upper_boundary	List of upfHdlScopeT	The list of HDL scopes forming the upper boundary of this power domain
upf_lower_boundary	List of upfBaseHdlT	The list of HDL objects forming the lower boundary of this power domain
upf_level_shifter_strategies	List of upfLevelShifterStrategyT	The list of level-shifter strategies defined for this power domain
upf_retention_strategies	List of upfRetentionStrategyT	The list of retention strategies defined for this power domain
upf_isolation_strategies	List of upfIsolationStrategyT	The list of isolation strategies defined for the power domain
upf_repeater_strategies	List of upfRepeaterStrategyT	The list of repeater strategies defined for the power domain
upf_pd_states	List of upfPowerStateT	List of states defined on power domain
upf_pd_state_transitions	List of upfPowerStateTransitionT	List of power state transitions defined by `describe_state_transition` upf command
Dynamic property (only available during simulation)		
upf_current_state	upfPowerStateT	The current state of the object during simulation

The object of upfPowerDomainT class is created when create_power_domain command is executed. As defined by UPF, the object is created in the scope where create_power_domain command was executed. The object contains various properties which capture the information coming from UPF and the application of UPF command on HDL design.

The following properties identify the objects that are defined within a power domain's scope. All such objects are defined within the same namespace, and therefore all such objects for a given domain must have unique names.

a) upf_supply_set_handles
b) upf_level_shifter_strategies
c) upf_retention_strategies
d) upf_isolation_strategies
e) upf_repeater_strategies
f) upf_pd_states
g) upf_pd_state_transition

Examples

UPF source: test.upf

```
1 set_scope dut_i
2 create_power_domain PD \
3  -elements { mid } \
4  -supply { primary }
5
6 set_isolation iso_strategy \
7  -domain PD
```

Object definition

Handle ID	/top/dut_i/PD
Properties	**Value**
upf_name	PD
upf_parent	/top/dut_i
upf_file	test.upf
upf_line	2
upf_creation_scope	/top/dut_i
upf_effective_extents	#UPFEXTENT1#
upf_supply_set_handles	{/top/dut_i/PD.primary}
upf_upper_boundary	{/top/dut_i/mid}
upf_isolation_strategies	{/top/dut_i/PD.iso_strategy}

10.6.1.2 Retention strategy

Class name	upfRetentionStrategyT	
Class membership	upfRetentionStrategyT, upfStrategyT, upfExtentClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>.<upf_name of Object>`	
Handle ID examples	`/top/dut_i/PD.ret1`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_effective_extents	upfExtentT	The upfExtentT object that points to the first element in the effective_element_list of corresponding UPF command (see 10.6.3.1 for more details)
upf_supply_set_handles	List of upfSupplySetT	The list of supply set handles defined on the object
upf_logic_refs	List of upfNamedRefT	The list of predefined names defined for the strategy
upf_is_no_retention	upfBooleanT	Flag for -no_retention
upf_save_condition	upfExpressionT	To capture -save_condition information
upf_restore_condition	upfExpressionT	To capture -restore_condtion information
upf_retention_condition	upfExpressionT	To capture -retention_condition information
upf_is_use_retention_as_primary	upfBooleanT	Flag for -use_retention_as_primary
upf_save_signal	upfSignalSenseT	Contains -save_signal information
upf_restore_signal	upfSignalSenseT	Contains -restore_signal information
upf_retention_parameters	upfRetentionParamE	Contains -parameter information

The object of upfRetentionStrategyT class is created when set_retention command is executed.

The following properties comprise the child namespaces:

a) upf_supply_set_handles
b) upf_logic_refs

The upf_supply_set_handles property will contain the predefined supply set handles **retention_supply** and **primary_supply** denoting the retention supply and primary supply respectively, of the retention registers.

The upf_logic_refs property will contain the predefined logic refs **save_signal** and **restore_signal** which denote the save and restore control signals.

The upf_parent property will point to the power domain in which this strategy was created.

The upf_creation_scope property will point to the HDL scope in which the power domain (parent) is created.

The upf_retention_parameters contains and enumerated value of type upfRetentionParamE as described in Table 15.

Table 15—Enumerated type upfRententionParamE

upfRetentionParamE	
Enumerated literals	**Description**
RET_SUP_COR	The enumerated literals map directly to values specified in -parameters option of set_retention command
NO_RET_SUP_COR	
SAV_RES_COR	
NO_SAV_RES_COR	

Examples

Upf source: test.upf

```
10 set_retention ret1 -domain PD \
11   -retention_supply PD.SSH1 \
12   -save_signal {ret_en negedge} \
13   -restore_signal {ret_en posedge} \
14   -retention_condition { !clk }
```

Object definition

Handle ID	/top/dut_i/PD.ret1
Properties	**Value**
upf_name	ret1
upf_parent	/top/dut_i/PD
upf_file	test.upf
upf_line	10
upf_creation_scope	/top/dut_i
upf_effective_extents	#UPFEXTENT1#
upf_supply_set_handles	{/top/dut_i/PD.ret1.retention_supply /top/dut_i/PD.ret1.primary_supply}
upf_logic_refs	{/top/dut_i/PD.ret1.save_signal /top/dut_i/PD.ret1.restore_signal}
upf_save_signal	#UPFSIGSENSE1#
upf_restore_signal	#UPFSIGSENSE2#
upf_retention_condition	#UPFEXPR1#

10.6.1.3 Isolation strategy

Class name	upfIsolationStrategyT	
Class membership	upfIsolationStrategyT, upfBoundaryStrategyT, upfStrategyT, upfExtentClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>.<upf_name of Object>`	
Handle ID examples	`/top/dut_i/PD.iso1`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_effective_extents	upfExtentT	The upfExtentT object that points to the first element in the effective_element_list of corresponding UPF command (see 10.6.3.1 for more details)
upf_supply_set_handles	List of upfSupplySetT	The list of supply set handles defined on the object
upf_logic_refs	List of upfNamedRefT	The list of predefined names defined for the strategy
upf_location	upfLocationE	Contains -location information
upf_applies_to	upfAppliesToFilterE	Contains -applies_to information
upf_source_filter	upfAbstractObjT	-source filter information
upf_sink_filter	upfAbstractObjT	-sink filter information
upf_name_prefix	upfStringT	-name_prefix information
upf_name_suffix	upfStringT	-name_suffix information
upf_is_use_equivalence	upfBooleanT	-use_equivalence information
upf_is_diff_supply_only	upfBooleanT	Flag for -diff_supply_only
upf_is_no_isolation	upfBooleanT	Flag for -no_isolation
upf_is_force_isolation	upfBooleanT	Flag for -force_isolation
upf_clamp_values	List of upfIsolationClampE	Information about -clamp_value
upf_user_clamp_values	List of upfStringT	Information about actual values when -clamp_value value is specified
upf_isolation_controls	List of upfSignalSenseT	Information about -isolation_signal

The object of upfIsolationStrategyT class is created when set_isolation command is executed.

The following properties comprise the child name spaces:

a) upf_supply_set_handles

b) upf_logic_refs

The upf_supply_set_handles property will contain the predefined supply set handle "**isolation_supply**" denoting the isolation supply set. If there are multiple supply sets defined on the strategy, then this list will accordingly contain those supply sets.

The upf_logic_refs property will contain the predefined logic refs **isolation_signal** which denotes the control signal specified in the isolation strategy. If there are multiple isolation controls specified then this list will be extended accordingly.

The property upf_user_clamp_values will be populated when -clamp value was specified in set_isolation or set_port_attributes commands. In this case, there will be a direct correspondence with the position of enumerated value specified in property upf_clamp_values and upf_user_clamp_values. In such case, if there are mixture of predefined clamp values and user defined clamp values, the standard values of 0, 1, any, Z, and latch will be used in upf_user_clamp_values for predefined clamp values.

NOTE—The upf_isolation_controls property will contain both the isolation control information and the sensitivity in the form of upfSignalSenseT object, whereas upf_logic_ref will point to the control signal via upfNamedRefT object.

Table 16—Enumerated type upfLocationE

upfLocationE	
Enumerated literals	**UPF mapping**
SELF	-location self
OTHER	-location other
PARENT	-location parent
AUTOMATIC	-location automatic
FANOUT	-location fanout

Table 17—Enumerated type upfAppliesToFilterE

upfAppliesToFilterE	
Enumerated literals	**UPF mapping**
UPF_FILTER_UNDEF	Undefined, when no information is available
UPF_FILTER_INPUTS	-applies_to inputs
UPF_FILTER_OUTPUTS	-applies_to outputs
UPF_FILTER_BOTH	-applies_to both

Table 18—Enumerated type upfPortDirE

upfPortDirE	
Enumerated literals	**UPF mapping**
UPF_DIR_UNDEF	Undefined, when no information is available
UPF_DIR_IN	-direction in
UPF_DIR_OUT	-direction out
UPF_DIR_INOUT	-direction inout

Table 19 —Enumerated type upfIsolationClampE

upfIsolationClampE	
Enumerated literals	**UPF mapping**
UPF_CLAMP_UNDEF	Undefined, when no information is available
UPF_CLAMP_ZERO	-*_clamp 0
UPF_CLAMP_ONE	-*_clamp 1
UPF_CLAMP_ZEE	-*_clamp Z
UPF_CLAMP_LATCH	-*_clamp latch
UPF_CLAMP_ANY	-*_clamp any
UPF_CLAMP_USER_VALUE	-*_clamp *value*

Examples

UPF Source: test.upf

```
20 set_isolation iso1 \
21     -domain PD \
22     -elements {a b c d} \
23     -isolation_supply {PD.SSH1} \
24     -clamp_value {1} \
25     -applies_to outputs \
26     -sink PD2 \
27     -isolation_signal cpu_iso \
28     -isolation_sense low -location parent
```

Object definition

Handle ID	/top/dut_i/PD.iso1
Properties	**Value**
upf_name	iso1
upf_parent	/top/dut_i/PD
upf_file	test.upf
upf_line	20
upf_creation_scope	/top/dut_i
upf_effective_extents	#UPFEXTENT1#
upf_supply_set_handles	{/top/dut_i/PD.isolation_supply}
upf_logic_refs	{/top/dut_i/PD.iso1.isolation_signal}
upf_clamp_values	{1}
upf_applies_to	UPF_FILTER_OUTPUTS
upf_sink_filter	/top/dut_i/PD2
upf_isolation_controls	{#UPFSIGSENSE2#}
upf_location	PARENT

10.6.1.4 Level-shifter strategy

Class name	upfLevelShifterStrategyT	
Class membership	upfLevelShifterStrategyT, upfBoundaryStrategyT, upfStrategyT, upfExtentClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>.<upf_name of Object>`	
Handle ID examples	`/top/dut_i/PD.ls1`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_effective_extents	upfExtentT	The upfExtentT object that points to the first element in the effective_element_list of corresponding UPF command (see 10.6.3.1 for more details)
upf_supply_set_handles	List of upfSupplySetT	The list of supply set handles defined on the object
upf_logic_refs	List of upfNamedRefT	The list of predefined names defined for the strategy
upf_location	upfLocationE	Contains -location information
upf_applies_to	upfAppliesToFilterE	Contains -applies_to information
upf_source_filter	upfAbstractObjT	-source filter information
upf_sink_filter	upfAbstractObjT	-sink filter information
upf_name_prefix	upfStringT	-name_prefix information
upf_name_suffix	upfStringT	-name_suffix information
upf_is_use_equivalence	upfBooleanT	-use_equivalence information
upf_is_no_shift	upfBooleanT	-no_shift information
upf_is_force_shift	upfBooleanT	-force_shift information
upf_threshold_value	upfRealT	-threshold information
upf_level_shift_rule	upfLevelShifterRuleE	-rule

The object of upfLevelShifterStrategyT class is created when set_level_shifter command is executed.

The following property comprises the child name spaces:

— upf_supply_set_handles

The upf_supply_set_handles property will contain the predefined supply set handles, input_supply, output_supply, and internal_supply.

The upf_logic_refs property will not be populated for objects of upfLevelShifterStrategyT type.

For possible values of upf_location property see Table 16.

For possible values of upf_applies_to property see Table 17.

For possible values of upf_level_shift_rule see Table 20.

Table 20 —Enumerated type upfLevelShifterRuleE

upfLevelShifterRuleE	
Enumerated literals	**UPF mapping**
UPF_LS_LOW_TO_HIGH	-rule low_to_high
UPF_LS_HIGH_TO_LOW	-rule high_to_low
UPF_LS_BOTH	-rule both

Examples

UPF source: test.upf

```
30 set_level_shifter ls1\
31     -domain PD \
32     -applies_to inputs \
33     -source PD.SSH1 \
34     -threshold 0.02 \
35     -rule both
```

Object definition

Handle ID	/top/dut_i/PD.ls1
Properties	**Value**
upf_name	ls1
upf_parent	/top/dut_i/PD
upf_file	test.upf
upf_line	30
upf_creation_scope	/top/dut_i
upf_effective_extents	#UPFEXTENT1#
upf_supply_set_handles	{/top/dut_i/PD.ls1.input_supply /top/dut_i/PD.ls1.output_supply /top/dut_i/PD.ls1.internal_supply}
upf_source_filter	/top/dut_i/PD.SSH1
upf_threshold_value	0.02
upf_applies_to	UPF_FILTER_INPUTS
upf_level_shift_rule	UPF_LS_BOTH

10.6.1.5 Repeater strategy

Class name	upfRepeaterStrategyT	
Class membership	upfRepeaterStrategyT, upfBoundaryStrategyT, upfStrategyT, upfExtentClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>.<upf_name of Object>`	
Handle ID examples	`/top/dut_i/PD.rs`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_effective_extents	upfExtentT	The upfExtentT object that points to the first element in the effective_element_list of corresponding UPF command (see 10.6.3.1 for more details)
upf_supply_set_handles	List of upfSupplySetT	The list of supply set handles defined on the object
upf_logic_refs	List of upfNamedRefT	The list of predefined names defined for the strategy
upf_location	upfLocationE	Not required for repeater strategy
upf_applies_to	upfAppliesToFilterE	Contains -applies_to information
upf_source_filter	upfAbstractObjT	-source filter information
upf_sink_filter	upfAbstractObjT	-sink filter information
upf_name_prefix	upfStringT	-name_prefix information
upf_name_suffix	upfStringT	-name_suffix information
upf_is_use_equivalence	upfBooleanT	-use_equivalence information

The object of upfRepeaterStrategyT class is created when set_repeater command is executed.

For possible values of upf_applies_to property see Table 3.

Examples

UPF source: test.upf

```
set_repeater repeat1\
   -domain PD \
   -applies_to outputs \
   -source PD.SSH1
```

Object definition

Handle ID	/top/dut_i/PD.repeat1
Properties	**Value**
upf_name	repeat1
upf_parent	/top/dut_i/PD
upf_file	test.upf
upf_line	36
upf_creation_scope	/top/dut_i
upf_effective_extents	#UPFEXTENT5#
upf_source_filter	/top/dut_i/PD.SSH1
upf_applies_to	UPF_FILTER_OUTPUTS

10.6.1.6 Supply set

Class name	upfSupplySetT	
Class membership	upfSupplySetT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`if [<Class Membership Of upf_parent> == upfHdlScopeT]` `  # Global Supply Set` `  <handle ID of upf_parent>/<upf_name of Object>` `else` `  # Local Supply Set` `  <handle ID of upf_parent>.<upf_name of Object>`	
Handle ID examples	`/top/dut_i/SS1        #Global Supply Set` `/top/dut_i/PD.primary #Local Supply Set`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_functions	List of upfNamedRefT	The functions of the supply net
upf_ss_states	List of upfPowerStateT	List of power states added by add_power_state command
upf_ss_transitions	List of upfPowerStateTransitionT	list of power state transitions defined by describe_state_transition
upf_equivalent_sets	List of upfSupplySetT	List of supply sets that are equivalent to the given supply set, this list contains only those supply sets that are marked as equivalent using set_equivalent, associate_supply_set command or similar such commands (e.g., set_isolation -isolation_supply)
Dynamic property (only available during simulation)		
upf_current_state	upfPowerStateT	The current state of the object during simulation

The object of upfSupplySetT class is created when create_supply_set command is executed or for supply set handles present on other UPF objects, e.g., power domain, retention strategy, etc.

If a create_supply_set command results in creation of this object, then that object is termed as global supply set. The handle ID of global supply set will be created differently than local supply set. In case of global supply set, the upf_parent property will be of HDL scope type (upfHdlScopeT).

A supply set created for supply set handles (either predefined or user defined) in UPF are termed as local supply set. In case of local supply set, the upf_parent property is the UPF object on which the supply set handle was created; e.g., PD.primary.

The upf_functions property will denote the functions of a supply set. This will contain objects of type upfNamedRefT which will point to the associated supply nets.

The following property comprises the child name spaces:

— upf_functions

The upf_functions property will consist of six predefined functions, **power**, **ground**, **nwell**, **pwell**, **deeppwell**, **deepnwell**. However, only the required functions will be populated in the object.

In case of UPF 2.0, the functions may contain user defined functions as well. In such case, the object will contain corresponding upfNamedRefT objects with appropriate flag set. This will not be the case with UPF 2.1 onwards.

The upf_file and upf_line properties will be populated for global supply sets only.

Examples

UPF source: test.upf

```
40 create_supply_set SS1
41     -function {power vdd}
42     -function {ground vss}
43 associate_supply_set SS1 -handle PD.primary
```

Object definition

Handle ID	/top/dut_i/SS1
Properties	**Value**
upf_name	SS1
upf_parent	/top/dut_i
upf_file	test.upf
upf_line	40
upf_creation_scope	/top/dut_i
upf_functions	{/top/dut_i/SS1.power /top/dut_i/SS1.ground}
upf_equivalent_sets	{/top/dut_i/SS1 /top/dut_i/PD.primary}

Object definition

Handle ID	/top/dut_i/PD.primary
Properties	**Value**
upf_name	primary
upf_parent	/top/dut_i/PD
upf_creation_scope	/top/dut_i
upf_functions	{/top/dut_i/PD.primary.power /top/dut_i/PD.primary.ground}
upf_equivalent_sets	{/top/dut_i/SS1 /top/dut_i/PD.primary}

10.6.1.7 Named object reference

Class name	upfNamedRefT	
Class membership	upfNamedRefT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>.<upf_name of Object>`	

Handle ID examples	`/top/dut_i/PD.primary.power    #Ref to Supply Set Function`
	`/top/dut_i/PD.ret1.save_signal #Ref to Strategy Control Signal`	

Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_ref_kind	upfNamedRefKindE	Enumerated value representing kind of reference, e.g., retention save
upf_ref_object	upfBaseNamedT	Reference to original UPF object, e.g., Logic Net

The object of upfNamedRefT class is created in the following scenarios:

a) Functions of supply set

b) Logic references (predefined names to refer to controls) of strategies

This object provides a named reference to some other object, supply nets in case of functions and logic nets in case of logic refs. The original object can be accessed by upf_ref_object property.

For possible values of upf_ref_kind property see Table 21.

Table 21 —Enumerated type upfNamedRefKindE

upfNamedRefKindE	
Enumerated literals	**UPF mapping**
UPF_REF_POWER	power function of supply set
UPF_REF_GROUND	ground function of supply set
UPF_REF_PWELL	pwell function of supply set
UPF_REF_NWELL	nwell function of supply set
UPF_REF_DEEPPWELL	deeppwell function of supply set
UPF_REF_DEEPNWELL	deepnwell function of supply set
UPF_REF_ISO_SIGNAL	reference to isolation control signal in set_isolation
UPF_REF_SAVE_SIGNAL	reference to save_signal in set_retention
UPF_REF_RESTORE_SIGNAL	reference to restore_signal in set_retention
UPF_REF_GENERIC_CLOCK	reference to UPF_GENERIC_CLOCK in set_retention
UPF_REF_GENERIC_DATA	reference to UPF_GENERIC_DATA in set_retention
UPF_REF_GENERIC_ASYNC_LOAD	reference to UPF_GENERIC_ASYNC_LOAD in set_retention
UPF_REF_GENERIC_OUTPUT	reference to UPF_GENERIC_OUTPUT in set_retention
UPF_REF_USER_DEFINED	some user defined ref handle

Examples

UPF source: test.upf

```
create_supply_set SS1 \
    -function {power vdd} \
    -function {ground vss}
set_retention ret1 -domain PD \
    -save_signal {ret_ctrl high} …
```

Object definition

Handle ID	/top/dut_i/SS1.power
Properties	**Value**
upf_name	power
upf_parent	/top/dut_i/SS1
upf_creation_scope	/top/dut_i
upf_ref_kind	UPF_REF_POWER
upf_ref_object	/top/dut_i/vdd

Object definition

Handle ID	/top/dut_i/PD.ret1.save_signal
Properties	**Value**
upf_name	save_signal
upf_parent	/top/dut_i/PD.ret1
upf_creation_scope	/top/dut_i
upf_ref_kind	UPF_REF_RET_SAVE_SIGNAL
upf_ref_object	/top/dut_i/ret_ctrl

10.6.1.8 Supply net

Class name	upfSupplyNetT	
Class membership	upfSupplyNetT, upfNetClassT, upfNetworkClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>/<upf_name of Object>[@upfSupplyNetT]`	
Handle ID examples	`/top/dut_i/vddnet       #Supply Net with unique name "vddnet"` `/top/dut_i/vdd@upfSupplyNetT #Same name as supply port "vdd"`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_hdl_implementation	upfHdlDeclT	The HDL object which is pre-implemented and already present
upf_root_driver	upfNetworkClassT	The upfNetworkClassT object which is the root supply driver (see 4.5). This will not be populated for root drivers themselves.
upf_fanin_conn	List of upfPortClassT	Contains the list of ports driving the net
upf_fanout_conn	List of upfPortClassT	Contains the list of ports that are receiving the value of the net
upf_resolve_type	upfResolveE	Enumerated value representing supply net resolution
Dynamic property (only available during simulation)		
upf_current_value	upfSupplyTypeT	The current value of the object during simulation

The object of upfSupplyNetT class is created when create_supply_net command is executed.

If a supply net is already present in HDL, then the property upf_hdl_implementation will point to the HDL object representing supply net in HDL.

For values of upf_resolve_type property see Table 22:

Table 22—Enumerated type upfResolveE

upfResolveE	
Enumerated literals	**UPF mapping**
UNRESOLVED	-resolve unresolved
ONE_HOT	-resolve one_hot
PARALLEL	-resolve parallel
PARALLEL_ONE_HOT	-resolve parallel_one_hot

Examples

UPF source: test.upf

```
50 create_supply_net vdd
```

Object definition

Handle ID	/top/dut_i/vdd@upfSupplyNetT
Properties	**Value**
upf_name	vdd
upf_parent	/top/dut_i
upf_file	test.upf
upf_line	50
upf_creation_scope	/top/dut_i
upf_root_driver	/top/dut_i/vdd
upf_fanin_conn	{/top/dut_i/vdd}
upf_resolve_type	UNRESOLVED

10.6.1.9 Supply port

Class name	upfSupplyPortT	
Class membership	upfSupplyPortT, upfPortClassT, upfNetworkClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>/<upf_name of Object>`	

Handle ID examples	`/top/dut_i/vdd   #Created in HDL Scope "dut_i"`
	`/top/dut_i/sw/ip #Created in power switch "sw"`	

Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_hdl_implementation	upfHdlDeclT	The HDL object which is pre-implemented and already present
upf_root_driver	upfNetworkClassT	The upfNetworkClassT object which is the root supply driver (see 4.5). This will not be populated for root drivers themselves.
upf_network_attributes	List of upfAttributeT	The different attributes added on the object via set_port_attributes or other equivalent means
upf_hiconn	List of upfNetworkClassT	Contains the list of objects connected to the hiconn side of the port resulting from the application of connect_supply_net and connect_supply_set commands
upf_loconn	List of upfNetworkClassT	Contains the list of objects connected to the loconn side of the port resulting from the application of connect_supply_net and connect_supply_set commands
upf_port_dir	upfPortDirE	The direction of the port
upf_sp_states	List of upfPortStateT	The port states added by add_port_state command
Dynamic property (only available during simulation)		
upf_current_value	upfSupplyTypeT	The current value of the object during simulation

The object of upfSupplyPortT class is created in the following scenarios:

a) create_supply_port command is executed.

b) create_power_switch is creating the input/output supply ports

The following properties comprise the child namespaces:

a) upf_sp_states

For possible values of upf_port_dir property see Table 18.

Examples

UPF source: test.upf

```
70 create_supply_port vdd \
71    -direction input
```

Object definition

Handle ID	/top/dut_i/vdd
Properties	**Value**
upf_name	vdd
upf_parent	/top/dut_i
upf_file	test.upf
upf_line	70
upf_creation_scope	/top/dut_i
upf_loconn	{/top/dut_i/vdd@upfSupplyNetT}
upf_port_dir	UPF_DIR_IN

10.6.1.10 Logic net

Class name	upfLogicNetT	
Class membership	upfLogicNetT, upfNetClassT, upfNetworkClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>/<upf_name of Object>[@upfLogicNetT]`	
Handle ID examples	`/top/dut_i/ctrl_iso`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_hdl_implementation	upfHdlDeclT	The HDL object which is pre-implemented and already present
upf_root_driver	upfNetworkClassT	The upfNetworkClassT object which is driving the current object. This will not be populated for root drivers themselves.
upf_fanin_conn	List of upfPortClassT	Contains the list of ports driving the net
upf_fanout_conn	List of upfPortClassT	Contains the list of ports that are receiving the value of the net
Dynamic property (only available during simulation)		
upf_current_value	upfBooleanT	The current value of the object during simulation

The object of upfLogicNetT class is created when create_logic_net command is executed.

Examples

UPF source: test.upf

```
80 create_logic_net sig
```

Object definition

Handle ID	/top/dut_i/sig
Properties	**Value**
upf_name	sig
upf_parent	/top/dut_i
upf_file	test.upf
upf_line	80
upf_creation_scope	/top/dut_i

10.6.1.11 Logic port

Class name	upfLogicPortT	
Class membership	upfLogicPortT, upfPortClassT, upfNetworkClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>/<upf_name of Object>`	

Handle ID examples	`/top/dut_i/ctrl_iso    #Created in HDL Scope "dut_i"`
	`/top/dut_i/sw/ctrl_sw #Created in power switch "sw"`	

Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_hdl_implementation	upfHdlDeclT	The HDL object which is pre-implemented and already present
upf_root_driver	upfNetworkClassT	The upfNetworkClassT object which is driving the current object. This will not be populated for root drivers themselves.
upf_network_attributes	List of upfAttributeT	The different attributes added on the object via set_port_attributes or other equivalent means
upf_hiconn	List of upfNetworkClassT	Contains the list of objects connected to the hiconn side of the port resulting from the application of connect_logic_net or equivalent commands
upf_loconn	List of upfNetworkClassT	Contains the list of objects connected to the loconn side of the port resulting from the application of connect_logic_net or equivalent commands
upf_port_dir	upfPortDirE	The direction of the port
Dynamic property (only available during simulation)		
upf_current_value	upfBooleanT	The current value of the object during simulation

The object of upfLogicPortT class is created in the following scenarios:

a) create_logic_port command is executed.

b) create_power_switch creates a control port.

For possible values of upf_port_dir property see Table 18.

Examples

UPF source: test.upf

```
85 create_logic_port iso_ctrl \
86    -direction input
```

Object definition

Handle ID	/top/dut_i/iso_ctrl
Properties	**Value**
upf_name	iso_ctrl
upf_parent	/top/dut_i
upf_file	test.upf
upf_line	85
upf_creation_scope	/top/dut_i
upf_port_dir	UPF_DIR_IN

10.6.1.12 Power switch

Class name	upfPowerSwitchT	
Class membership	upfPowerSwitchT, upfExtentClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf parent>/<upf name of Object>`	
Handle ID examples	`/top/dut i/sw`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_effective_extents	upfExtentT	The upfExtentT object that points to the first element in the list resulting from -instance option of create_power_switch command
upf_supply_set_handles	List of upfSupplySetT	The list containing the supply set which is specified by -supply_set option of create_power_switch command
upf_output_supply_port	upfSupplyPortT	The supply port created by -output_supply_port option of create_power_switch command
upf_input_supply_ports	List of upfSupplyPortT	The supply ports created by -input_supply_port option of create_power_switch command
upf_control_ports	List of upfLogicPortT	The logic ports created by -control_port option of create_power_switch command
upf_ack_ports	List of upfAckPortT	The ack port created by -ack_port option of create_power_switch command
upf_sw_states	List of upfPowerSwitchStateT	The list of switch states created by create_power_switch -on_state, -partial_on_state and -error_state

The object of upfPowerSwitchT class is created when create_power_switch command is executed.

The following properties comprise the child namespaces:

a) upf_supply_set_handles
b) upf_output_supply_port
c) upf_input_supply_ports
d) upf_control_ports
e) upf_ack_ports
f) upf_sw_states

Examples

UPF source: test.upf

```
80 create_power_switch sw1 \
81    -output_supply_port {vout vdd_sw} \
82    -input_supply_port {vin vdd} \
83    -control_port {ss_ctrl sw_en} \
84    -on_state {ss_on vin {ss_ctrl}} \
85    -off_state {ss_off {!ss_ctrl}}
```

Object definition

Handle ID	/top/dut_i/sw1
Properties	**Value**
upf_name	sw1
upf_parent	/top/dut_i
upf_file	test.upf
upf_line	80
upf_creation_scope	/top/dut_i
upf_output_supply_port	/top/dut_i/sw1/vout
upf_input_supply_ports	{/top/dut_i/sw1/vin}
upf_control_ports	{/top/dut_i/sw1/ss_ctrl}
upf_sw_states	{/top/dut_i/sw1.ss_on /top/dut_i/sw1.ss_off}

10.6.1.13 Ack port

Class name	upfAckPortT	
Class membership	upfAckPortT, upfPortClassT, upfNetworkClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>/<upf_name of Object>`	
Handle ID examples	`/top/dut_i/sw/ack`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_hdl_implementation	upfHdlDeclT	The HDL object which is pre-implemented and already present
upf_root_driver	upfNetworkClassT	The upfNetworkClassT object which is driving the current object. This will not be populated for root drivers themselves.
upf_hiconn	List of upfNetworkClassT	Contains the list of objects connected to the hiconn side of the port resulting from the application of connect_logic_net or equivalent commands
upf_port_dir	upfPortDirE	The direction of the port
upf_ack_delay	upfStringT	The ack delay specified in UPF command
Dynamic property (only available during simulation)		
upf_current_value	upfBooleanT	The current value of the object during simulation

The object of upfAckPortT class is created when create_power_switch command is executed with the -ack_port option specified.

The upf_port_dir property will always be having upf_dir_out (see Table 18) as value for upfAckPortT.

Examples

UPF source: test.upf

```
create_power_switch sw2 \
   -output_supply_port {vout vdd_sw} \
   -input_supply_port {vin vdd} \
   -control_port {ss_ctrl sw_en} \
   -on_state {ss_on vin {ss_ctrl}} \
   -off_state {ss_off {!ss_ctrl}} \
   -ack_port {ts_ack ack} \
   -ack_delay {ts_ack 100ns} \
   -supply_set ss_aon
```

Object definition

Handle ID	/top/dut_i/sw2/ts_ack
Properties	**Value**
upf_name	ts_ack
upf_parent	/top/dut_i/sw2
upf_file	test.upf
upf_line	90
upf_creation_scope	/top/dut_i
upf_hiconn	{/top/dut_i/ack}
upf_ack_delay	100ns

10.6.1.14 Power state

Class name	upfPowerStateT	
Class membership	upfPowerStateT, upfStateClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>.<upf_name of Object>`	
Handle ID examples	`/top/dut_i/PD.drowsy`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_is_illegal	upfBooleanT	Will have value true when -illegal is specified in corresponding add_power_state command
upf_logic_expr	upfExpressionT	The expression specified by -logic_expr option of add_power_state command
upf_supply_expr	upfExpressionT	The expression specified by -supply_expr option of add_power_state command
upf_simstate	upfSimstateE	The value specified by -simstate option of add_power_state command
Dynamic property (only available during simulation)		
upf_is_active	upfBooleanT	The is_active is true when the state is active at a specific time during simulation.

The object of upfPowerStateT class is created when add_power_state command is executed. The handle of the object on which the power state has been added is present in upf_parent property on the object. The upf_simstate and upf_supply_expr properties will only be present on states added on objects of upfSupplySetT class.

For possible values of upf_simstate property, from highest to lowest priority, see Table 23.

Table 23 —Enumerated type upfSimstateE

upfSimstateE	
Enumerated literals	**UPF mapping**
CORRUPT	-simstate CORRUPT or -simstate NOT_NORMAL
CORRUPT_ON_ACTIVITY	-simstate CORRUPT_ON_ACTIVITY or -simstate NOT_NORMAL
CORRUPT_ON_CHANGE	-simstate CORRUPT_ON_CHANGE or -simstate NOT_NORMAL
CORRUPT_STATE_ON_ACTIVITY	-simstate CORRUPT_STATE_ON_ACTIVITY or -simstate NOT_NORMAL
CORRUPT_STATE_ON_CHANGE	-simstate CORRUPT_STATE_ON_CHANGE or -simstate NOT_NORMAL
NORMAL	-simstate NORMAL

Examples

UPF source: test.upf

```
110 add_power_state PD \
111    -state {S1 -logic_expr {PD.primary == ON}}
```

Object definition

Handle ID	/top/dut_i/PD.S1
Properties	**Value**
upf_name	S1
upf_parent	/top/dut_i/PD
upf_file	test.upf
upf_line	110
upf_creation_scope	/top/dut_i
upf_logic_expr	#UPFEXPR1#

10.6.1.15 Power switch state

Class name	upfPowerSwitchStateT	
Class membership	upfPowerSwitchStateT, upfStateClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>.<upf_name of Object>`	
Handle ID examples	`/top/dut_i/sw.sw_on`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_is_illegal	upfBooleanT	This will be true when state is created from -error_state option of create_power_switch command
upf_switch_expr	upfExpressionT	The boolean expr specified in -on_state, -on_partial_state, -off_state and -error_state options of create_power_switch command
upf_switch_output_state	upfSupplyStateE	Captures the state of the output of the switch, i.e. -on_state has state as FULL_ON, -off_state has state as OFF, -on_partial_state has state as PARTIAL_ON, -error_state has state as UNDETERMINED
upf_input_supply_port	upfSupplyPortT	The handle of input supply port which will be connected when the state is on or partial_on

The object of upfPowerSwitchStateT is created when a create_power_switch command is executed. The object is created to capture information about the following options:

a) -on_state

b) -on_partial_state

c) -off_state

d) -error_state

This object has no child namespace.

The property upf_switch_output_state maintains information about the output of the switch when the state is active. For possible values of upf_switch_output_state property see Table 24.

Table 24—Enumerated type upfSupplyStateE

upfSupplyStateE	
Enumerated literals	**UPF mapping**
OFF	-off_state or reference to OFF for supply net/port state
FULL_ON	-on_state or reference to FULL_ON for supply net/port state
PARTIAL_ON	-on_partial_state or reference to PARTIAL_ON for supply net/port state
UNDETERMINED	-error_state or reference to UNDETERMINED for supply net/port state

Examples

UPF source: test.upf

```
110 create_power_switch simple_switch2 \
111   -output_supply_port {vout VDD_SW} \
112   -input_supply_port {vin VDD} \
113   -control_port {ss_ctrl sw_ena} \
114   -on_state {ss_on vin {ss_ctrl}} \
115   -off_state {ss_off {!ss_ctrl}}
```

Object definition

Handle ID	/top/dut_i/simple_switch2.ss_on
Properties	**Value**
upf_name	ss_on
upf_parent	/top/dut_i/simple_switch2
upf_file	test.upf
upf_line	110
upf_creation_scope	/top/dut_i
upf_switch_expr	#UPFEXPR1#
upf_switch_output_state	FULL_ON
upf_input_supply_port	/top/dut_i/simple_switch2/vin

10.6.1.16 Supply port state

Class name	upfSupplyPortStateT	
Class membership	upfSupplyPortStateT, upfStateClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>.<upf_name of Object>`	
Handle ID examples	`/top/dut_i/vdd.on1V`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	The UPF object on which the state was created
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_supply_state	upfSupplyStateE	Not required for states which are "*" (don't cares) in PSTs
upf_volt_min	upfRealT	The min voltage value
upf_volt_nom	upfRealT	The nominal voltage value
upf_volt_max	upfRealT	The maximum voltage value
upf_volt_kind	upfVoltKindE	The enum specifying whether nom, doublet or triplet was specified in UPF

The object of upfSupplyPortStateT class is created when add_port_state command is executed. This object is also created to capture information about reference to "*" don't care state in PSTs.

The upf_is_illegal property will not be populated for objects of this class.

The upf_volt_kind property captures information whether user had specified just the nominal voltage or doublet or triplet. For possible values of upf_volt_kind property, see Table 25.

Table 25—Enumerated type upfVoltKindE

upfVoltKindE	
Enumerated literals	**UPF mapping**
NOM	When only nominal value is specified in add_port_state command
DOUBLET	When a doublet is specified in add_port_state command
TRIPLET	When a triplet is specified in add_port_state command

Examples

UPF source: test.upf

```
110 add_port_state vdd -state {on1V 1.0}
```

Object definition

Handle ID	/top/dut_i/vdd.on1V
Properties	**Value**
upf_name	on1V
upf_parent	/top/dut_i/vdd
upf_file	test.upf
upf_line	110
upf_creation_scope	/top/dut_i
upf_supply_state	FULL_ON
upf_volt_nom	1.0
upf_volt_kind	NOM

10.6.1.17 PST state

Class name	upfPstStateT	
Class membership	upfPstStateT, upfStateClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>.<upf_name of Object>`	
Handle ID examples	`/top/dut_i/chip_pst.chip_on`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	The UPF object on which the state was created
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_is_illegal	upfBooleanT	Not required for PST states
upf_supply_states	List of upfSupplyPortStateT	The list of port states specified by add_pst_state command
Dynamic property (only available during simulation)		
upf_is_active	upfBooleanT	The is_active is true when the state is active at a specific time during simulation

The object of upfPstStateT class is created when add_pst_state command is executed.

Examples

UPF source: test.upf

```
110 add_pst_state chip_on -state {on1V off}
```

Object definition

Handle ID	/top/dut_i/chip_pst.chip_on
Properties	**Value**
upf_name	chip_on
upf_parent	/top/dut_i/chip_pst
upf_file	test.upf
upf_line	110
upf_creation_scope	/top/dut_i
upf_supply_states	{/top/dut_i/vdd.on1V /top/dut_i/vdd1.off}

10.6.1.18 PST

Class name	upfPowerStateTableT	
Class membership	upfPowerStateTableT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>/<upf_name of Object>`	
Handle ID examples	`/top/dut_i/chip_pst`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_pst_states	List of upfPstStateT	The states created by add_pst_state command
upf_pst_header	List of upfBaseNamedT	The list of supplies forming column of PST
Dynamic property (only available during simulation)		
upf_current_state	upfPowerStateT	The current state of the object during simulation

The object of upfPowerStateTableT class is created when create_pst command is executed.

The following property comprises child namespace:

— upf_pst_states

Examples

UPF source: test.upf

```
110 create_pst chip_pst -supplies {vdd vdd1}
```

Object definition

Handle ID	/top/dut_i/chip_pst
Properties	**Value**
upf_name	chip_pst
upf_parent	/top/dut_i
upf_file	test.upf
upf_line	110
upf_creation_scope	/top/dut_i
upf_pst_states	{/top/dut_i/chip_pst.chip_on}
upf_pst_header	{/top/dut_i/vdd /top/dut_i/vdd1}

10.6.1.19 Power state transition

Class name	upfPowerStateTransitionT	
Class membership	upfPowerStateTransitionT, upfStateClassT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>.<upf_name of Object>`	
Handle ID examples	`/top/dut_i/PdA.turn_on`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_is_illegal	upfBooleanT	The legality of transition as specified in describe_state_transition
upf_to_states	List of upfPowerStateT	The list of states determined from processing of describe_state_transition command
upf_from_states	List of upfPowerStateT	The list of states determined from processing of describe_state_transition command

The object of this class is created when describe_state_transition command is executed.

This object does not have any child namespaces.

Examples

UPF source: test.upf

```
110 describe_state_transition turn_on
111    -object PdA -from {SLEEP_MODE} \
112    -to {HIGH_SPEED_MODE} -illegal
```

Object definition

Handle ID	/top/dut_i/PdA.turn_on
Properties	**Value**
upf_name	turn_on
upf_parent	/top/dut_i/PdA
upf_file	test.upf
upf_line	110
upf_creation_scope	/top/dut_i
upf_is_illegal	true
upf_to_states	{/top/dut_i/PdA.HIGH_SPEED_MODE}
upf_from_states	{/top/dut_i/PdA.SLEEP_MODE}

10.6.1.20 Composite domain

Class name	upfCompositeDomainT	
Class membership	upfCompositeDomainT, upfBaseUpfT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>/<upf_name of Object>`	
Handle ID examples	`/top/dut_i/CD`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_file	upfStringT	Filename where object was created
upf_line	upfIntegerT	Line number where object was created
upf_creation_scope	upfBaseHdlT	The HDL scope in which the object was created
upf_supply_set_handles	List of upfSupplySetT	The list of supply set handles defined on the object
upf_pd_states	List of upfPowerStateT	List of states defined on composite domain
upf_pd_state_transitions	List of upfPowerStateTransitionT	List of power state transitions defined by `describe_state_transition` upf command
upf_subdomains	List of upfBaseUpfT	List of subdomains that belong to the composite domain, it can only be upfPowerDomainT or upfCompositeDomainT
Dynamic property (only available during simulation)		
upf_current_state	upfPowerStateT	The current state of the object during simulation

The object of this class is created when create_composite_domain command is executed.

Examples

UPF source: test.upf

```
110 create_composite_domain CD \
111    -subdomains {dut_i/pd1 dut_i/pd2} \
112    -supply {primary SS_system}
```

Object definition

Handle ID	/top/dut_i/CD
Properties	**Value**
upf_name	CD
upf_parent	/top/dut_i
upf_file	test.upf
upf_line	110
upf_creation_scope	/top/dut_i
upf_supply_set_handles	{/top/dut_i/CD.primary}
upf_subdomains	{/top/dut_i/pd1 /top/dut_i/pd2}

10.6.2 HDL objects

10.6.2.1 HDL scope

Class name	upfHdlScopeT	
Class membership	upfHdlScopeT, upfBaseHdlT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>/<upf_name of Object>`	
Handle ID examples	`/top/dut_i`	
Property	**Return value**	**description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_cell_info	upfCellT	The information about cell modeled at this HDL object
upf_hdl_attributes	List of upfAttributeT	The different attributes added on the object via set_design_attributes or other equivalent means
upf_extents	List of upfExtentT	The various upfExtentT pointing to this HDL object
upf_hdl_ports	List of upfHdlPortClassT	List of ports on the HDL instance that have PA information
upf_hdl_items	List of upfHdlNetClassT	List of nets on the HDL instance that have PA information or are used in power management
upf_items	List of upfBaseUpfT	List of UPF objects created in this scope
upf_child_instances	List of upfHdlScopeT	List of child instances

The object of upfHdlScopeT represents an instance in the logic hierarchy.

NOTE—Only instances which are necessary for capturing the power intent are present as upfHdlScopeT in the information model. There can be more instances in the actual design hierarchy but not present in the information model as they do not participate in the power management. See UPF information model and other HDL information models (e.g., VPI, VHPI, etc.) for more details.

The following properties comprise the child name spaces:

a) upf_hdl_ports
b) upf_hdl_items
c) upf_items
d) upf_child_instances

Examples

UPF source: test.upf

```
set_scope /top/dut_i
create_power_domain PD -elements {.}
```

Object definition

Handle ID	/top/dut_i
Properties	**Value**
upf_name	dut_i
upf_parent	/top
upf_extents	{#UPFEXTENT1#}
upf_hdl_ports	{/top/dut_i/port1 /top/dut_i/port2}
upf_hdl_items	{/top/dut_i/ctrl}
upf_items	{/top/dut_i/PD /top/dut_i/vdd}
upf_child_instances	{/top/dut_i/mid}

10.6.2.2 HDL scalar port

Class name	upfHdlPortBitT	
Class membership	upfHdlPortBitT, upfHdlPortClassT, upfHdlDeclT, upfBaseHdlT, upfBaseNamedT, upfBaseT	

Handle ID	`if [<Class Membership Of upf_parent> == upfHdlScopeT]`
	`  <handle ID of upf_parent>/<upf_name of Object>`
	`else`
	`  # Bit of Multi-bit object`
	`  <handle ID of upf_parent><upf_name of Object>`	

Handle ID examples	`/top/dut_i/port1`
	`/top/dut_i/port2[1]`
	`/top/dut_i/complexport@3`	

Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_cell_info	upfCellT	The information about cell modeled at this HDL object
upf_hdl_attributes	List of upfAttributeT	The different attributes added on the object via set_port_attributes or other equivalent means
upf_extents	List of upfExtentT	The various upfExtentT pointing to this HDL object
upf_port_dir	upfPortDirE	The direction of the port
upf_normalized_idx	upfIntegerT	The normalized index of the bit object when the object is acting as a bit of a upfMultiBitPortT object
upf_smallest_atomic_slice	upfHdlMultiBitSliceT	The handle of the smallest slice that represents and atomic object in HDL that can be represented as a valid RTL name

The object of upfHdlPortBitT represents a scalar port in the logic hierarchy. This object will represent any scalar ports that have single bit representation or a bit of a multi-bit port.

For possible values of upf_port_dir property see Table 3.

The upf_normalized_idx property is populated when the object is representing a bit of a multi-bit type object. The upf_smallest_atomic_slice property is only populated in special cases where an atomic type in HDL is requires multiple bits to represent the value. Please refer to 10.4.3.3 for more details.

Examples: Scalar port

UPF source: test.upf

```
70 set_isolation iso -domain PD -elements {port1}
```

Object definition

Handle ID	/top/dut_i/port1
Properties	**Value**
upf_name	port1
upf_parent	/top/dut_i
upf_extents	{#UPFEXTENT1#}
upf_port_dir	UPF_DIR_OUT

Examples: Bit of a multi-bit port

UPF source: test.upf

```
70 set_isolation iso -domain PD -elements {port2[2]}
```

Object definition

Handle ID	/top/dut_i/port2[2]
Properties	**Value**
upf_name	[2]
upf_parent	/top/dut_i/port2
upf_extents	{#UPFEXTENT2#}
upf_port_dir	UPF_DIR_IN
upf_normalized_idx	2

Examples: Bit of a multi-bit VHDL record

UPF source: test.upf

```
70 set_isolation iso -domain PD -elements {complexport.f1}
```

Object definition

Handle ID	/top/dut_i/complexport@3
Properties	**Value**
upf_name	@3
upf_parent	/top/dut_i/complexport
upf_extents	{#UPFEXTENT3#}
upf_port_dir	UPF_DIR_IN
upf_normalized_idx	3
upf_smallest_atomic_slice	/top/dut_i/complexport@31:0

10.6.2.3 HDL multi-bit port

Class name	upfHdlPortMultiBitT	
Class membership	upfHdlPortMultiBitT, upfHdlPortClassT, upfHdlDeclT, upfBaseHdlT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>/<upf_name of Object>`	
Handle ID examples	`/top/dut_i/port2`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_cell_info	upfCellT	The information about cell modeled at this HDL object
upf_hdl_attributes	List of upfAttributeT	The different attributes added on the object via set_port_attributes or other equivalent means
upf_extents	List of upfExtentT	The various upfExtentT pointing to this HDL object
upf_port_dir	upfPortDirE	The direction of the port
upf_hdl_width	upfIntegerT	Size of the port in number of bits
upf_normalized_bits	List of upfHdlPortBitT	List of paHdlPortBitT objects corresponding to each normalized width

The object of upfHdlPortMultiBitT represents a multi-bit object in the information model. This can be a vector, multi-dimensional array, or any other complex type object which requires multiple bits to represent. The multi-bit object contains bits which are normalized in the form `width-1 downto 0` (see 10.4.3.3.2).

For possible values of upf_port_dir property see Table 3.

Examples

UPF source: test.upf

```
70 set_isolation iso -domain PD -elements {port2}
```

Object definition

Handle ID	/top/dut_i/port2
Properties	**Value**
upf_name	port2
upf_parent	/top/dut_i
upf_extents	{#UPFEXTENT1#}
upf_port_dir	UPF_DIR_OUT
upf_hdl_width	3
upf_normalized_bits	{/top/dut_i/port2@0 /top/dut_i/port2@1 /top/dut_i/port2@2}

10.6.2.4 HDL scalar net

Class name	upfHdlNetBitT	
Class membership	upfHdlNetBitT, upfHdlNetClassT, upfHdlDeclT, upfBaseHdlT, upfBaseNamedT, upfBaseT	

Handle ID	`if [<Class Membership Of upf_parent> == upfHdlScopeT]`
	`  <handle ID of upf_parent>/<upf_name of Object>`
	`else`
	`  # Bit of Multi-bit object`
	`  <handle ID of upf_parent><upf_name of Object>`	

Handle ID examples	`/top/dut_i/net1`
	`/top/dut_i/net2[1]`
	`/top/dut_i/complexnet@3`	

Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_cell_info	upfCellT	The information about cell modeled at this HDL object
upf_hdl_attributes	List of upfAttributeT	The different attributes added on the object via set_port_attributes or other equivalent means
upf_extents	List of upfExtentT	The various upfExtentT pointing to this HDL object
upf_normalized_idx	upfIntegerT	The normalized index of the bit object when the object is acting as a bit of a upfMultiBitNetT object
upf_smallest_atomic_slice	upfHdlMultiBitSliceT	The handle of the smallest slice that represents and atomic object in HDL that can be represented as a valid RTL name

The object of upfHdlNetBitT represents a scalar port in the logic hierarchy. This object will represent any scalar nets that have single bit representation or a bit of a multi-bit net.

The upf_normalized_idx property is populated when the object is representing a bit of a multi-bit type object. The upf_smallest_atomic_slice property is only populated in special cases where an atomic type in HDL is requires multiple bits to represent the value. Please refer to 10.4.3.3 for more details.

Examples: Scalar port

UPF source: test.upf

```
set_isolation iso \
   -domain PD \
   -isolation_signal ctrl \
   -isolation_sense high
```

Object definition

Handle ID	/top/dut_i/ctrl
Properties	**Value**
upf_name	ctrl
upf_parent	/top/dut_i

Examples: Bit of a multi-bit port

UPF source: test.upf

```
set_isolation iso -domain PD -isolation_signal ctrl[2] \
   -isolation_sense high ...
```

Object definition

Handle ID	/top/dut_i/ctrl[2] or /top/dut_i/ctrl@2
Properties	**Value**
upf_name	[2] or @2
upf_parent	/top/dut_i/ctrl
upf_normalized_idx	2

10.6.2.5 HDL multibit net

Class name	upfHdlNetMultiBitT	
Class membership	upfHdlNetMultiBitT, upfHdlNetClassT, upfHdlDeclT, upfBaseHdlT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent>/<upf_name of Object>`	
Handle ID examples	`/top/dut_i/reg_arr`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_cell_info	upfCellT	The information about cell modeled at this HDL object
upf_hdl_attributes	List of upfAttributeT	The different attributes added on the object via set_port_attributes or other equivalent means
upf_extents	List of upfExtentT	The various upfExtentT pointing to this HDL object
upf_hdl_width	upfIntegerT	Size of the net in number of bits
upf_normalized_bits	List of upfHdlNetBitT	List of paHdlNetBitT objects corresponding to each normalized width

The object of upfHdlNetMultiBitT represents a multi-bit net object in the information model. This can be a vector, multi-dimensional array or any other complex type object which requires multiple bits to represent. The multi-bit object contains bits which are normalized in the form `width-1 downto 0`. Please refer to 10.4.3.3 for more details.

Examples

UPF source: test.upf

```
70 set_retention ret -domain PD -elements {reg_arr}
```

Object definition

Handle ID	/top/dut_i/reg_arr
Properties	**Value**
upf_name	reg_arr
upf_parent	/top/dut_i
upf_extents	{#UPFEXTENT6# }
upf_hdl_width	3
upf_normalized_bits	{/top/dut_i/reg_arr@0 /top/dut_i/reg_arr@1 /top/dut_i/reg_arr@2}

10.6.2.6 HDL multi-bit slice

Class name	upfHdlMultiBitSliceT	
Class membership	upfHdlMultiBitSliceT, upfBaseNamedT, upfBaseT	
Handle ID	`<handle ID of upf_parent><upf_name of Object>` `Where upf_name of this object is constructed as` `@<upf_msb>:<upf_lsb>`	
Handle ID examples	`/top/dut_i/complex_rec@3:2`	
Property	**Return value**	**Description**
upf_name	upfStringT	Name of object
upf_parent	upfBaseNamedT	Parent of object
upf_msb	upfIntegerT	normalized msb info of slice
upf_lsb	upfIntegerT	normalized lsb info of slice
upf_slice_bits	List of upfHdlNetBitT	List of paHdlNetBitT or pdHdlPortBitT objects corresponding to each normalized width

The object of upfHdlMultiBitSliceT is created to represent a group of consequtive bits of a multi-bit object, e.g., field of a record, slice of array/multi-dimensional arrays.

The upf_msb will always be greater than upf_lsb property of this object. These properties normalized ranges.

Please refer to 10.4.3.3 for more details.

Examples

UPF source: test.upf

```
70 set_retention ret -domain PD -elements {complex_rec.f2}
```

Object definition

Handle ID	/top/dut_i/complex_rec@3:2
Properties	**Value**
upf_name	@3:2
upf_parent	/top/dut_i/complex_rec
upf_msb	3
upf_lsb	2
upf_slice_bits	{/top/dut_i/complex_rec@3 /top/dut_i/complex_rec@2}

10.6.3 Relationship objects

10.6.3.1 UPF extent

Class name	upfExtentT	
Class membership	upfExtentT, upfBaseRelationshipT, upfBaseT	
Handle ID	`#UPFEXTENT<tool generated counter>#`	
Handle ID examples	`#UPFEXTENT91#`	
Property	**Return value**	**Description**
upf_hdl_element	upfBaseHdlT	Handle of element in the effective element list
upf_cells	List of upfBaseHdlT	Cells inserted for element in effective element list
upf_object	upfExtentClassT	Handle of UPF object for which the extent was created
upf_next_extent	upfExtentT	Handle to the upfExtentT object that points to the next element in the effective_element_list

The object of upfExtentT class captures the information about the extent of power domains and strategies. More specifically, it contains the information about the following:

a) effective_element_list (see 5.9) for power domains and other strategies
b) -instance for strategies

The property **upf_hdl_element** points to the element that is part of the effective element list of the object. The element also stores the back reference to this upfExtentT object in the **upf_extents** property (see 10.6.2). The UPF object for which the extent is created is referred to as **upf_object** property. The **upf_cells** property captures the list of cells that are inserted for the specific extent by a given UPF object.

The upfExtentT object also captures information about the -instance relationship for a given object. For such cases, the port for which the power-management cell is marked as -instance is captured as **upf_hdl_element** and the actual cell instance is referred to as **upf_cells**.

The HDL objects present in **upf_cells** property can be of:

a) upfHdlScopeT type when the cells are explicit instantiations in HDL (-instance) or inserted by the application of UPF
b) upfHdlDeclT when the cells are inferred at RTL stage, e.g., registers/latches modeled as always blocks

The upfExtentT object captures the effective_element_list information through **upf_next_extent** property which points to the next object in the effective_element_list in any particular order. The object that represents the first element in the effective_element_list is called as extent_head and is stored as **upf_effective_extents** property in upfExtentClassT objects. The extent_head acts as a root from which the effective element list can be calculated by traversing the **upf_next_extent** property. The helper proc **query_effective_extent_list** (see C.1.3) can be used to get the flattened list of upfExtentT objects.

Examples

Example 1: -instance information

UPF source: test.upf

```
70 set_isolation iso -domain PD \
71    -instance {{mid/iso_inst mid/port_iso}} …
```

Object definition

Handle ID	#UPFEXTENT3#
Properties	**Value**
upf_hdl_element	/top/dut_i/mid/port_iso
upf_cells	{/top/dut_i/mid/iso_inst}
upf_object	/top/dut_i/PD.iso

UML object diagram (see Figure 14)

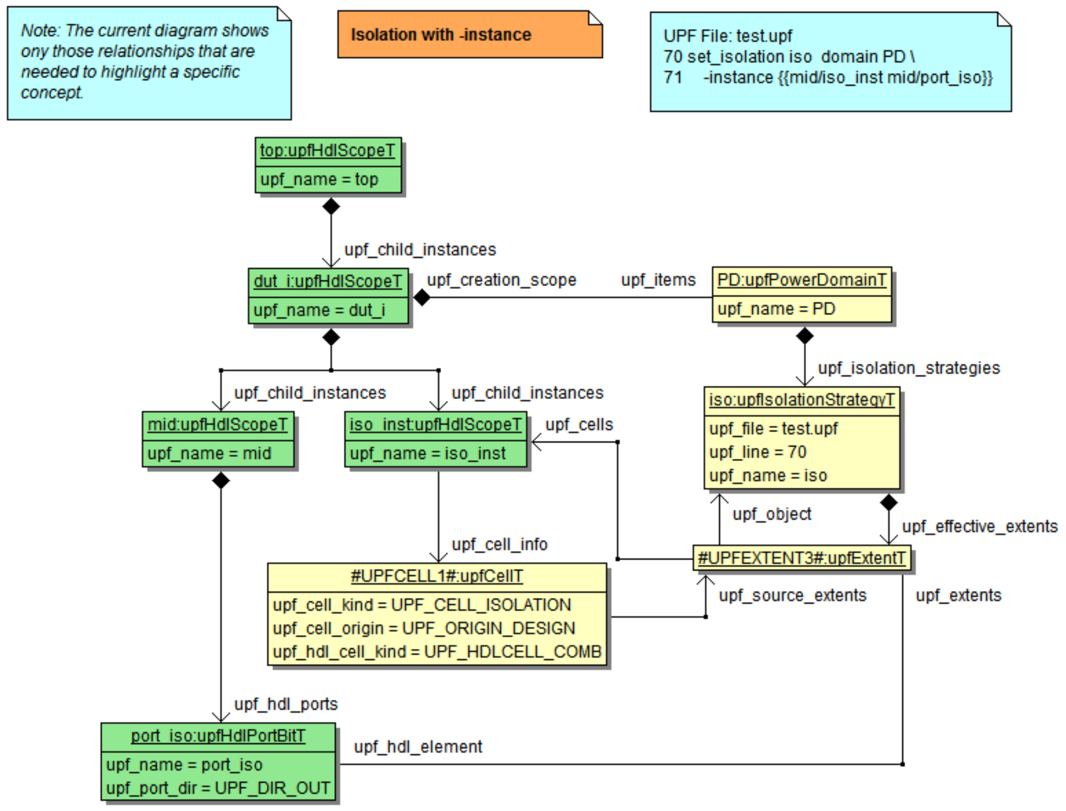


Figure 14—UML object diagram denoting—instance versuss upfExtentT relationship

Example 2: Effective element list of isolation strategy

UPF source: test.upf

```
1 set_scope /top/dut_i
2 create_power_domain PD -elements { m1 }
3 set_isolation iso -domain PD \
  -applies_to both ...
```

Object definitions

Handle ID	#UPFEXTENT1#
Properties	**Value**
upf_hdl_element	/top/dut_i/m1/p1
upf_cells	{/top/dut_i/p1_UPF_ISO}
upf_object	/top/dut_i/PD.iso
upf_next_extent	#UPFEXTENT2#

Handle ID	#UPFEXTENT2#
Properties	**Value**
upf_hdl_element	/top/dut_i/m1/p2[0]
upf_cells	{/top/dut_i/\p2[0]_UPF_ISO}
upf_object	/top/dut_i/PD.iso
upf_next_extent	#UPFEXTENT3#

Handle ID	#UPFEXTENT3#
Properties	**Value**
upf_hdl_element	/top/dut_i/m1/p2[1]
upf_cells	{/top/dut_i/\p2[1]_UPF_ISO}
upf_object	/top/dut_i/PD.iso

UML Object Diagram, see Figure 15

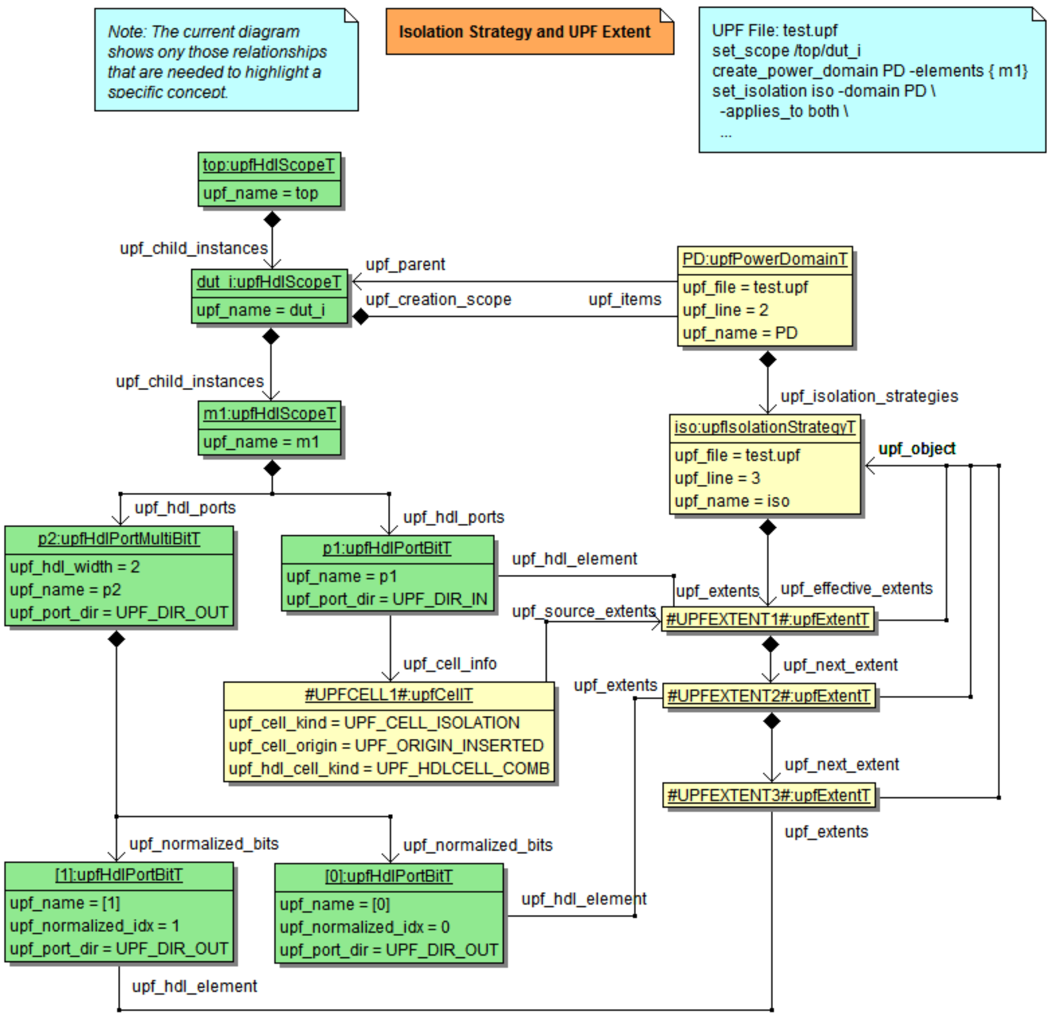


Figure 15—UML object diagram denoting effective_element_list of isolation strategy

Example 3: Inferred corruption logic inserted by power domain (implicit connections)

UPF source: test.upf

```
30 create_power_domain PD -elements {.}
```

HDL source: mid.v

```
module mid1;
...
assign comb = a && b;
always_ff @(posedge clk) q_ff <= d;
...
```

Object definition

Handle ID	#UPFEXTENT3#
Properties	**Value**
upf_hdl_element	/top/dut_i/m1
upf_cells	{/top/dut_i/comb /top/dut_i/m1/q_ff}
upf_object	/top/dut_i/PD

UML object diagram, see Figure 16

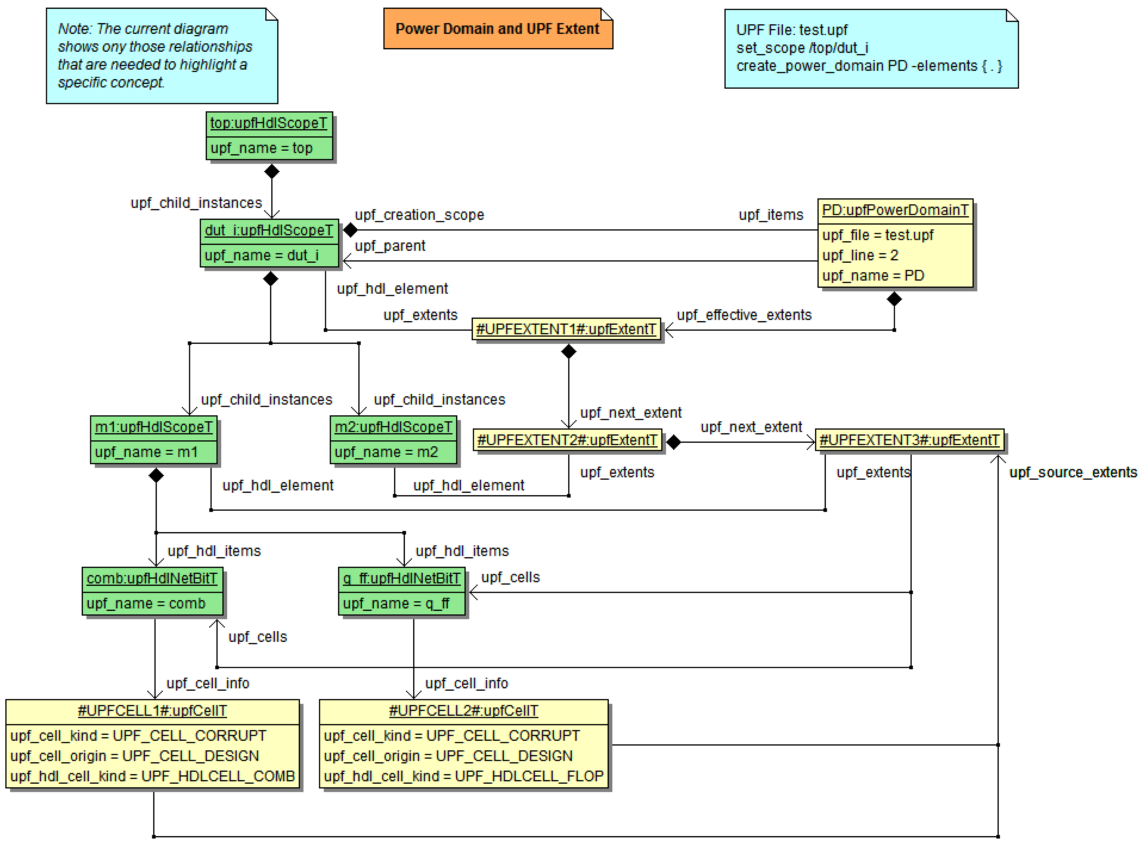


Figure 16—UML object diagram denoting power domain and its extent

Example 4: Extent of donut power domains

UPF source: test.upf

```
create_power_domain PD1 \
  -elements {top top/m2/b3}
create_power_domain PD2 \
  -elements {top/m2}
```

HDL source: dut.v

```
module top;
  mid1 m1();
  mid2 m2();
  mid3 m3();
endmodule
module mid1;
  bot b1();
  bot b2();
endmodule
module mid2;
  bot b3();
endmodule
module mid3;
endmodule
module bot;
endmodule
```

Object definitions

Handle ID	#UPFEXTENT1#
Properties	**Value**
upf_hdl_element	/top
upf_object	/top/ PD1
upf_next_extent	#UPFEXTENT2#

Handle ID	#UPFEXTENT3#
Properties	**Value**
upf_hdl_element	/top/m2/b3
upf_object	/top/ PD1
upf_next_extent	#UPFEXTENT4#

Handle ID	#UPFEXTENT2#
Properties	**Value**
upf_hdl_element	/top/m1
upf_object	/top/ PD1
upf_next_extent	#UPFEXTENT5#

Handle ID	#UPFEXTENT4#
Properties	**Value**
upf_hdl_element	/top/m3
upf_object	/top/ PD1

Handle ID	#UPFEXTENT5#
Properties	**Value**
upf_hdl_element	/top/m1/b1
upf_object	/top/ PD1
upf_next_extent	#UPFEXTENT6#

Handle ID	#UPFEXTENT6#
Properties	**Value**
upf_hdl_element	/top/m1/b2
upf_object	/top/ PD1
upf_next_extent	#UPFEXTENT3#

Handle ID	#UPFEXTENT7#
Properties	**Value**
upf_hdl_element	/top/m2
upf_object	/top/ PD2

UML object diagram, see Figure 17

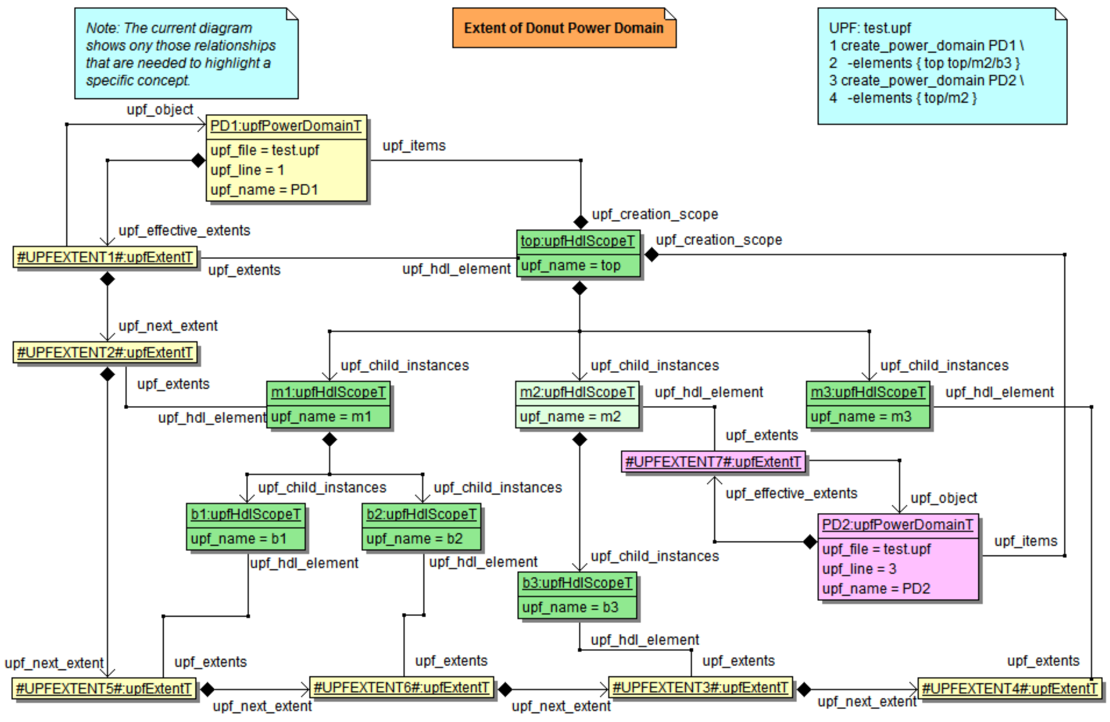


Figure 17 —UML object diagram showing extent information of donut style power domain

10.6.3.2 Control sensitivity

Class name	upfSignalSenseT	
Class membership	upfSignalSenseT, upfBaseRelationshipT, upfBaseT	
Handle ID	`#UPFSIGSENSE<tool generated counter>#`	
Handle ID examples	`#UPFSIGSENSE1#`	
Property	**Return value**	**Description**
upf_signal_sensitivity	upfSignalSenseKindE	Sensitivity of control signal (-*_sense option)
upf_control_signal	upfBaseNamedT	Handle to control signal

The object of type upfSignalSenseT is used to represent the relationship between control signals and their sensitivity.

Table 26 is the mapping of UPF commands and the properties of upfSignalSenseT object.

Table 26—Mapping of UPF commands and properties of upfSignalSenseT

UPF command	Property name
set_isolation -isolation_signal	upf_control_signal
set_isolation -isolation_sense	upf_signal_sensitivity
set_retention -save_signal {logic_net <high | low | posedge | negedge>}	upf_control_signal for logic_net
	upf_signal_sensitivity for sense

For possible values of upf_signal_sensitivity property see Table 27.

Table 27—Enumerated type upfSignalSenseKindE

upfSignalSenseKindE	
Enumerated literals	**UPF mapping**
UPF_SENSE_HIGH	high
UPF_SENSE_LOW	low
UPF_SENSE_POSEDGE	posedge
UPF_SENSE_NEGEDGE	negedge

Examples

UPF source: test.upf

```
set_scope /top
set_retention ret \
    -domain PD \
    -save_signal {ret_ctrl high}
```

Object definition

Handle ID	#UPFSIGSENSE1#
Properties	**Value**
upf_control_signal	/top/ret_ctrl
upf_signal_sensitivity	UPF_SENSE_HIGH

10.6.3.3 Cell information

Class name	upfCellT	
Class membership	upfCellT, upfBaseRelationshipT, upfBaseT	
Handle ID	`#UPFCELL<tool generated counter>#`	
Handle ID examples	`#UPFCELL1#`	
Property	**Return value**	**Description**
upf_model_name	upfStringT	The name of model corresponding to cell. This will be optional as it will not be present for cells that are inferred at RTL.
upf_cell_kind	upfCellKindE	The enumerated value representing kind of a cell inferred from UPF, e.g., retention, isolation, corruption, etc.
upf_hdl_cell_kind	upfHdlCellKindE	The enumerated value representing kind of cell determined from HDL, e.g., flop, latch, memory, etc.
upf_cell_origin	upfCellOriginE	The enumerated value indicating the source of insertion of this cell, whether inserted by UPF or already present in design
upf_source_extents	List of upfExtentT	The list of upfExtentT object which caused insertion of this cell

The object of type upfCellT represents the details of the cell information that is created or inferred by UPF. This object is created when UPF is applied on the HDL design. For possible values of upf_cell_kind property see Table 28.

Table 28—Enumerated type upfCellKindE

upfCellKindE	
Enumerated literals	**UPF mapping**
UPF_CELL_NONE	Not a cell
UPF_CELL_ISOLATION	Represents isolation cell
UPF_CELL_LEVEL_SHIFTER	Represents level-shifter cell
UPF_CELL_ISO_LS_COMBO	Represents isolation and level-shifter combo cell
UPF_CELL_RETENTION	Represents retention cell
UPF_CELL_SWITCH	Represents a switch cell
UPF_CELL_REPEATER	Represents a repeater or buffer cell
UPF_CELL_CORRUPT	Represents any standard cell which can get corrupted
UPF_CELL_MACRO	Represents a macro cell or power model

The cell information can be present on either scope or items (ports or nets). If cell information is present on an item (port or net) it represents an inferred logic which is not yet present in the design. This scenario is typically present at RTL state where a statement or expression represents some synthesizable logic. In all other cases, where there is explicit instantiation of cell, the cell information is present on the scope type object.

The upf_hdl_cell_kind property contains information about the kind of cell inferred from HDL. For possible values of upf_hdl_cell_kind property see Table 29.

Table 29—Enumerated type upfHdlCellKindE

upfHdlCellKindE	
Enumerated literals	**UPF mapping**
UPF_HDLCELL_NONE	not a cell
UPF_HDLCELL_COMB	represents a combinatorial logic
UPF_HDLCELL_FLOP	represents a flip-flop
UPF_HDLCELL_LATCH	represents a latch
UPF_HDLCELL_MEM	represents a memory like RAM, etc.

The upf_cell_origin property represents the source of a particular cell. For possible values of upf_cell_origin property see Table 30.

Table 30—Enumerated type upfHdlCellKindE

upfCellOriginE	
Enumerated literals	**UPF mapping**
UPF_ORIGIN_UNKNOWN	When cell origin is not known
UPF_ORIGIN_DESIGN	When cell is present in design itself
UPF_ORIGIN_INSERTED	When cell is inserted by UPF after application of strategy (e.g., isolation) and using default model
UPF_ORIGIN_INSERTED_MAP	When cell is inserted by UPF after application of strategy and using user specified model via map_* and use_interface_cell commands
UPF_ORIGIN_INFERRED	When cell is inferred by UPF at RTL. This information will only be present on cells which are set on HDL Port or Nets group of objects.

Examples

Example 1: Inferred isolation cell

UPF source: test.upf

```
set_scope /top
set_isolation iso -domain PD -elements {port1} -clamp_value 1 ...
```

Object definition

Handle ID	#UPFCELL1#
Properties	**Value**
upf_cell_kind	isolation_cell
upf_hdl_cell_kind	UPF_HDLCELL_COMB
upf_cell_origin	UPF_ORIGIN_INSERTED

Example 2: -instance of isolation

UPF source: test.upf

```
set_scope /top
set_isolation iso \
  -domain PD \
  -instance {port1 iso_inst} \
  -clamp_value 1
```

Object definition

Handle ID	#UPFCELL2#
Properties	**Value**
upf_model_name	iso_model
upf_cell_kind	UPF_CELL_ISOLATION
upf_hdl_cell_kind	UPF_HDLCELL_COMB
upf_cell_origin	UPF_ORIGIN_DESIGN

10.6.3.4 Expression

Class name	upfExpressionT	
Class membership	upfExpressionT, upfBaseRelationshipT, upfBaseT	
Handle ID	`#UPFEXPR<tool generated counter>#`	
Handle ID examples	`#UPFEXPR1#`	
Property	**Return value**	**Description**
upf_expr_string	upfStringT	String representation of expression
upf_expr_operands	List of upfBaseNamedT	List of operands used in the expression
Dynamic property (only available during simulation)		
upf_current_value	upfBooleanT	The current value of the object during simulation

The object of upfExpressionT class is a relationship object that captures the boolean expression information defined in UPF.

Examples

UPF source: test.upf

```
add_power_state PD \
   -state {ON -logic_expr {primary == ON}}
```

Object definition

Handle ID	#UPFEXPR1#
Properties	**Value**
upf_expr_string	primary == ON
upf_expr_operands	/top/dut_i/PD.primary

10.6.3.5 UPF attributes

Class name	upfAttributeT	
Class membership	upfAttributeT, upfBaseRelationshipT, upfBaseT	
Handle ID	`#UPFATTR<tool generated counter>#`	
Handle ID examples	`#UPFATTR1#`	
Property	**Return value**	**Description**
upf_file	upfStringT	The file name of the source where the attribute was defined
upf_line	upfIntegerT	The line number of the source where the attribute is defined
upf_attr_name	upfStringT	The name of the attribute set on object
upf_attr_value	upfStringT	The value of the attribute set on object

The object of upfAttributeT class is a relationship object that captures the information about various predefined and user defined attributes added on the objects in UPF. This is typically the information specified by **set_design_attributes** or **set_port_attributes** command. It can also be the predefined UPF attributes specified in HDL or Liberty specifications.

Examples

UPF source: test.upf

```
30 set_port_attributes -ports {my_Logic_Port} -clamp_value 1
```

Object definition

Handle ID	#UPFATTR1#
Properties	**Value**
upf_file	test.upf
upf_line	30
upf_attr_name	UPF_clamp_value
upf_attr_value	1

11. Information model application programmable interface (API)

11.1 Tcl interface

11.1.1 Overview

Subclause 11.1 defines the Tcl Interface for the information model. The commands defined in 11.1 are only available during phase 5 (see 8.3.6) of the UPF-processing phases. In this phase, only the following UPF commands will be available:

— set_scope
— bind_checker
— find_objects
— load_upf

All commands available in this phase have an immediate effect. The first occurance of any command defined in 11.1 will indicate the start of phase 5.

It shall be an error if any other UPF command is called during phase 5.

11.1.2 Basic UPF query commands

11.1.2.1 upf_query_object_properties

Purpose	Query properties on a given object	

Syntax	`upf_query_object_properties object_handle`
	`[-property property_keyword ]`	

Arguments	`object_handle`	UPF handle of the given object
	`-property property_keyword`	Return the value of specified property. By default all properties are returned as key value pairs.
Return value	Returns a string containing value of the specified property or empty string if not found	

The upf_query_object_properties command returns the result of querying the specified property on the object.

— If the return type of the property is a list, then a Tcl list containing values is returned.
— If -property is not specified, then all the property values are returned in the form of list. If the returned value is already a list then its represented as list of list

It shall be an error if:

— object_handle is not a valid UPF handle
— property_keyword is not a valid property ID

— object_handle is not present in the information model
— property_keyword is a valid property ID but not possible on the given object type

Syntax examples

Example 1: Get simple-name of object

```
upf_query_object_properties /top/dut_i/pd.iso_strategy \
    -property upf_name
```

Output

```
iso_strategy
```

Example 2: Get all properties on power-domain "pd"

```
upf_query_object_properties /top/dut_i/pd
```

Output

```
{ \
    {upf_name pd} {upf_file test.upf} {upf_line 11} \
    {upf_parent /top/dut_i} \
    {upf_creation_scope /top/dut_i} \
    {upf_effective_extents #UPFEXTENT1#} \
    {upf_supply_set_handles {\
        /top/dut_i/pd.primary \
    } \
    {upf_isolation_strategies {/top/dut_i/pd.iso_strategy}}
}
```

11.1.2.2 upf_query_object_type

Purpose	Query type of given object	
Syntax	`upf_query_object_type object_handle`	
Arguments	`object_handle`	UPF handle of the given object
Return value	Returns the keyword representing object class type or empty string if object is not present in information model	

The upf_query_object_type command returns the type of specified object.

It shall be an error if:

— object_handle is not a valid UPF handle
— object_handle is not present in the information model

Syntax examples

Example 1: Get type of UPF object

```
upf_query_object_type /top/dut_i/pd
```

Output

```
upfPowerDomainT
```

Example 2: Get type of HDL object

```
upf_query_object_type /top/dut_i
```

Output

```
upfHdlScopeT
```

Example 3: Get type of relationship object

```
upf_query_object_type #UPFEXTENT1#
```

Output

```
upfExtentT
```

11.1.2.3 upf_object_in_class

Purpose	Check if object belongs to particular class	
Syntax	`upf_object_in_class object_handle -class <CLASS_ID>`	
Arguments	`object_handle`	UPF handle of the given object
	`-class <CLASS_ID>`	Valid ids from class membership of an object
Return value	Returns `1` if object_handle belongs to CLASS_ID and `0` if it does not	

The upf_object_in_class command is used to check if the given handle belongs to a specified class. This is useful to create more robust scripts that have error checking built into them.

It shall be an error if:

— object_handle is not a valid UPF handle
— object_handle is not present in the information model
— CLASS_ID is not a valid class name

Syntax examples

Example 1: Check if object belongs to strategy

```
upf_object_in_class /top/dut_i/pd.iso_strategy \
    -class upfStrategyT
```

Output

```
1
```

Example 2: Object doesn't belong to class

```
upf_object_in_class /top/dut_i/pd \
    -class upfStrategyT
```

Output

```
0
```

11.1.2.4 upf_query_object_pathname

Purpose	Get hierarchical pathname for a given handle	
Syntax	`upf_query_object_pathname object_handle` `[-relative_to <object handle>]`	
Arguments	`object_handle`	UPF handle of the given object
	`-relative_to <object_handle>`	Handle to ancestor object of group UPF or HDL. By default the value is null. In that case, the absolute hierarchical pathname is returned.
Return value	Returns the string representing the RTL pathname of given handle or an empty string if error	

The upf_query_object_pathname is a helper query command that is used to return the hierarchical pathname relative to given scope. The valid handle types for this command are handles that belong to the following UPF objects and HDL objects group. If a relationship object is passed to this command then an empty string will be returned.

— If -relative_to option is specified with a valid handle which is an ancestor of the given handle, then a relative pathname is returned.

— If the handle specified in -relative_to option is not an ancestor to given object handle then an empty string is returned.

— If no -relative_to option is specified then the full hierarchical pathname is returned.

It shall be an error if:

— object_handle is not a valid UPF handle

— object_handle is not present in the information model

NOTE—This query command can be useful in cases where the handle ID is some modification of hierarchical path. This happens in case of multi-bit type HDL objects where the handle ID contains the normalization information, e.g., /top/dut_i/mid/net@63:32.

Syntax examples

Example 1: Get relative pathname

```
upf_query_object_pathname /top/dut_i/pd.iso_strategy \
    -relative_to /top
```

Output

```
dut_i/pd.iso_strategy
```

Example 2: Get RTL name-mapped of multi-bit slice

```
upf_query_object_pathname /top/dut_i/mid/net@63:32
```

Output

```
/top/dut_i/mid/net.f1(2)
```

Example 3: Get list of all nets in the scope of a given power domain

```
set cs [upf_query_object_properties /top/dut_i/pd \
        -property upf_creation_scope ]

set scope [upf_query_object_pathname $cs]
find_objects $scope -pattern * -object_type net
```

Output

```
/top/dut_i/net1
/top/dut_i/net2
```

11.1.3 Complex UPF query command

11.1.3.1 Introduction

The *complex UPF query command* is a high-level query command that can be built using basic UPF query commands. It performs some advanced queries like searching for names using patterns.

11.1.3.2 query_upf

Purpose	Find objects (including UPF created or inferred objects) in the logic hierarchy

Syntax	**query_upf** <*domain_name* | *scope*> **-pattern** *search_pattern* [**-object_type** <**inst** | **port** | **supply_port** | **net** | **supply_net** | **supply_set**>] [**-inst_type** <**level_shifter** | **isolation_cell** | **switch_cell** | **retention_cell** | **all**>] | [**-direction** <**in** | **out** | **inout**>] [**-transitive** [<**TRUE** | **FALSE**>]] [**-regexp** | **-exact**] [**-ignore_case**] [**-non_leaf** | **-leaf_only**]	
Arguments	*domain_name* | *scope*	Either a power domain or a scope can be specified. If a power domain is specified, the search is restricted to that power domain; otherwise the search is restricted to the specified scope.
	-pattern *search_pattern*	The string used for searching. By default, *search_pattern* is treated as a Tcl `glob` expression.
	-object_type <**inst** | **port** | **supply_port** | **net** | **supply_net** | **supply_set**>	Limits the objects returned. By default, all objects are returned.
	-inst_type <**level_shifter** | **isolation_cell** | **switch_cell** | **retention_cell** | **all**>	If **-object** is **inst**, this option limits the type of instances returned to be level-shifter, isolation, switch, or retention cells. The default is **all**, which returns all instances.
	-direction <**in** | **out** | **inout**>	If **-object** is **port**, then **-direction** can be used to restrict the directions of the returned ports.
	-transitive [<**TRUE** | **FALSE**>]	If **-transitive** is not specified at all, the default is **-transitive FALSE**. If **-transitive** is specified without a value, the default value is **TRUE**.
	-regexp | **-exact**	**-regexp** enables support for regular expression in the specified *search_pattern*. **-exact** disallows wildcard expansion on the specified *search_pattern*. If neither **-regexp** or **-exact** are specified, then *search_pattern* is interpreted as a Tcl `glob` expression.
	-ignore_case	Performs case-insensitive searches. By default, all matches are case sensitive.
	-non_leaf | **-leaf_only**	If **-non_leaf** is specified, only non-leaf instances are returned; if **-leaf_only** is specified, only leaf-level instances are returned. By default, both leaf and non-leaf instances are returned.
Return value	Returns a list of names (relative to the current scope) of objects that match the search criteria; when nothing is found that matches the search criteria, a *null string* is returned. The list contains just the object names, without any indication of object type. The list may contain names of more than one type of object.	

The **query_upf** command searches for instances, nets, supply nets, ports, and supply ports in and below the *scope* or within the extent of a *domain_name*. This command works on the logic hierarchy and can be executed post-UPF annotation.

The **query_upf** command works on the logic hierarchy from a domain-centric or hierarchy-centric approach. A *domain-centric approach* restricts the search to instances, net, or ports that are logically within the extent of the specified *domain_name*. A *hierarchy-centric approach* searches in the *scope* only, or in and below the *scope* when **-transitive** is specified.

A domain-centric search examines all logical levels that are members of the specified domain. Based on Figure 18 and Figure 19, the command `query_upf {PD1} -pattern *` looks for any object (port, net, or instance) matching the specified string in the logical hierarchies `A`, `A/B`, `A/C`, or `A/B/D/F`.

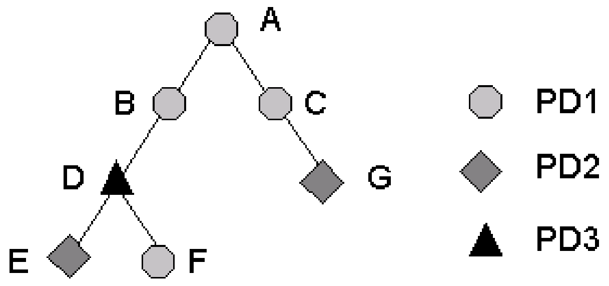


Figure 18—Logic hierarchy

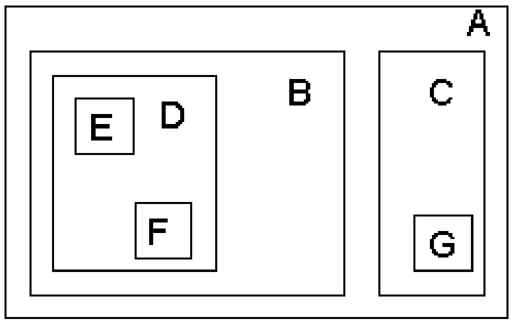


Figure 19—Physical layout

If searching for inputs into `PD3`, the command

```
query_upf {PD3} -pattern * -object_type port -direction in
```

returns any inputs from {`B->D`, `F->D`, and `E->D`}.

-inst_type only returns instances of a particular type. For example, to find all level-shifters in the domain `PD3`, the following **query_upf** command could be used:

```
query_upf {PD3} -pattern * -inst_type level_shifter -object inst
```

The following conditions also apply:

— **-transitive** is ignored in a domain-centric search.

— The specified *domain_name* or *scope* cannot start with `..` or `/`, i.e., **query_upf** shall be referenced from the current scope, and reside in the current scope or below it.

— All elements returned are referenced to the current scope.

— If *domain_name* or *scope* is specified as **.** (a dot), the current scope is used as the root of the search.

— **query_upf** takes a *scope* argument. The specified scope may reference a generate block as the root of the search.

— For details on pattern matching and wildcarding, see 6.30.2 and Table 5.

Syntax examples

```
query_upf A/B/D \
-pattern *BW1* \
-object inst \
-transitive
```

11.1.3.2.1 Tcl code using basic queries

This subclause describes the functionality of query_upf command using basic Tcl queries.

NOTE—The Tcl code shown here is only provided for illustration purposes in order to explain the semantics and behavior of Tcl queries. The tools are free to provide native implementation of this command.

Tcl code

```
#----------------------------------------------------------------
# Helper proc to check for kind of pattern matching
#----------------------------------------------------------------
proc is_matching_pattern {
  name
  pattern
  search_type
} {
  if { ($name != "")
    && (
        ($search_type == "glob"  && [string match $pattern $name])
    ||  ($search_type == "regex" && [regexp $pattern $name])
    ||  ($search_type == "exact" && $pattern eq $name)
       )
  } {
    return 1;
  }
  return 0;
}

#----------------------------------------------------------------
# Helper proc to search pattern on the object by property and
# filter the results based on class type
#----------------------------------------------------------------
proc query_objects_by_property {
  scope
  property
  pattern
  class
  search_type
} {
  set result {};
  set children [upf_query_object_properties $scope \
                -property $property];
  foreach child $children {
    if {[upf_object_in_class $child -class $class]} {
      set name [upf_query_object_properties $child \
                -property upf_name];
      if {[is_matching_pattern $name $pattern $search_type]} {
        lappend result [upf_query_object_pathname $child \
                        -relative_to .];
      }
    }
  }
  return $result;
}

#--------------------------------------------------------------
# Helper proc to filter the scopes based on their cell type
#--------------------------------------------------------------
proc query_inst_by_type {
  inst
  type
} {
  set status 0
  set cell_info [upf_query_object_properties $inst \
                 -property upf_cell_info]
  set cell_kind [upf_query_object_properties $cell_info \
                 -property upf_cell_kind]
  if {$type == "all" || $type == "isolation_cell"} {
    if {$cell_kind eq "UPF_CELL_ISOLATION"} {
      set status 1
    }
  }
  if {$type == "all" || $type == "level_shifter"} {
    if {$cell_kind eq "UPF_CELL_LEVEL_SHIFTER"} {
      set status 1
    }
  }
  if {$type == "all" || $type == "switch_cell"} {
    if {$cell_kind eq "UPF_CELL_SWITCH"} {
      set status 1
    }
  }
  if {$type == "all" || $type == "retention_cell"} {
    if {$cell_kind eq "UPF_CELL_RETENTION"} {
      set status 1
    }
  }
  return $status;
}

#--------------------------------------------------------------
# UPF query_upf command based on UPF Information Model
# this proc searches with a given scope
#--------------------------------------------------------------
proc query_upf_scope {
  scope
  pattern
  object_type
  inst_type
  direction
  transitive
  search_type
  leaf_only
} {
  set result {}
# Error check to ensure $scope is an HDL Scope
  if {[upf_query_object_type $scope] != "upfHdlScopeT"} {
      return $result;
  }

# Search HDL Objects
```

```
  if {$object_type == ""
   || $object_type == "inst"
   || $object_type == "port"
   || $object_type == "net"
  } {
  set result [concat $result [find_objects $scope -pattern $pattern \
                            -object_type $object_type $leaf_only
                            -transitive $transitive $direction $search_type]
  }
# Search UPF Objects
  if {$object_type == "" || $object_type == "supply_port"} {
    set result [concat $result [query_objects_by_property $scope \
                                upf_items $pattern upfSupplyPortT \
                                $search_type]]
  }
  if {$object_type == "" || $object_type == "supply_net"} {
    set result [concat $result [query_objects_by_property $scope \
                                upf_items $pattern upfSupplyNetT \
                                $search_type]]
  }
  if {$object_type == "" || $object_type == "supply_set"} {
    set result [concat $result [query_objects_by_property $scope \
                                upf_items $pattern upfSupplySetT \
                                $search_type]]
  }
# Transitive behavior for UPF Objects only
  if {$object_type   == ""
   || $inst_type  == "level_shifter"
   || $inst_type  == "isolation_cell"
   || $inst_type  == "switch_cell"
   || $inst_type  == "retention_cell"
   || $inst_type  == "all"
  } {
    set child_scopes [query_objects_by_property $scope \
                      upf_child_instances $pattern \
                      upfHdlScopeT $search_type]
# Check if scope belongs to some Power management cell
    foreach child $child_scopes {
# Filter matched results based on type
      if {[query_inst_by_type $child $inst_type]} {
        set result [concat $result $child]
      }
    }
  }
  if {$transitive == "true"} {
# Recursively call query_upf on child_scopes
    set child_scopes [upf_query_object_properties $scope \
                      -property upf_child_instances];
    foreach child $child_scopes {
      set result [concat $result [query_upf $child $pattern \
                                  $object_type $inst_type \
                                  $direction $transitive \
                                  $search_type $leaf_only]]
    }
  }
  return $result;
}

#---------------------------------------------------------------
# UPF query_upf command based on UPF Information Model
# this proc searches with a given Power Domain
#---------------------------------------------------------------
proc query_upf_domain {
  pd
  pattern
  object_type
  inst_type
  direction
  search_type
  leaf_only
} {
  set result {}
  if {[upf_query_object_type $pd] != "upfPowerDomainT" } {
    return $result;
  }
  set extent_head [upf_query_object_properties $pd \
              -property upf_effective_extents];
  # Utility proc to return a list of effective extents
  # See C.1.3 for more details.
  set extents [query_effective_extent_list $extent_head]
  foreach extent $extents {
    set scope [upf_query_object_properties $extent \
              -property upf_hdl_element];
    set result [concat $result [query_upf_scope $scope $pattern \
                               $object_type $inst_type $direction \
                               "false" $search_type $leaf_only]];
  }
  return $result;
}

#--------------------------------------------------------------
# Top level query_upf
#--------------------------------------------------------------
proc query_upf {
  scope
  pattern
  object_type
  inst_type
  direction
  transitive
  search_type
  leaf_only
} {
  if {[upf_query_object_type $scope] == "upfPowerDomainT" } {
    return [query_upf_domain $scope $pattern $object_type \
            $inst_type $direction $search_type $leaf_only];
  } else {
    return [query_upf_scope $scope $pattern $object_type $inst_type \
            $direction $transitive $search_type $leaf_only];
  }
}
```

11.2 HDL interface

11.2.1 Introduction

The HDL interface to the information model allows user to create HDL descriptions which access the information model objects directly in HDL. This interface can be used to create the following:

- Abstract testbenches to manipulate UPF objects directly in simulation
- Checker/coverage models

— Simulation models that directly manipulate UPF objects during simulation

11.2.2 Representation of property types in HDL

11.2.2.1 Introduction

The objects and properties in the information model can be accessed by a HDL object of upfHandleT type.

The detailed mapping of property types in information model and HDL is shown in Table 31.

Table 31 —Property HDL type mapping

Property type	Type name	SV	VHDL
String	upfStringT	string	string
Integer	upfIntegerT	int	integer
Boolean	upfBooleanT	bit	bit
Float	upfRealT	real	real
Enumerated	Type names with suffix E	enum types	enum types
Handle to objects/properties	upfHandleT	chandle	integer
List of handle to other objects	upfHandleT	chandle	integer

11.2.2.2 upfSupplyTypeT

Class name	upfSupplyTypeT	
Class membership	upfSupplyTypeT, upfBaseT	
Handle ID	Any tool assigned ID. Only valid during simulation.	
Property	**Return value**	**Description**
upf_state	upfSupplyStateE	The current state of the upfSupplyTypeT object
upf_voltage	upfIntegerT	The current voltage of the upfSupplyTypeT object expressed in integer value in micro-volts

The upfSupplyTypeT class is a special class that represents the current value of a supply type object (supply net/port). The class maintains two pieces of information:

a) Current state of supply object, in the form of enumerated type upfSupplyStateE

b) Current voltage of supply object represented as integer value in micro-volts

The upfSupplyTypeT class ensures the supply net state and voltage values can be easily propagated and modeled in various HDLs. The HDL representation of this class is shown in Table 32.

Table 32—HDL representation of upfSupplyTypeT

Type name	SV	VHDL

upfSupplyTypeT	struct {
	  upfSupplyStateE state;
	  upfIntegerT voltage;
	} upfSupplyTypeT
		type upfSupplyTypeT is record
		  state : upfSupplyStateE;
		  voltage: upfIntegerT;
		end record;

NOTE—In order to ensure backward compatibility with earlier versions of UPF, the UPF packages also define supply_net_type datatype which is exactly similar to upfSupplyTypeT.

11.2.2.3 Native HDL representation

The objects that also have dynamic properties will also have a native HDL representation defined in the HDL package. The native HDL representation is a structure/record type in HDL that contains two fields:

a) A value field corresponding to the dynamic property of the object to allow continuous monitoring of dynamic properties

b) A handle field of type upfHandleT to allow access to other properties of the object

The native HDL representation is achieved by the following structure/record types in HDL, see Table 33.

Table 33—HDL types for native HDL representation

Type name	SV	VHDL

upfPdSsObjT	struct {
	  upfHandleT handle;
	  upfPowerStateObjT current_state;
	} upfPdSsObjT
		type upfPdSsObjT is record
		  handle : upfHandleT;
		  current_state: upfPowerStateObjT;
		end record;

upfPowerStateObjT	struct {
	  upfHandleT handle;
	  upfBooleanT is_active;
	} upfPowerStateObjT
		type upfPowerStateObjT is record
		  handle: upfHandleT;
		  is_active: upfBooleanT;
		end record;

upfBooleanObjT	struct {
	  upfHandleT handle;
	  upfBooleanT current_value;
	} upfBooleanObjT
		type upfBooleanObjT is record
		  handle: upfHandleT;
		  current_value: upfBooleanT;
		end record;

upfSupplyObjT	struct {
	  upfHandleT handle;
	  upfSupplyTypeT current_value;
	} upfSupplyObjT
		type upfSupplyObjT is record
		  handle: upfHandleT;
		  current_value: upfSupplyTypeT;
		end record;

The mapping of native HDL representation types and information model objects is shown in Table 34.

Table 34—Information model objects with native HDL representation

Information model types	Dynamic properties	HDL type mapping
upfPowerDomainT	current_state	upfPdSsObjT
upfSupplySetT	current_state	upfPdSsObjT
upfCompositeDomainT	current_state	upfPdSsObjT
upfPstStateT	is_active	upfPowerStateObjT
upfPowerStateT	is_active	upfPowerStateObjT
upfAckPortT	current_value	upfBooleanObjT
upfExpressionT	current_value	upfBooleanObjT
upfLogicNetT	current_value	upfBooleanObjT
upfLogicPortT	current_value	upfBooleanObjT
upfSupplyNetT	current_value	upfSupplyObjT
upfSupplyPortT	current_value	upfSupplyObjT

NOTE—There are two types, upfSupplyObjT and upfSupplyTypeT defined in the UPF package that represent supply nets/ports in UPF. An object of type upfSupplyObjT is a native HDL representation corresponding to a supply net/port created in UPF. Whereas, if an object declared in the HDL scope is of type upfSupplyTypeT, then it represents the supply net/port created directly in HDL. The object of upfSupplyTypeT only contains voltage and supply information. Whereas, the object of upfSupplyObjT can contain additional properties as defined for upfSupplyNetT or upfSupplyPortT type, when the mirroring relationship is established.

11.2.2.4 Enumerated types

The enumerated types in the information model are represented as corresponding enum types in HDL. The typenames will be the same as the typename of enumerated types and also there will be direct mapping between the enumerated literal in information model and HDL definition.

The access functions will return the position values of the enumerated literals and hence they need to be converted to corresponding literal before they are used. See 11.2.3.2.3 upf_get_value_int for more details.

11.2.2.5 Class ID

The classes in the information model will be identified in HDL by the enumerated values defined in upfClassIdE enumerated type.

Table 35 provides the mapping between information model classes and corresponding class id in HDL type.

Table 35—Mapping between class name and class ID in HDL

upfClassIdE			
S. no.	**Object/base class name**	**Class name**	**Class keyword**
1	Root class	upfBaseT	UPF_BASE
2	Base class for named objects	upfBaseNamedT	UPF_BASE_NAMED
3	Base class for relationship objects	upfBaseRelationshipT	UPF_BASE_RELATIONSHIP
4	Base class for hdl objects	upfBaseHdlT	UPF_BASE_HDL
5	Base class for upf objects	upfBaseUpfT	UPF_BASE_UPF
6	Base class for objects having extents	upfExtentClassT	UPF_EXTENT_CLASS
7	Base class for hdl declarations	upfHdlDeclT	UPF_HDL_DECL
8	Base class for network upf objects	upfNetworkClassT	UPF_NETWORK_CLASS
9	Base class for state objects	upfStateClassT	UPF_STATE_CLASS
10	Base class for hdl net objects	upfHdlNetClassT	UPF_HDL_NET_CLASS
11	Base class for hdl port objects	upfHdlPortClassT	UPF_HDL_PORT_CLASS
12	Base class for upf net objects	upfNetClassT	UPF_NET_CLASS
13	Base class for upf port objects	upfPortClassT	UPF_PORT_CLASS
14	Base class for upf strategies	upfStrategyT	UPF_STRATEGY
15	Base class for upf boundary strategies	upfBoundaryStrategyT	UPF_BOUNDARY_STRATEGY
16	Ack port	upfAckPortT	UPF_ACK_PORT
17	Isolation strategy	upfIsolationStrategyT	UPF_ISOLATION_STRATEGY
18	Level-shifter strategy	upfLevelShifterStrategyT	UPF_LEVEL_SHIFTER_STRATEGY
19	Logic net	upfLogicNetT	UPF_LOGIC_NET
20	Logic port	upfLogicPortT	UPF_LOGIC_PORT
21	Object ref handles	upfNamedRefT	UPF_NAMED_REF
22	Power domain	upfPowerDomainT	UPF_POWER_DOMAIN
23	Power state	upfPowerStateT	UPF_POWER_STATE
24	PST	upfPowerStateTableT	UPF_POWER_STATE_TABLE
25	State transition	upfPowerStateTransitionT	UPF_POWER_STATE_TRANSITION
26	Switch state	upfPowerSwitchStateT	UPF_POWER_SWITCH_STATE
27	Power switch	upfPowerSwitchT	UPF_POWER_SWITCH
28	PST state	upfPstStateT	UPF_PST_STATE
29	Repeater strategy	upfRepeaterStrategyT	UPF_REPEATER_STRATEGY
30	Retention strategy	upfRetentionStrategyT	UPF_RETENTION_STRATEGY
31	Supply net	upfSupplyNetT	UPF_SUPPLY_NET
32	Supply net	upfSupplyPortStateT	UPF_SUPPLY_PORT_STATE
33	Supply port	upfSupplyPortT	UPF_SUPPLY_PORT
34	Supply sets	upfSupplySetT	UPF_SUPPLY_SET
35	HDL multi-bit slice	upfHdlMultiBitSliceT	UPF_HDL_MULTI_BIT_SLICE
36	HDL scalar net	upfHdlNetBitT	UPF_HDL_NET_BIT
37	HDL multi-bit net	upfHdlNetMultiBitT	UPF_HDL_NET_MULTI_BIT
38	HDL scalar port	upfHdlPortBitT	UPF_HDL_PORT_BIT
39	HDL multi-bit port	upfHdlPortMultiBitT	UPF_HDL_PORT_MULTI_BIT
40	HDL scope	upfHdlScopeT	UPF_HDL_SCOPE
41	UPF attributes	upfAttributeT	UPF_ATTRIBUTE
42	Cell information	upfCellT	UPF_CELL
43	Expressions	upfExpressionT	UPF_EXPRESSION
44	Extent object	upfExtentT	UPF_EXTENT
45	Sensitivity of controls	upfSignalSenseT	UPF_SIGNAL_SENSE
46	HDL object representing supply net value	upfSupplyTypeT	UPF_SUPPLY_TYPE
47	Composite domain	upfCompositeDomainT	UPF_COMPOSITE_DOMAIN
48	Basic Boolean property type	upfBooleanT	UPF_BOOLEAN
49	Basic string property type	upfStringT	UPF_STRING
50	Basic integer property type	upfIntegerT	UPF_INTEGER
51	Basic real property type	upfRealT	UPF_REAL

11.2.2.6 Property ID

Table 36 provides the mapping between the property names and property IDs in HDL type.

Table 36—Mapping between property name and property ID in HDL

upfPropertyIdE				
S. no.	**Property name**	**Base class hierarchy**	**Return type**	**Property ID**
1	upf_parent	upfBaseNamedT	upfBaseNamedT	UPF_PARENT
2	upf_name	upfBaseNamedT	upfStringT	UPF_NAME
3	upf_hdl_attributes	upfBaseHdlT	List of upfAttributeT	UPF_HDL_ ATTRIBUTES
4	upf_extents	upfBaseHdlT	List of upfExtentT	UPF_EXTENTS
5	upf_cell_info	upfBaseHdlT	upfCellT	UPF_CELL_INFO
6	upf_creation_scope	upfBaseUpfT	upfBaseHdlT	UPF_CREATION _SCOPE
7	upf_line	upfBaseUpfT	upfIntegerT	UPF_LINE
8	upf_file	upfBaseUpfT	upfStringT	UPF_FILE
9	upf_effective _extents	upfExtentClassT	upfExtentT	UPF_EFFECTIVE _EXTENTS
10	upf_supply_ set_handles	upfExtentClassT	List of upfSupplySetT	UPF_SUPPLY_ SET_HANDLES
11	upf_lower_boundary	upfPowerDomainT	List of upfBaseHdlT	UPF_LOWER_ BOUNDARY
12	upf_isolation_ strategies	upfPowerDomainT	List of upfIsolationStrategyT	UPF_ISOLATION _STRATEGIES
13	upf_level_shifter_ strategies	upfPowerDomainT	List of upfLevelShifterStrategyT	UPF_LEVEL_SHIFTER _STRATEGIES
14	upf_pd_states	upfPowerDomainT	List of upfPowerStateT	UPF_PD_STATES
15	upf_pd_state_ transitions	upfPowerDomainT	List of upfPowerStateTransitionT	UPF_PD_STATE_ TRANSITIONS
16	upf_subdomains	upfCompositeDomainT	List of upfBaseUpfT	UPF_SUBDOMAINS
17	upf_repeater_ strategies	upfPowerDomainT	List of upfRepeaterStrategyT	UPF_REPEATER_ STRATEGIES
18	upf_retention_ strategies	upfPowerDomainT	List of upfRetentionStrategyT	UPF_RETENTION_ STRATEGIES
19	upf_current_state	upfPowerDomainT	upfPowerStateT	UPF_CURRENT_STATE
20	upf_functions	upfSupplySetT	List of upfNamedRefT	UPF_FUNCTIONS
21	upf_ss_states	upfSupplySetT	List of upfPowerStateT	UPF_SS_STATES
22	upf_ss_transitions	upfSupplySetT	List of upfPowerStateTransitionT	UPF_SS_ TRANSITIONS
23	upf_equivalent_sets	upfSupplySetT	List of upfSupplySetT	UPF_EQUIVALENT _SETS
24	upf_logic_refs	upfStrategyT	List of upfNamedRefT	UPF_LOGIC_REFS
25	upf_is_no_retention	upfRetentionStrategyT	upfBooleanT	UPF_IS_NO_RETENTION
26	upf_is_use_retention_ as_primary	upfRetentionStrategyT	upfBooleanT	UPF_IS_USE_RETENTION _AS_PRIMARY

Table 36—Mapping between property name and property ID in HDL *(continued)*

27	upf_restore_ condition	upfRetentionStrategyT	upfExpressionT	UPF_RESTORE_ CONDITION
28	upf_retention_ condition	upfRetentionStrategyT	upfExpressionT	UPF_RETENTION_ CONDITION
29	upf_save_condition	upfRetentionStrategyT	upfExpressionT	UPF_SAVE_CONDITION
30	upf_retention_ parameters	upfRetentionStrategyT	upfRetentionParamE	UPF_RETENTION_ PARAMETERS
31	upf_restore_signal	upfRetentionStrategyT	upfSignalSenseT	UPF_RESTORE_SIGNAL
32	upf_save_signal	upfRetentionStrategyT	upfSignalSenseT	UPF_SAVE_SIGNAL
33	upf_sink_filter	upfBoundaryStrategyT	upfAbstractObjT	UPF_SINK_FILTER
34	upf_source_filter	upfBoundaryStrategyT	upfAbstractObjT	UPF_SOURCE_FILTER
35	upf_is_use_ equivalence	upfBoundaryStrategyT	upfBooleanT	UPF_IS_USE_ EQUIVALENCE
36	upf_location	upfBoundaryStrategyT	upfLocationE	UPF_LOCATION
37	upf_applies_to	upfBoundaryStrategyT	upfPortDirE	UPF_APPLIES_TO
38	upf_name_prefix	upfBoundaryStrategyT	upfStringT	UPF_NAME_PREFIX
39	upf_name_suffix	upfBoundaryStrategyT	upfStringT	UPF_NAME_SUFFIX
40	upf_clamp_values	upfIsolationStrategyT	List of upfIsolationClampE	UPF_CLAMP_VALUES
41	upf_isolation_ controls	upfIsolationStrategyT	List of upfSignalSenseT	UPF_ISOLATION_ CONTROLS
42	upf_user_clamp_ values	upfIsolationStrategyT	List of upfStringT	UPF_USER_CLAMP _VALUES
43	upf_is_diff_supply _only	upfIsolationStrategyT	upfBooleanT	UPF_IS_DIFF_SUPPLY _ONLY
44	upf_is_force_ isolation	upfIsolationStrategyT	upfBooleanT	UPF_IS_FORCE_ ISOLATION
45	upf_is_no_isolation	upfIsolationStrategyT	upfBooleanT	UPF_IS_NO_ISOLATION
46	upf_is_force_shift	upfLevelShifterStrategyT	upfBooleanT	UPF_IS_FORCE_SHIFT
47	upf_is_no_shift	upfLevelShifterStrategyT	upfBooleanT	UPF_IS_NO_SHIFT
48	upf_level_shift_rule	upfLevelShifterStrategyT	upfLevelShifterRuleE	UPF_LEVEL_SHIFT_RULE
49	upf_threshold_value	upfLevelShifterStrategyT	upfRealT	UPF_THRESHOLD_ VALUE
50	upf_is_illegal	upfStateClassT	upfBooleanT	UPF_IS_ILLEGAL
51	upf_is_active	upfPowerStateT	upfBooleanT	UPF_IS_ACTIVE
52	upf_logic_expr	upfPowerStateT	upfExpressionT	UPF_LOGIC_EXPR
53	upf_supply_expr	upfPowerStateT	upfExpressionT	UPF_SUPPLY_EXPR
54	upf_simstate	upfPowerStateT	upfSimstateE	UPF_SIMSTATE
55	upf_pst_header	upfPowerStateTableT	List of upfBaseNamedT	UPF_PST_HEADER
56	upf_pst_states	upfPowerStateTableT	List of upfPstStateT	UPF_PST_STATES
57	upf_from_states	upfPowerStateTransitionT	List of upfPowerStateT	UPF_FROM_STATES
58	upf_to_states	upfPowerStateTransitionT	List of upfPowerStateT	UPF_TO_STATES
59	upf_switch_expr	upfPowerSwitchStateT	upfExpressionT	UPF_SWITCH_EXPR

Table 36—Mapping between property name and property ID in HDL *(continued)*

60	upf_input_ supply_port	upfPowerSwitchStateT	upfSupplyPortT	UPF_INPUT_ SUPPLY_PORT
61	upf_switch_ output_state	upfPowerSwitchStateT	upfSupplyStateE	UPF_SWITCH_ OUTPUT_STATE
62	upf_supply_states	upfPstStateT	List of upfSupplyPortStateT	UPF_SUPPLY_STATES
63	upf_volt_max	upfSupplyPortStateT	upfRealT	UPF_VOLT_MAX
64	upf_volt_min	upfSupplyPortStateT	upfRealT	UPF_VOLT_MIN
65	upf_volt_nom	upfSupplyPortStateT	upfRealT	UPF_VOLT_NOM
66	upf_supply_state	upfSupplyPortStateT	upfSupplyStateE	UPF_SUPPLY_STATE
67	upf_volt_kind	upfSupplyPortStateT	upfVoltKindE	UPF_VOLT_KIND
68	upf_network_ attributes	upfNetworkClassT	List of upfAttributeT	UPF_NETWORK_ ATTRIBUTES
69	upf_hdl_ implementation	upfNetworkClassT	upfHdlDeclT	UPF_HDL_ IMPLEMENTATION
70	upf_root_driver	upfNetworkClassT	upfNetworkClassT	UPF_ROOT_DRIVER
71	upf_fanin_conn	upfNetClassT	List of upfPortClassT	UPF_FANIN_CONN
72	upf_fanout_conn	upfNetClassT	List of upfPortClassT	UPF_FANOUT_CONN
73	upf_hiconn	upfPortClassT	List of upfNetworkClassT	UPF_HICONN
74	upf_loconn	upfPortClassT	List of upfNetworkClassT	UPF_LOCONN
75	upf_port_dir	upfPortClassT	upfPortDirE	UPF_PORT_DIR
76	upf_ack_delay	upfAckPortT	upfStringT	UPF_ACK_DELAY
77	upf_ref_object	upfNamedRefT	upfBaseNamedT	UPF_REF_OBJECT
78	upf_ref_kind	upfNamedRefT	upfNamedRefKindE	UPF_REF_KIND
79	upf_ack_ports	upfPowerSwitchT	List of upfAckPortT	UPF_ACK_PORTS
80	upf_control_ports	upfPowerSwitchT	List of upfLogicPortT	UPF_CONTROL_PORTS
81	upf_sw_states	upfPowerSwitchT	List of upfPowerSwitchStateT	UPF_SW_STATES
82	upf_input_supply _ports	upfPowerSwitchT	List of upfSupplyPortT	UPF_INPUT_SUPPLY _PORTS
83	upf_output_supply _port	upfPowerSwitchT	upfSupplyPortT	UPF_OUTPUT_SUPPLY _PORT
84	upf_resolve_type	upfSupplyNetT	upfResolveE	UPF_RESOLVE_TYPE
85	upf_sp_states	upfSupplyPortT	List of upfPortStateT	UPF_SP_STATES
86	upf_slice_bits	upfHdlMultiBitSliceT	List of upfHdlNetBitT	UPF_SLICE_BITS
87	upf_lsb	upfHdlMultiBitSliceT	upfIntegerT	UPF_LSB
88	upf_msb	upfHdlMultiBitSliceT	upfIntegerT	UPF_MSB
89	upf_normalized_bits	upfHdlPortMultiBitT	List of upfHdlPortBitT	UPF_NORMALIZED_BITS
90	upf_hdl_width	upfHdlPortMultiBitT	upfIntegerT	UPF_HDL_WIDTH
91	upf_items	upfHdlScopeT	List of upfBaseUpfT	UPF_ITEMS
92	upf_hdl_items	upfHdlScopeT	List of upfHdlDeclT	UPF_HDL_ITEMS
93	upf_hdl_ports	upfHdlScopeT	List of upfHdlDeclT	UPF_HDL_PORTS
94	upf_child_instances	upfHdlScopeT	List of upfHdlScopeT	UPF_CHILD_INSTANCES
95	upf_attr_name	upfAttributeT	upfStringT	UPF_ATTR_NAME

Table 36—Mapping between property name and property ID in HDL *(continued)*

96	upf_attr_value	upfAttributeT	upfStringT	UPF_ATTR_VALUE
97	upf_source_extents	upfCellT	List of upfExtentT	UPF_SOURCE_EXTENTS
98	upf_cell_kind	upfCellT	upfCellKindE	UPF_CELL_KIND
99	upf_cell_origin	upfCellT	upfCellOriginE	UPF_CELL_ORIGIN
100	upf_hdl_cell_kind	upfCellT	upfHdlCellKindE	UPF_HDL_CELL_KIND
101	upf_model_name	upfCellT	upfStringT	UPF_MODEL_NAME
102	upf_expr_operands	upfExpressionT	List of upfBaseNamedT	UPF_EXPR_OPERANDS
103	upf_current_value	upfExpressionT	upfBooleanT	UPF_CURRENT_VALUE
104	upf_expr_string	upfExpressionT	upfStringT	UPF_EXPR_STRING
105	upf_cells	upfExtentT	List of upfBaseHdlT	UPF_CELLS
106	upf_hdl_element	upfExtentT	upfBaseHdlT	UPF_HDL_ELEMENT
107	upf_object	upfExtentT	upfExtentClassT	UPF_OBJECT
108	upf_control_signal	upfSignalSenseT	upfBaseNamedT	UPF_CONTROL_SIGNAL
109	upf_signal_sensitivity	upfSignalSenseT	upfSignalSenseKindE	UPF_SIGNAL_SENSITIVITY
110	upf_voltage	upfSupplyTypeT	upfIntegerT	UPF_VOLTAGE
111	upf_state	upfSupplyTypeT	upfSupplyStateE	UPF_STATE
112	upf_normalized _idx	upfHdlPortBitT, upfHdlNetBitT	upfIntegerT	UPF_NORMALIZED _IDX
113	upf_smallest_ atomic_slice	upfHdlPortBitT, upfHdlNetBitT	upfHdlMultiBitSliceT	UPF_SMALLEST_ ATOMIC_SLICE
114	upf_upper_boundary	upfPowerDomainT	upfHdlScopeT	UPF_UPPER_BOUNDARY
115	upf_next_extent	upfExtentT	upfExtentT	UPF_NEXT_EXTENT

11.2.3 HDL access functions

11.2.3.1 Accessing objects and properties

11.2.3.1.1 upf_get_handle_by_name

Purpose	Get a handle to a given object from the pathname	
Syntax	`upfHandleT upf_get_handle_by_name(upfStringT pathname, upfHandleT relative_to = null);`	
Arguments	`pathname`	A string representing handle ID for an object
	`relative_to`	An optional handle to the object from which the relative pathname is given
Return value	Returns the handle to the specified property or null if not found	

The function upf_get_handle_by_name returns the handle to the object in the information model from the given handle id. The handle id is defined as per 10.3.2.

The pathname can also be a relative pathname when relative_to is passed with a valid handle. In that case, the hierarchical path ID is constructed from the hierarchical path ID of "relative_to" suffixed with pathname string along with appropriate separator character (‘/’, ‘.’) in between.

It shall be an error if:

— pathname is not a valid handle ID

— relative_to is not a valid UPF handle

Syntax examples

Example 1: Get handle to a power domain

SV code

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

Example 2: Get handle to a strategy

SV code

```
initial begin
  upfHandleT scope = upf_get_handle_by_name("/top/dut_i");
  upfHandleT iso   = upf_get_handle_by_name("pd.iso", scope);
  ...
end
```

11.2.3.1.2 upf_query_object_properties

Purpose	Query properties on a given object	
Syntax	`upfHandleT upf_query_object_properties(upfHandleT object_handle, upfPropertyIdE attr);`	
Arguments	`object_handle`	UPF handle of the given object
	`attr`	The enumerated value corresponding to the given property on the object
Return value	Returns the handle to the specified property or null if not found	

The function upf_query_object_properties returns the handle to the property corresponding to the enumerated value passed in attr.

The value returned from this function can be one of the following

— handle to a property of basic type

— handle to an object

— handle to the iterator for list of objects

For basic properties a handle to the property value is returned. The exact value is then accessed from the handle using immediate access functions defined in 11.2.3.2 and 11.2.3.3.

For handle to an iterator, the appropriate iterator access functions need to be used to access the individual elements of the list.

It shall be an error if:

— object_handle is not a valid object
— attr is not a valid property on the given object
— attr is not a valid value defined in Table 36.

NOTE—The function upf_query_object_properties can also return the handle to dynamic properties present on the object.

Syntax examples

Example 1: Get simple-name of power domain

SV code

```
initial begin
  upfHandleT pd = upf_get_handle_by_name("/top/dut_i/pd");
  upfHandleT name = upf_query_object_properties(pd, upf_name);
  $display( "PD Name: %s", upf_get_value_str(name));
end
```

Output

```
  PD Name: pd
```

Example 2: Print full hier-path of creation scope of power domain

SV code

```
initial begin
  upfHandleT pd = upf_get_handle_by_name("/top/dut_i/pd");
  upfHandleT scope = upf_query_object_properties(pd,
                     upf_creation_scope);
  $display( "Creation Scope: %s", upf_query_object_pathname(name));
end
```

Output

```
  Creation Scope: /top/dut_i
```

Example 3: Print isolation strategy name from power domain

SV code

```
initial begin
  upfHandleT pd = upf_get_handle_by_name("/top/dut_i/pd");
  upfHandleT iso_list = upf_query_object_properties(pd,
                        upf_isolation_strategies);
  upfHandleT first_iso = upf_iter_get_next(iso_list);
  if (first_iso) begin
    upfHandleT name = upf_query_object_properties(first_iso,
                        upf_name);
    $display( "Iso Name: %s", upf_get_value_str(name));
  end
end
```

Output

```
  Iso Name: iso
```

Example 4: 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
```

11.2.3.1.3 upf_iter_get_next

Purpose	Get the next handle from the iterator	
Syntax	`upfHandleT upf_iter_get_next(upfHandleT iter_handle);`	
Arguments	`iter_handle`	Handle to the iterator
Return value	Returns the handle to the next element in the iterator or 0 if no element is present	

The upf_iter_get_next function returns the handle to the next element in the iterator.

It shall be an error if iter_handle is not a valid iterator handle.

Syntax examples

Example 1: Print function names of the supply set

SV code

```
initial begin
  upfHandleT ss = upf_get_handle_by_name("/top/dut_i/pd.primary");
  upfHandleT func_list = upf_query_object_properties(pd,
                         upf_functions);
  upfHandleT func = upf_iter_get_next(func_list);
  while (func != 0) begin
    upfHandleT name = upf_query_object_properties(func,
                       upf_name);
    $display( "Function Name: %s", upf_get_value_str(name));
    func = upf_iter_get_next(func_list);
  end
end
```

Output

```
  Function Name: power
  Function name: ground
```

11.2.3.2 Immediate read access

11.2.3.2.1 Overview

All objects in the information model allow read access to the properties including the dynamic properties. The immediate read access returns the current value of the dynamic property at specific time when read access functions are called.

Table 37 provides the mapping of basic properties and the read access routine to get the value.

Table 37 —Immediate read access for basic properties

Type name	SV	VHDL
upfStringT	upf_get_value_str()	upf_get_value_str()
upfIntegerT	upf_get_value_int()	upf_get_value_int()
upfBooleanT	upf_get_value_int()	upf_get_value_int()
All Enumerated Types	upf_get_value_int()	upf_get_value_int()
upfRealT	upf_get_value_real()	upf_get_value_real()

11.2.3.2.2 upf_get_value_str

Purpose	Get the string value from property handle	
Syntax	`upfStringT upf_get_value_str(upfHandleT attr);`	
Arguments	`attr`	Handle to the property
Return value	Returns string value of the given property handle or empty string if error	

The upf_get_value_str function returns the string value of the given property handle.

It shall be an error if attr is not a valid property handle of upfStringT type.

Syntax examples

Example 1: Get simple-name of power domain

SV code

```
initial begin
  upfHandleT pd = upf_get_handle_by_name("/top/dut_i/pd");
  upfHandleT name = upf_query_object_properties(pd, upf_name);
  $display( "PD Name: %s", upf_get_value_str(name));
end
```

Output

```
  PD Name: pd
```

11.2.3.2.3 upf_get_value_int

Purpose	Get the integer value from property handle	
Syntax	`upfIntegerT upf_get_value_int(upfHandleT attr);`	
Arguments	`attr`	Handle to the property
Return value	Returns integer value of the given property handle	

The upf_get_value_int function returns the integer value of the given property handle.

It shall be an error if attr is not a valid property handle of upfIntegerT or equivalent enumerated types.

NOTE—The function upf_get_value_int() is used to access values of enumerated and Boolean types. In case of VHDL language, the return value of the function needs to be converted the appropriate enumerated/Boolean type to avoid syntax errors.

Syntax examples

Example 1: Get UPF line number of power domain

SV code

```
initial begin
  upfHandleT pd = upf_get_handle_by_name("/top/dut_i/pd");
  upfHandleT file_line = upf_query_object_properties(pd, upf_line);
  $display( "UPF Line: %d", upf_get_value_int(file_line));
end
```

Output

```
  UPF Line: 22
```

Example 2: Get level-shifter rule

VHDL code

```
process
  variable ls: upfHandleT;
  variable rule_attr: upfHandleT;
  variable rule: upfLevelShifterRuleE;
begin
  ls := upf_get_handle_by_name("/top/dut_i/pd.ls");
  rule_attr := upf_query_object_properties(pd, upf_level_shift_rule);
  rule := upfLevelShifterRuleE'val(upf_get_value_int(rule_attr));
  ...
end
```

11.2.3.2.4 upf_get_value_real

Purpose	Get the real value from property handle	
Syntax	`upfRealT upf_get_value_real(upfHandleT attr);`	
Arguments	`attr`	Handle to the property
Return value	Returns real value of the given property handle	

The upf_get_value_real function returns the real value of the given property handle.

It shall be an error if property is not a valid property handle of upfRealT.

Syntax examples

Example 1: Get threshold value from the level-shifter strategy

SV code

```
initial begin
  upfHandleT ls = upf_get_handle_by_name("/top/dut_i/pd.ls");
  upfHandleT threshold_value = upf_query_object_properties(ls,
                                 upf_threshold_value );
  $display( "Threshold Value : %f",
            upf_get_value_real(threshold_value));
end
```

Output

```
  Threshold Value: 1.000000
```

11.2.3.2.5 upf_get_label_upfSupplyTypeT

Purpose	Get the value of supply net type from the property handle	
Syntax	`upfSupplyTypeT  upf_get_label_upfSupplyTypeT(upfHandleT attr);`	
Arguments	`attr`	Handle to the property
Return value	Returns upfSupplyTypeT for given property handle	

The upf_get_value_upfSupplyTypeT function returns the label of type upfSupplyTypeT value on the given property handle.

It shall be an error if property is not a valid property handle of upfSupplyTypeT.

Syntax examples

Example 1: Get value of a supply

SV code

```
upfHandleT snetHandle;
upfHandleT snetValHandle;
upfSupplyTypeT snet;
initial begin
  snetHandle = upf_get_handle_by_name("/top/dut_i/VDD@upfSupplyNetT");
  snetValHandle = upf_query_object_properties(snetHandle,
                  UPF_CURRENT_VALUE);
  if (upf_handle_in_class(snetValHandle, UPF_SUPPLY_TYPE)) begin
    snet = upf_get_label_upfSupplyTypeT(snetValHandle);
    $display("The state of supply: %s", snet.state.name);
  end
end
```

Output

```
The state of supply: FULL_ON
```

11.2.3.2.6 get_supply_value

Purpose	Get the supply value of a net	
Syntax	`upfSupplyTypeT  get_supply_value(string name);`	
Arguments	`name`	A string representing pathname of supply net
Return value	Returns upfSupplyTypeT for given property handle	

The get_supply_value() returns the aggregate supply net value of the specified supply port or supply net.

It shall be an error if name is not a UPF created supply net/port or an HDL object of upfSupplyTypeT or upfSupplyObjT.

The following is the description of get_supply_value function based on basic access functions.

SV code

```
function automatic upfSupplyTypeT get_supply_value
    (string name);            // A string representing
                              // pathname of supply net
    upfHandleT hSupplyNet;
    upfHandleT hSupplyNetValue;

    hSupplyNet      = upf_get_handle_by_name(name);
    hSupplyNetValue = upf_query_object_properties(hSupplyNet,
                                                    UPF_CURRENT_VALUE);

    // Returns upfSupplyTypeT for given property handle.
    return upf_get_label_upfSupplyTypeT(hSupplyNetValue);
  endfunction:get_supply_value
```

Syntax examples

Example 1: Reading the supply value

SV code

```
upfSupplyTypeT vdd_value;
...
initial begin
  vdd_value = get_supply_value("/top/dut_i/vdd");
  ...
  vdd_value = get_supply_value("vdd_local");
  ...
  vdd_value = get_supply_value("vdd_obj");
  ...
end
```

11.2.3.2.7 get_supply_voltage

Purpose	Get the supply voltage of a net	
Syntax	`upfRealT get_supply_voltage(upfSupplyTypeT arg);`	
Arguments	`arg`	An HDL object of upfSupplyTypeT
Return value	Returns supply voltage of given upfSupplyTypeT	

The get_supply_voltage returns the floating-point representation of the voltage value of the specified supply net value in Volts.

The following is the description of get_supply_voltage function based on basic access functions

SV code

```
function upfRealT get_supply_voltage(upfSupplyTypeT arg);
  return upf_convert_int_uvolts_to_real_volts(arg.voltage);
endfunction
```

Syntax examples

Example 1: Reading the supply value

SV code

```
upfRealT vdd_voltage;
upfSupplyTypeT vdd_local;
upfSupplyObjT vdd_obj;
...
```

```
initial begin
  vdd_voltage = get_supply_voltage(get_supply_value("/top/dut_i/vdd"));
  ...
  vdd_voltage = get_supply_voltage(vdd_local);
  ...
  vdd_voltage = get_supply_voltage(vdd_obj.current_value);
  ...
end
```

11.2.3.2.8 get_supply_on_state

Purpose	Get the supply FULL_ON state	
Syntax	`upfBooleanT get_supply_on_state(upfSupplyTypeT arg);`	
Arguments	`arg`	An object of upfSupplyTypeT
Return value	Returns 1 if the upfSupplyTypeT is FULL_ON	

The get_supply_on_state returns the on/off state of the specified supply net. It returns 1 when supply net state is FULL_ON or PARTIAL_ON and set_partial_on_translation FULL_ON is called. It returns 0 in other cases.

The following is the description of get_supply_on_state function based on basic access functions.

SV code

```
  function automatic upfBooleanT get_supply_on_state
    (supply_net_type arg);    //An object of upfSupplyTypeT
    upfBooleanT     state = 0;

    //The get_supply_on_state returns the on/off state of the specified
    //supply net. It returns 1 when supply net state is FULL_ON or
    //PARTIAL_ON and set_partial_on_translation FULL_ON is called. It
    //returns 0 in other cases.

    // The following is the equation/Logic to return the supply_on_state
    // value for this function call.  However, this function
    // relies on the value of "partial_on_translation" set from
    // upf commands.  The variable partial_on_translation must be
    // extracted from the simulator in some form or another.
    // This call to "$partial_on_translation@ is just a model and may
    // not be the final implementation of the simulator


    if ((arg.state === FULL_ON) ||
        ((arg.state === PARTIAL_ON ) &&
         ($partial_on_translation === FULL_ON)) )
      state = 1'b1;
    else
      state = 1'b0;

    return (state);
  endfunction:get_supply_on_state
```

Syntax examples

Example 1: Reading the supply value

SV code

```
bit is_on;
upfSupplyTypeT vdd_local;
upfSupplyObjT vdd_obj;
...
initial begin
  is_on = get_supply_on_state(get_supply_value("/top/dut_i/vdd"));
  ...
  is_on = get_supply_on_state(vdd_local);
  ...
  is_on = get_supply_on_state(vdd_obj.current_value);
  ...
end
```

11.2.3.2.9 get_supply_state

Purpose	Get the state from a supply net	
Syntax	`upfSupplyStateE get_supply_ state(upfSupplyTypeT arg);`	
Arguments	`arg`	An object of upfSupplyTypeT
Return value	Returns state of the supply net	

The `get_supply_state` returns the full/partial on/off state of the specified supply net.

The following is the description of get_supply_state function based on basic access functions.

SV code

```
  function automatic upfSupplyStateE get_supply_state
    (upfSupplyTypeT arg);     //  An object of upfSupplyTypeT

    // Return Value: Returns state of the supply net
    return (arg.state);
  endfunction:get_supply_state
```

Syntax examples

Example 1: Reading the supply value

SV code

```
state net_state;
upfSupplyTypeT vdd_local;
upfSupplyObjT vdd_obj;
...
initial begin
  snet_state = get_supply_state(get_supply_value("/top/dut_i/vdd"));
  ...
  snet_state = get_supply_state(vdd_local);
  ...
  snet_state = get_supply_state(vdd_obj.current_value);
  ...
end
```

11.2.3.3 Immediate write access

The following objects in the information model allow write access to the dynamic properties defined on them.

a) upfPowerStateT

b) upfLogicNetT

c) upfLogicPortT

d) upfSupplyNetT

e) upfSupplyPortT

The write access is allowed only in special circumstances when the object does not have any existing driver.

The write access to the upf_is_active property of an object of upfPowerStateT type is allowed when the object is a Deferred power state (see 4.7.3).

The write access to upfLogicNetT and upfSupplyNetT is allowed when it has no driver associated with it.

The write access to upfLogicPortT/upfSupplyPortT is allowed when it is a root driver.

The immediate write access results in the value being transferred immediately when the function is called and the value remains on the object until another call to write is made. Since the object does not have any other driver associated with it, it maintains the value until it is changed by another call.

Table 38 provides the mapping between basic property types and corresponding hdl function to write the value.

Table 38 —Immediate write access for basic properties

Type name	SV	VHDL
upfStringT	-	-
upfIntegerT	upf_set_value_int()	upf_set_value_int()
upfBooleanT	upf_set_value_int()	upf_set_value_int()
All Enumerated Types	upf_set_value_int()	upf_set_value_int()
upfRealT	-	-
upfSupplyTypeT	upf_set_value_upfSupplyTypeT()	upf_set_value_upfSupplyTypeT()
NOTE—upfStringT and upfRealT types do not have write access as there aren't any dynamic properties of these types. In order to set voltage on a supply type object, the real value needs to be converted to integer in micro-volts and then set using upf_set_value_int(). See 11.2.3.3.1 for more details.		

It shall be an error when:

— Write is performed on any other property than the dynamic properties.

— Write is performed on the object and there is an existing driver.

11.2.3.3.1 upf_set_value_int

Purpose	Sets the integer value on the dynamic property	
Syntax	`upfBooleanT upf_set_value_int(upfHandleT attr, upfIntegerT value);`	
Arguments	`attr`	Handle to the property
	`value`	Value
Return value	Returns 1 on success, 0 on failure	

The upf_set_value_int() provides an immediate write access to integer type dynamic properties.

It shall be an error when:

— attr is a null handle.

— attr is not a valid dynamic property of upfIntegerT type.

Syntax examples

Example 1: Change the state of root supply port and logic port acting as isolation control during simulation

SV code

```
initial begin
  upfHandleT vdd = upf_get_handle_by_name("/top/dut_i/vdd");
  upfHandleT supplyValue = upf_query_object_properties(vdd,
                          UPF_CURRENT_VALUE);
  upfHandleT state = upf_query_object_properties(supplyValue,
                     upf_state);
  upf_set_value_int(state, (upfIntegerT)FULL_ON);
  upfHandleT iso_ctrl = upf_get_handle_by_name("/top/iso_logic");
  upfHandleT ctrl_value = upf_get_object_properties(iso_ctrl,
                         UPF_CURRENT_VALUE);
  (void)upf_set_value_int(ctrl_value, 0);
  #10 (void)upf_set_value_int(ctrl_value, 1);
  #10 (void)upf_set_value_int(ctrl_value, 0);
end
```

11.2.3.3.2 upf_set_value_upfSupplyTypeT

Purpose	Sets the integer value on the dynamic property	
Syntax	`upfBooleanT upf_set_value_upfSupplyTypeT(upfHandleT attr, upfSupplyTypeT value);`	
Arguments	`attr`	Handle to the property
	`value`	Value of type upfSupplyTypeT
Return value	Returns 1 on success, 0 on failure	

The upf_set_value_upfSupplyTypeT() provides an immediate write access to the current value of a supply type object.

It shall be an error when:

— attr is a null handle.

— attr is not a valid dynamic property of upfSupplyTypeT type.

Syntax examples

Example 1: Change the value of UPF root supply port from a supply net declared in testbench

SV code

```
initial begin
  upfHandleT vdd = upf_get_handle_by_name("/top/dut_i/vdd");
  upfHandleT supplyValue = upf_query_object_properties(vdd,
                           UPF_CURRENT_VALUE);
  upf_set_value_upfSupplyTypeT(supplyValue,
                                get_supply_value("vddlocal");
end
```

11.2.3.3.3 supply_on

Purpose	Change state of supply to FULL_ON and specify voltage	
Syntax	`upfBooleanT supply_on(string supply_name, real value = 1.0);`	
Arguments	`supply_name`	A string representing pathname of the root supply
	`value`	Voltage value in real
Return value	Returns 1 on success, 0 on failure	

The supply_on function immediately changes the current value of supply type object to FULL_ON and specified voltage by effecting the changes to root supply driver of the given supply type object.

It shall be an error if:

— supply_name is not a UPF created supply net/port or an HDL object of upfSupplyTypeT or upfSupplyObjT.

— supply_name is a resolved supply net, output supply port of a Power switch or continuously driven from HDL source.

The following is the description of supply_on function based on basic access functions.

SV code

```
function automatic upfBooleanT supply_on
    (upfStringT supply_name,      // A string representing pathname
                                // of the root supply
     upfRealT value = 1.0);       // Voltage value in real

    upfBooleanT bStatus = 0;
    upfHandleT  hState;
    upfHandleT  hSupplyValue;
    upfHandleT  hSupply;
    upfHandleT  hVoltage;
    upfHandleT  hRootSupply;

    hSupply = upf_get_handle_by_name(supply_name);
    if (hSupply === null) return 0;

    hRootSupply =  upf_query_object_properties(hSupply,
                               UPF_ROOT_DRIVER);
    if (hRootSupply != null)
      hSupply = hRootSupply;

    hSupplyValue =  upf_query_object_properties(hSupply,
                               UPF_CURRENT_VALUE);
    if (hSupplyValue === null) return 0;

    hVoltage = upf_query_object_properties(hSupplyValue,
                                       UPF_VOLTAGE);
    if (hVoltage === null) return 0;

    bStatus = upf_set_value_int(hVoltage,
                 upf_convert_real_volts_to_int_uvolts(value));
    if (!bStatus) return 0;

    hState  = upf_query_object_properties(hSupplyValue, UPF_STATE);

    bStatus = upf_set_value_int(hState, FULL_ON);
    return bStatus;
  endfunction:supply_on
```

Syntax examples

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_on("vdd_local", 0.9);
  ...
  status = supply_on("vdd_value", 0.9);
  ...
end
```

11.2.3.3.4 supply_off

Purpose	Change state of supply to OFF	
Syntax	`upfBooleanT supply_off(upfStringT supply_name);`	
Arguments	`supply_name`	A hierarchical path ID for a root supply
Return value	Returns 1 on success, 0 on failure	

The supply_off function immediately changes the state of current value of supply type object to OFF and voltage to 0.

It shall be an error if:

— supply_name is not a UPF created supply net/port or an HDL object of upfSupplyTypeT or upfSupplyObjT.

— supply_name is a resolved supply net, output supply port of a Power switch or continuously driven from HDL source.

The following is the description of supply_off function based on basic access functions.

SV code

```
function automatic upfBooleanT supply_off
    (upfStringT supply_nzme);  // A hierarchical path ID for a root supply.

    upfBooleanT bStatus = 0;
    upfHandleT  hState;
    upfHandleT  hSupplyValue;
    upfHandleT  hSupply;
    upfHandleT  hVoltage;
    upfHandleT  hRootSupply;

    hSupply = upf_get_handle_by_name(supply_name);
    if (hSupply === null) return 0;

    hRootSupply =  upf_query_object_properties(hSupply,
                                UPF_ROOT_DRIVER);
    if (hRootSupply != null)
      hSupply = hRootSupply;

    hSupplyValue = upf_query_object_properties(hSupply,
                                           UPF_CURRENT_VALUE);
    if (hSupplyValue === null) return 0;

    hVoltage = upf_query_object_properties(hSupplyValue,
                                        UPF_VOLTAGE);
    if (hVoltage === null) return 0;

    bStatus = upf_set_value_int(hVoltage, 0);
    if (!bStatus) return 0;

    hState  = upf_query_object_properties(hSupplyValue, UPF_STATE);
    bStatus = upf_set_value_int(hState, OFF);

    return bStatus;
endfunction:supply_off
```

Syntax examples

Example 1: Changing supply voltages

SV code

```
upfSupplyObjT vdd_local;
upfSupplyTypeT vdd_value;
initial begin
  status = supply_off("/top/dut_i/vdd");
  ...
  status = supply_off("vdd_local");
  ...
  status = supply_off("vdd_value");
  ...
end
```

11.2.3.3.5 supply_partial_on

Purpose	Change state of supply to PARTIAL_ON and specify voltage	
Syntax	`upfBooleanT supply_partial_on(upfStringT supply_name, upfRealT value = 1.0);`	
Arguments	`supply_name`	A string representing pathname of the root supply
	`Value`	Voltage value in real
Return value	Returns 1 on success, 0 on failure	

The supply_partial_on function immediately changes the current value of supply type object to PARTIAL_ON and specified voltage.

It shall be an error if:

— supply_name is not a UPF created supply net/port or an HDL object of upfSupplyTypeT or upfSupplyObjT.

— supply_name is a resolved supply net, output supply port of a Power switch or continuously driven from HDL source.

The following is the description of supply_partial_on function based on basic access functions.

SV code

```
function automatic upfBooleanT supply_partial_on
    (upfStringT supply_name,   // A string representing pathname of the root
supply
     upfRealT   value = 1.0); // Voltage value in real

    upfBooleanT bStatus = 0;
    upfHandleT  hState;
    upfHandleT  hSupplyValue;
    upfHandleT  hSupply;
    upfHandleT  hVoltage;
    upfHandleT  hRootSupply;
    hSupply = upf_get_handle_by_name(supply_name);
    if (hSupply === null) return 0;

    hRootSupply =  upf_query_object_properties(hSupply,
                                UPF_ROOT_DRIVER);
    if (hRootSupply != null)
      hSupply = hRootSupply;

    hSupplyValue = upf_query_object_properties(hSupply,
                                            UPF_CURRENT_VALUE);
    if (hSupplyValue === null) return 0;

    hVoltage = upf_query_object_properties(hSupplyValue,
                                        UPF_VOLTAGE);
    if (hVoltage === null) return 0;

    bStatus = upf_set_value_int(hVoltage,
                upf_convert_real_volts_to_int_uvolts(value));
    if (!bStatus) return 0;

    hState  = upf_query_object_properties(hSupplyValue, UPF_STATE);
    bStatus = upf_set_value_int(hState, PARTIAL_ON);

    return bStatus;
  endfunction:supply_partial_on
```

Syntax examples

Example 1: Changing supply voltages

SV code

```
upfSupplyObjT vdd_local;
upfSupplyTypeT vdd_value;
initial begin
  status = supply_partial_on("/top/dut_i/vdd", 1.2);
  ...
  status = supply_partial_on("vdd_local", 0.9);
  ...
  status = supply_partial_on("vdd_value", 0.9);
  ...
end
```

11.2.3.3.6 set_supply_state

Purpose	Appy supply state to a given named object	
Syntax	`upfBooleanT set_supply_state (upfStringT object_name, upfSupplyStateE supply_state);`	
Arguments	`object_name`	The hierarchical path ID of a supply port, net or supply set function
	`supply_state`	Enumerated value representing the state of the supply net
Return value	Returns 1 on success, 0 on failure	

The `set_supply_state` function applies a supply state to a given named supply net object.

It shall be an error if:

— object_name is not a valid supply port, net, or supply set function name.
— Different values are assigned to the supply object in the same cycle.

The following is the description of set_supply_state function based on basic access functions.

SV code

```
  ///////////////////////////////////////
  // Purpose: Assign the specified supply state to this object.
  // Return Value: Returns 1 on success, 0 on failure
  function automatic upfBooleanT set_supply_state
    (upfStringT object_name,  // the hierarchical path ID of a supply port,
                              //   net, or supply set function
     upfSupplyStateE supply_state);// one of OFF, PARTIAL_ON, FULL_ON,
                              //   UNDETERMINED

    // It shall be an error if :
    //   1) object_name is not a valid supply port, net, or
    //      supply set function name
    //   2) different values are assigned to the supply object
    //      in the same cycle
    upfBooleanT bStatus = 0;
    upfHandleT  hState;
    upfHandleT  hSupplyValue;
    upfHandleT  hSupply;
    upfHandleT  hRootSupply;

    hSupply = upf_get_handle_by_name(object_name);
    if (hSupply == null) return 0;

    hRootSupply =  upf_query_object_properties(hSupply,
                                UPF_ROOT_DRIVER);
    if (hRootSupply != null)
      hSupply = hRootSupply;

    hSupplyValue = upf_query_object_properties(hSupply,
                                            UPF_CURRENT_VALUE);
    if (hSupplyValue === null) return 0;

    hState  = upf_query_object_properties(hSupplyValue, UPF_STATE);
    bStatus = upf_set_value_int(hState, supply_state);

    return bStatus;
  endfunction:set_supply_state
```

Syntax examples

Example 1: Change state of the primary power and ground of a domain

SV code

```
initial begin
  set_supply_state("PD.primary.power", FULL_ON);
  set_supply_state("PD.primary.ground",FULL_ON);
end
```

11.2.3.3.7 set_power_state_by_handle

Purpose	Activates the specified power state of an object	
Syntax	`bit set_power_state_by_handle(upfHandleT object, upfHandleT power_state);`	
Arguments	`object`	Handle to the UPF object
	`power_state`	Handle of the power state present on the object
Return value	Returns 1 on success, 0 on failure	

The set_power_state_by_handle function activates the specified power state (see 9.3.1) of the given object.

It shall be an error if:

- object is not a valid handle.
- power_state is not a valid handle of a power state present on object.

Syntax examples

Example 1: Change state of the primary supply set of domain

SV code

```
initial begin
  upfHandleT ss = upf_get_handle_by_name("/top/dut_i/PD.primary");
  upfHandleT on = upf_get_handle_by_name("ON", ss);
  upfHandleT off = upf_get_handle_by_name("OFF", ss);
  set_power_state_by_handle(ss, on);
  #10 set_power_state_by_handle(ss, off);
  #10 set_power_state_by_handle(ss, on);
end
```

11.2.3.3.8 set_power_state

Purpose	Activates the specified power state of an object	
Syntax	`upfBooleanT set_power_state(string object, string power_state);`	
Arguments	`object`	Hierarchical path ID of object having power state
	`power_state`	Relative path ID of power state with respect to the object
Return value	Returns 1 on success, 0 on failure	

The set_power_state function (see 9.3.1) activates the specified power state of the named object by invoking the set_power_state_by_handle function.

It shall be an error if:

- object is not a valid pathname for power domain or supply set.
- power_state is not a valid power state present on power domain or supply set.

The following is the description of set_power_state function based on basic access functions.

SV code

```
  function automatic upfBooleanT set_power_state
    (upfStringT object_name, // Hierarchical path ID of object
                             //   having power state
     upfStringT power_state);// Relative path ID of power state
                             //   with respect to the object

    // It shall be an error if :
    //   1) object_name is not a valid name of a supply set, power
    //      domain, composite domain, group, model, or instance
    //   2) power_state is not the name of a power state of the
    //      specified object
    //   3) different power states that are not related by refinement
    //      are made active for this object in the same cycle

    upfBooleanT bStatus = 0;
    upfHandleT  hPd_ss;
    upfHandleT  hPower_state_handle;


    hPd_ss = upf_get_handle_by_name(object_name);
    if (hPd_ss == null) return 0;

    hPower_state_handle = upf_get_handle_by_name(power_state,
                                                 hPd_ss);
    if (hPower_state_handle == null) return 0;

    bStatus = set_power_state_by_handle(hPd_ss,
                                        hPower_state_handle);
    return bStatus;
  endfunction:set_power_state
```

Syntax examples

Example 1: Change state of the domain

SV code

```
initial begin
  set_power_state("/top/dut_i/PD", "domain_on");
  #10 set_power_state("/top/dut_i/PD", "domain_off");
  #10 set_power_state("/top/dut_i/PD", "domain_on");
end
```

11.2.3.4 Continuous access

There is also a continuous access provided for objects which have native HDL representation. This access enables continuous monitoring of dynamic values of an object in the information model. It enables user to sensitize an always block or process statement using dynamic values on the objects.

The continuous access is achieved by declaring an object of corresponding native HDL representation type defined in the HDL package and then calling the upf_create_object_mirror function to create the mirroring relationship. The continuous access is only allowed in one direction, i.e., from source to destination.

11.2.3.4.1 upf_create_object_mirror

Purpose	Create a continuous monitor that monitors the dynamic property on the given object	
Syntax	`upfBooleanT upf_create_object_mirror(upfStringT src, upfStringT dst);`	
Arguments	`src`	A string representing hierarchical path ID of the source object whose value will be continuously monitored
	`dst`	A string representing hierarchical path ID of the destination object on which the value will be transferred from source object
Return value	Returns 1 when mirroring is successful or 0 otherwise	

The function upf_create_object_mirror creates the mirroring relationship from **src** object to **dst** object. This function can be used to provide continuous read access to the dynamic values of object from the information model in HDL environment. To achieve this, user declares a HDL object in local scope of the corresponding native HDL representation type. The upf_create_object_mirror function is called with src as the object in the information model and dst as the local object. This establishes the mirroring relationship and the values from the src object is continuously transferred to the local object. In this case, the handle field of the local HDL object also maintains the handle information of the src object. This can be used to query other properties present on the src object.

The function can also be used to transfer values to objects in information model when the object (which is the dst) does not have an existing driver and supports write access (see 11.2.3.3). In such case, the user has to declare a local object of matching native HDL representation and assign values just like an HDL object to the field that represents the dynamic property on the dst object. The upf_create_object_mirror function is called where src becomes the local object and dst is the object in the information model. In this case, the handle field of the local object is initialized to 0. See Example 3 for more details.

The upf_create_object_mirror function needs to be called only once for a set of src, dst pair. This can be achieved by an initial block in SV or a process with wait statement in VHDL.

The string specified in src or dst can also be a relative pathname. In such case, the handle ID is constructed from handle ID of the current instance scope in which the function is called.

It shall be an error if:

— src and dst represent objects that do not have native HDL representation (see 11.2.2.3).

— dst does not support write access.

— dst already has a driver associated with it.

— upf_create_object_mirror is called multiple times on same src/dst pair.

Syntax examples

Example 1: 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
```

Example 2: Check value of retention save signal

VHDL code

```
...
  signal save: upfBooleanObjT;
begin
  process
    variable status : upfBooleanT;
  begin
    status := upf_create_object_mirror("/top/dut_i/pd.ret.save_signal",
              "save");
    wait;
  end process;
  process (save.current_value)
  begin
    ...
  end process;
end architecture;
```

Example 3: Drive the value of logic port from HDL

SV code

```
module tb;
  upfBooleanObjT iso_ctrl;
  upfBooleanT status;
  initial begin
    status = upf_create_object_mirror("iso_ctrl",
             "/top/dut_i/logic_iso_ctrl");
  end
  initial begin
    iso_ctr.handle = 0;
    iso_ctrl.current_value = 1’b0;
    #10 iso_ctrl.current_value = 1’b1;
    #10 iso_ctrl.current_value = 1’b0;
  end
endmodule
```

11.2.3.5 Utility functions

11.2.3.5.1 upf_query_object_type

Purpose	Get the class id of the given object handle	
Syntax	`upfClassIdE upf_query_object_type(upfHandleT handle);`	
Arguments	`handle`	Handle to the object or property
Return value	Returns enumerated value representing class of the given object or property	

The upf_query_object_type function returns the enumerated value representing the class of the given object or property handle.

It shall be an error if handle is not a valid object/property handle.

NOTE—The upf_query_object_type function defined in HDL returns the class IDs which are slightly different than what is returned in its Tcl counterpart (see 11.1.2.2). In the Tcl version, the class ID is the class name, but in HDL it is the enumerated literal defined in upfClassIdE type.

Syntax examples

Example 1: Get UPF line number of power domain

SV code

```
initial begin
  upfHandleT pd = upf_get_handle_by_name("/top/dut_i/pd");
  upfHandleT file_line = upf_query_object_properties(pd, upf_line);
  upfClassIdE pd_type = upf_query_object_type(pd);
  if (upf_query_object_type(file_line) == UPF_INTEGER) begin
    $display( "UPF Line: %d", upf_get_value_int(file_line));
  end
end
```

Output

```
UPF Line: 22
```

11.2.3.5.2 upf_handle_in_class

Purpose	Check if object belongs to particular class	
Syntax	`upfBooleanT upf_handle_in_class(upfHandleT handle, upfClassIdE class_id);`	
Arguments	`handle`	Handle to the object or property
	`class_id`	The enumerated value representing the class of the object
Return value	Returns 1 when handle belongs to class and 0 otherwise	

The upf_handle_in_class function returns 1 when the object belongs to the specified class and 0 otherwise. This function is used to check for the class membership of the given handle and is used to write more robust HDL description and avoiding error scenarios.

It shall be an error if:

— handle is not a valid object/property handle.

— class_id is not a valid enumerated value defined in upfClassIdE.

Syntax examples

Example 1: Get UPF line number of power domain

SV code

```
initial begin
  upfHandleT pd = upf_get_handle_by_name("/top/dut_i/pd");
  upfHandleT file_line = upf_query_object_properties(pd, upf_line);
  if (upf_handle_in_class(file_line, UPF_INTEGER)) begin
    $display( "UPF Line: %d", upf_get_value_int(file_line));
  end
end
```

Output

```
UPF Line: 22
```

11.2.3.5.3 upf_query_object_pathname

Purpose	Get the pathname of the given handle	
Syntax	`upfStringT upf_query_object_pathname(upfHandleT object, upfHandleT relative_to = null);`	
Arguments	`object`	Handle to a valid object
	`relative_to`	An optional handle to the object from which the relative pathname is required
Return value	Returns the hierarchical pathname of the given object	

The function upf_get_handle_by_name returns the valid hierarchical pathname of the given handle. The returned pathname for an HDL object represents a valid RTL style name and can be used to query information from other information models. See 10.4.3.2 for more details.

The behavior of the function is similar to the Tcl query upf_query_object_pathname (see 11.1.2.4)

It shall be an error if:

- object is not a valid UPF handle.
- relative_to is not a valid UPF handle.
- relative_to is not in the ancestor hierarchy of the object.

Syntax examples

Example 1: Get relative path of power domain

SV code

```
initial begin
  upfHandleT pd = upf_get_handle_by_name("/top/dut_i/pd");
  upfHandleT top = upf_get_handle_by_name("/top");
  $display("PD: %s", upf_query_object_pathname(pd, top));
  ...
end
```

Output

```
PD: dut_i/pd
```

Example 2: Get RTL name of multi-bit slice

SV code

```
initial begin
  upfHandleT net = upf_get_handle_by_name(
                   "/top/dut_i/mid/net@63:32");
  upfHandleT parent = upf_query_object_properties(net, upf_parent);
  upfHandleT name = upf_query_object_properties(net, upf_name);
  $display("RTL Name of %s is %s",
           upf_get_value_str(name),
           upf_query_object_pathname(net, parent));
  ...
end
```

Output

```
RTL Name of net@63:32 is net.f1(2)
```

11.2.3.5.4 upf_convert_real_volts_to_int_uvolts

Purpose	Get the pathname of the given handle	
Syntax	`upfIntegerT upf_convert_real_volts_to_int_uvolts(upfRealT volt);`	
Arguments	`volt`	Voltage value in volts represented as a real number
Return value	Returns the converted value of voltage in real converted to integer value in micro volts	

The function upf_convert_real_volts_to_int_uvolts() converts the voltage value in volts represented as real value to micro volts represented as integer.

SV code

```
  function automatic upfIntegerT upf_convert_real_volts_to_int_uvolts
      (upfRealT volt);    // Voltage value in volts represented
                          //  as a real number

    return (volt * 1.0E6); // returns int value in uV
  endfunction:upf_convert_real_volts_to_int_uvolts
```

Syntax examples

Example 1: Set voltage value on a supply net

SV code

```
initial begin
  upfHandleT vdd = upf_get_handle_by_name("/top/dut_i/vdd");
  upfHandleT supplyValueH = upf_query_object_properties(vdd,
                         UPF_CURRENT_VALUE);
  upfHandleT voltageH = upf_query_object_properties(supplyValueH,
                       upf_voltage);
  (void)upf_set_value_int(voltageH,
        upf_convert_real_volts_to_int_uvolts(1.0))
end
```

11.2.3.5.5 upf_convert_int_uvolts_to_real_volts

Purpose	Get the pathname of the given handle	
Syntax	`upfRealT upf_convert_int_uvolts_to_real_volts (upfIntegerT uvolt);`	
Arguments	`uvolt`	Voltage value in micro volts represented as a integer number
Return value	Returns the converted value of voltage in micro volts represented as integer to volts represented as real value	

The function upf_convert_int_uvolts_to_real_volts () converts the voltage in micro volts represented as integer to volts represented as real value.

SV code

```
  function automatic upfRealT upf_convert_int_uvolts_to_real_volts
    (upfIntegerT uvolt);  // Voltage value in micro volts
                          //   represented as a integer number

    return (uvolt / 1.0E6);
  endfunction:upf_convert_int_uvolts_to_real_volts
```

Syntax examples

Example 1: Get voltage of a supply net

SV code

```
initial begin
  upfHandleT vdd = upf_get_handle_by_name("/top/dut_i/vdd");
  upfHandleT supplyValueH = upf_query_object_properties(vdd,
                            UPF_CURRENT_VALUE);
  upfHandleT voltageH = upf_query_object_properties(supplyValueH,
                        upf_voltage);
  upfRealT volts = upf_convert_int_uvolts_to_real_volts(
                   upf_get_value_int(voltageH));
  $display("Voltage of %s is %f V",
           upf_query_object_pathname(vdd),
           volts);
end
```

Output

```
  Voltage of /top/dut_i/vdd is 1.000000 V
```

11.2.4 SystemVerilog UPF package

```
////////////////////////////////////////////////////////////////////
// 3.0 Package Declaration
////////////////////////////////////////////////////////////////////


package UPF;

  //////////////////////////////////////
  // basic types
  //////////////////////////////////////

  typedef string      upfStringT;
  typedef int         upfIntegerT;
  typedef bit         upfBooleanT;
  typedef real        upfRealT;
  typedef chandle     upfHandleT;
  typedef int         upfIteratorT;

  typedef enum
    {OFF,          // off_state or reference to OFF for
                   //   supply net/port state
     FULL_ON,      // on_state or reference to FULL_ON
                   //   for supply net/port state
     PARTIAL_ON,   // on_partial_state or reference to
                   //   PARTIAL_ON for supply net/port state
     UNDETERMINED  // error_state or reference to UNDETERMINED
```

```
                 //  for supply net/port state
  } upfSupplyStateE;

typedef struct
  {upfSupplyStateE state;
    upfIntegerT    voltage;   // Voltage in microvolts
  } upfSupplyTypeT;

// Legacy support: backward compatibility
typedef upfSupplyStateE state;
typedef upfSupplyTypeT  supply_net_type;


//////////////////////////////////////
// native HDL Representation
//////////////////////////////////////

typedef struct
  {upfHandleT         handle;
   upfBooleanT        is_active;
  }upfPowerStateObjT;

typedef struct
  {upfHandleT         handle;
   upfPowerStateObjT  current_state;
  }upfPdSsObjT;

typedef struct
  {upfHandleT         handle;
   upfBooleanT        current_value;
  }upfBooleanObjT;

typedef struct
  {upfHandleT         handle;
   supply_net_type    current_value;
  }upfSupplyObjT;


//////////////////////////////////////
// HDL Type Mapping
//////////////////////////////////////

typedef upfPdSsObjT       upfCompositeDomainT;
typedef upfPdSsObjT       upfPowerDomainT;
typedef upfPdSsObjT       upfSupplySetT;

typedef upfPowerStateObjT upfPStStateT;
typedef upfPowerStateObjT upfPowerStateT;

typedef upfBooleanObjT    upfAckPortT;
typedef upfBooleanObjT    upfExpressionT;
typedef upfBooleanObjT    upfLogicNetT;
typedef upfBooleanObjT    upfLogicPortT;

typedef upfSupplyObjT     upfSupplyNetT;
typedef upfSupplyObjT     upfSupplyPortT;

typedef upfHandleT        upfAttributeT;
typedef upfHandleT        upfCellT;
typedef upfHandleT        upfExtentT;
typedef upfHandleT        upfHdlMultiBitSliceT;
typedef upfHandleT        upfHdlNetBitT;
typedef upfHandleT        upfHdlNetMultiBitT;
typedef upfHandleT        upfHdlPortBitT;
typedef upfHandleT        upfHdlPortMultiBitT;
typedef upfHandleT        upfHdlScopeT;
typedef upfHandleT        upfIsolationStrategyT;
typedef upfHandleT        upfLevelShifterStrategyT;
typedef upfHandleT        upfNamedRefT;
typedef upfHandleT        upfPowerStateTableT;
typedef upfHandleT        upfPowerStateTransitionT;
typedef upfHandleT        upfPowerSwitchStateT;
typedef upfHandleT        upfPowerSwitchT;
typedef upfHandleT        upfRepeaterStrategyT;
typedef upfHandleT        upfRetentionStrategyT;
typedef upfHandleT        upfSignalSenseT;
typedef upfHandleT        upfSupplyPortStateT;


//////////////////////////////////////
// Enumeration types
//////////////////////////////////////

// The classes in the information model will be identified in
//  HDL by the enumerated values defined in upfClassIdE
//  enumerated type :
typedef enum
 {
  //Abstract Class Id        //Class Name
  UPF_BASE,                    //upfBaseT
  UPF_BASE_HDL,                //upfBaseHdlT
  UPF_BASE_NAMED,              //upfBaseNamedT
  UPF_BASE_RELATIONSHIP,       //upfBaseRelationshipT
  UPF_BASE_UPF,                //upfBaseUpfT
  UPF_BOUNDARY_STRATEGY,       //upfBoundaryStrategyT
  UPF_EXTENT_CLASS,            //upfExtentClassT
  UPF_HDL_DECL,                //upfHdlDeclT
  UPF_HDL_NET_CLASS,           //upfHdlNetClassT
  UPF_HDL_PORT_CLASS,          //upfHdlPortClassT
  UPF_NET_CLASS,               //upfNetClassT
  UPF_NETWORK_CLASS,           //upfNetworkClassT
  UPF_PORT_CLASS,              //upfPortClassT
  UPF_STATE_CLASS,             //upfStateClassT
  UPF_STRATEGY,                //upfStrategyT

  //Concrete Class Id
  UPF_ACK_PORT,                //upfAckPortT
  UPF_ATTRIBUTE,               //upfAttributeT
  UPF_CELL,                    //upfCellT
  UPF_COMPOSITE_DOMAIN,        //upfCompositeDomainT
  UPF_EXPRESSION,              //upfExpressionT
  UPF_EXTENT,                  //upfExtentT
  UPF_HDL_MULTI_BIT_SLICE,     //upfHdlMultiBitSliceT
  UPF_HDL_NET_BIT,             //upfHdlNetBitT
  UPF_HDL_NET_MULTI_BIT,       //upfHdlNetMultiBitT
  UPF_HDL_PORT_BIT,            //upfHdlPortBitT
  UPF_HDL_PORT_MULTI_BIT,      //upfHdlPortMultiBitT
  UPF_HDL_SCOPE,               //upfHdlScopeT
  UPF_ISOLATION_STRATEGY,      //upfIsolationStrategyT
  UPF_LEVEL_SHIFTER_STRATEGY,  //upfLevelShifterStrategyT
  UPF_LOGIC_NET,               //upfLogicNetT
  UPF_LOGIC_PORT,              //upfLogicPortT
  UPF_NAMED_REF,               //upfNamedRefT
  UPF_POWER_DOMAIN,            //upfPowerDomainT
  UPF_POWER_STATE,             //upfPowerStateT
  UPF_POWER_STATE_TABLE,       //upfPowerStateTableT
  UPF_POWER_STATE_TRANSITION,  //upfPowerStateTransitionT
  UPF_POWER_SWITCH_STATE,      //upfPowerSwitchStateT
  UPF_POWER_SWITCH,            //upfPowerSwitchT
  UPF_PST_STATE,               //upfPstStateT
  UPF_REPEATER_STRATEGY,       //upfRepeaterStrategyT
  UPF_RETENTION_STRATEGY,      //upfRetentionStrategyT
  UPF_SIGNAL_SENSE,            //upfSignalSenseT
  UPF_SUPPLY_NET,              //upfSupplyNetT
  UPF_SUPPLY_PORT,             //upfSupplyPortT
  UPF_SUPPLY_PORT_STATE,       //upfSupplyPortStateT
  UPF_SUPPLY_SET,              //upfSupplySetT
  UPF_SUPPLY_TYPE,             //upfSupplyTypeT
  // Basic Property Types
  UPF_BOOLEAN,                 //upfBooleanT
  UPF_STRING,                  //upfStringT
  UPF_INTEGER,                 //upfIntegerT
  UPF_REAL                     //upfRealT
 } upfClassIdE;


// The following table provides the mapping between the
//   property names and property IDs in HDL type.
typedef enum
  {
// proptery id                         Class Name
  UPF_ACK_DELAY,                       //upfAckPortT
  UPF_ACK_PORTS,                       //upfPowerSwitchT
  UPF_APPLIES_TO,                      //upfBoundaryStrategyT
  UPF_ATTR_NAME,                       //upfAttributeT
  UPF_ATTR_VALUE,                      //upfAttributeT
  UPF_CELL_INFO,                       //upfBaseHdlT
  UPF_CELL_KIND,                       //upfCellT
  UPF_CELL_ORIGIN,                     //upfCellT
  UPF_CELLS,                           //upfExtentT
  UPF_CHILD_INSTANCES,                 //upfHdlScopeT
  UPF_CLAMP_VALUES,                    //upfIsolationStrategyT
  UPF_CONTROL_PORTS,                   //upfPowerSwitchT
  UPF_CONTROL_SIGNAL,                  //upfSignalSenseT
  UPF_CREATION_SCOPE,                  //upfBaseUpfT
  UPF_CURRENT_STATE,                   //upfPowerDomainT,
                                       //  upfCompositeDomainT,
                                       //  upfSupplySetT,
                                       //  upfPowerStateTableT
  UPF_CURRENT_VALUE,                   //upfExpressionT,
                                       //  upfSupplyNetT,
                                       //  upfSupplyPortT,
                                       //  upfLogicNetT,
                                       //  upfLogicPortT,
                                       //  upfAckPortT
  UPF_EFFECTIVE_EXTENTS,               //upfExtentClassT
  UPF_EQUIVALENT_SETS,                 //upfSupplySetT
  UPF_EXPR_OPERANDS,                   //upfExpressionT
  UPF_EXPR_STRING,                     //upfExpressionT
  UPF_EXTENTS,                         //upfBaseHdlT
  UPF_FANIN_CONN,                      //upfNetClassT
  UPF_FANOUT_CONN,                     //upfNetClassT
  UPF_FILE,                            //upfBaseT
  UPF_FROM_STATES,                     //upfPowerStateTransitionT
  UPF_FUNCTIONS,                       //upfSupplySetT
  UPF_HDL_ATTRIBUTES,                  //upfBaseHdlT
  UPF_HDL_CELL_KIND,                   //upfCellT
  UPF_HDL_ELEMENT,                     //upfExtentT
  UPF_HDL_IMPLEMENTATION,              //upfNetworkClassT
```

```
UPF_HDL_ITEMS,                   //upfHdlScopeT
UPF_HDL_PORTS,                   //upfHdlScopeT
UPF_HDL_WIDTH,                   //upfHdlNetMultiBitT,
                                 //  upfHdlPortMultiBitT
UPF_HICONN,                      //upfPortClassT
UPF_INPUT_SUPPLY_PORT,           //upfPowerSwitchStateT
UPF_INPUT_SUPPLY_PORTS,          //upfPowerSwitchT
UPF_IS_ACTIVE,                   //upfPowerStateT,
                                 //  upfPstStateT
UPF_IS_DIFF_SUPPLY_ONLY,         //upfIsolationStrategyT
UPF_IS_FORCE_ISOLATION,          //upfIsolationStrategyT
UPF_IS_FORCE_SHIFT,              //upfLevelShifterStrategyT
UPF_IS_ILLEGAL,                  //upfStateClassT
UPF_IS_NO_ISOLATION,             //upfIsolationStrategyT
UPF_IS_NO_RETENTION,             //upfRetentionStrategyT
UPF_IS_NO_SHIFT,                 //upfLevelShifterStrategyT
UPF_IS_USE_EQUIVALENCE,          //upfBoundaryStrategyT
UPF_IS_USE_RETENTION_AS_PRIMARY, //upfRetentionStrategyT
UPF_ISOLATION_CONTROLS,          //upfIsolationStrategyT
UPF_ISOLATION_STRATEGIES,        //upfPowerDomainT
UPF_ITEMS,                       //upfHdlScopeT
UPF_LEVEL_SHIFT_RULE,            //upfLevelShifterStrategyT
UPF_LEVEL_SHIFTER_STRATEGIES,    //upfPowerDomainT
UPF_LINE,                        //upfBaseT
UPF_LOCATION,                    //upfBoundaryStrategyT
UPF_LOCONN,                      //upfPortClassT
UPF_LOGIC_EXPR,                  //upfPowerStateT
UPF_LOGIC_REFS,                  //upfStrategyT
UPF_LOWER_BOUNDARY,              //upfPowerDomainT
UPF_LSB,                         //upfHdlMultiBitSliceT
UPF_MODEL_NAME,                  //upfCellT
UPF_MSB,                         //upfHdlMultiBitSliceT
UPF_NAME,                        //upfBaseT
UPF_NAME_PREFIX,                 //upfBoundaryStrategyT
UPF_NAME_SUFFIX,                 //upfBoundaryStrategyT
UPF_NETWORK_ATTRIBUTES,          //upfNetworkClassT
UPF_NEXT_EXTENT,                 //upfExtentT
UPF_NORMALIZED_BITS,             //upfHdlNetMultiBitT,
                                 //  upfHdlPortMultiBitT
UPF_NORMALIZED_IDX,              //upfHdlNetBitT,
                                 //  upfHdlPortBitT
UPF_OBJECT,                      //upfExtentT
UPF_OUTPUT_SUPPLY_PORT,          //upfPowerSwitchT
UPF_PARENT,                      //upfBaseT
UPF_PD_STATE_TRANSITIONS,        //upfPowerDomainT,
                                 //  upfCompositeDomainT
UPF_PD_STATES,                   //upfPowerDomainT,
                                 //  upfCompositeDomainT
UPF_PORT_DIR,                    //upfHdlPortClassT,
                                 //  upfPortClassT
UPF_PST_HEADER,                  //upfPowerStateTableT
UPF_PST_STATES,                  //upfPowerStateTableT
UPF_REF_KIND,                    //upfNamedRefT
UPF_REF_OBJECT,                  //upfNamedRefT
UPF_REPEATER_STRATEGIES,         //upfPowerDomainT
UPF_RESOLVE_TYPE,                //upfSupplyNetT
UPF_RESTORE_CONDITION,           //upfRetentionStrategyT
UPF_RESTORE_SIGNAL,              //upfRetentionStrategyT
UPF_RETENTION_CONDITION,         //upfRetentionStrategyT
UPF_RETENTION_PARAMETERS,        //upfRetentionStrategyT
UPF_RETENTION_STRATEGIES,        //upfPowerDomainT
UPF_ROOT_DRIVER,                 //upfNetworkClassT
UPF_SAVE_CONDITION,              //upfRetentionStrategyT
```

```
  UPF_SAVE_SIGNAL,                 //upfRetentionStrategyT
  UPF_SIGNAL_SENSITIVITY,          //upfSignalSenseT
  UPF_SIMSTATE,                    //upfPowerStateT
  UPF_SINK_FILTER,                 //upfBoundaryStrategyT
  UPF_SLICE_BITS,                  //upfHdlMultiBitSliceT
  UPF_SMALLEST_ATOMIC_SLICE        //upfHdlNetBitT,
                                   //  upfHdlPortBitT
  UPF_SOURCE_EXTENTS,              //upfCellT
  UPF_SOURCE_FILTER,               //upfBoundaryStrategyT
  UPF_SP_STATES,                   //upfSupplyPortT
  UPF_SS_STATES,                   //upfSupplySetT
  UPF_SS_TRANSITIONS,              //upfSupplySetT
  UPF_STATE,                       //upfSupplyTypeT
  UPF_SUBDOMAINS,                  //upfCompositeDomainT
  UPF_SUPPLY_EXPR,                 //upfPowerStateT
  UPF_SUPPLY_SET_HANDLES,          //upfExtentClassT,
                                   //  upfCompositeDomainT
  UPF_SUPPLY_STATE,                //upfSupplyPortStateT
  UPF_SUPPLY_STATES,               //upfPstStateT
  UPF_SW_STATES,                   //upfPowerSwitchT
  UPF_SWITCH_EXPR,                 //upfPowerSwitchStateT
  UPF_SWITCH_OUTPUT_STATE,         //upfPowerSwitchStateT
  UPF_THRESHOLD_VALUE,             //upfLevelShifterStrategyT
  UPF_TO_STATES,                   //upfPowerStateTransitionT
  UPF_UPPER_BOUNDARY,              //upfPowerDomainT
  UPF_USER_CLAMP_VALUES,           //upfIsolationStrategyT
  UPF_VOLTAGE,                     //upfSupplyTypeT
  UPF_VOLT_KIND,                   //upfSupplyPortStateT
  UPF_VOLT_MAX,                    //upfSupplyPortStateT
  UPF_VOLT_MIN,                    //upfSupplyPortStateT
  UPF_VOLT_NOM                     //upfSupplyPortStateT
  } upfPropertyIdE;

typedef enum
  {UPF_FILTER_UNDEF,      // Undefined
   UPF_FILTER_INPUTS,     // -applies_to inputs
   UPF_FILTER_OUTPUTS,    // -applies_to outputs
   UPF_FILTER_BOTH        // -applies_to both
   }upfAppliesToFilterE;

typedef enum
 {UPF_CELL_NONE,          // not a cell
  UPF_CELL_ISOLATION,     // represents isolation cell
  UPF_CELL_LEVEL_SHIFTER, // represents level shifter cell
  UPF_CELL_ISO_LS_COMBO,  // represents isolation and
                          //   level shifter combo cell
  UPF_CELL_RETENTION,     // represents retention cell
  UPF_CELL_SWITCH,        // represents a switch cell
  UPF_CELL_REPEATER,      // represents a repeater or buffer cell
  UPF_CELL_CORRUPT,       // represents any standard cell
                          //  which can get corrupted
  UPF_CELL_MACRO          // represents a macro cell
                          //  or power model
 }upfCellKindE;

typedef enum
 {UPF_ORIGIN_UNKNOWN,      // When cell origin is not known
  UPF_ORIGIN_DESIGN,       // When cell is present in
                           //  design itself
  UPF_ORIGIN_INSERTED,     // When cell is inserted by UPF after
                           //  application of strategy
                           //  (e.g. isolation ) and using
                           //  default model.
  UPF_ORIGIN_INSERTED_MAP, // When cell is inserted by UPF after
                           //  application of strategy and using
                           //  user specified model via map_*
                           //  and use_interface_cell commands
  UPF_ORIGIN_INFERRED      // When cell is inferred by UPF at
                           //  RTL.  This information will only
                           //  be present on cells which are set
                           //  on HDL Port or Nets group
                           //  of objects.
 }upfCellOriginE;

typedef enum
 {UPF_HDLCELL_NONE,        // not a cell
  UPF_HDLCELL_COMB,        // represents a combinatorial logic
  UPF_HDLCELL_FLOP,        // represents a flip flop
  UPF_HDLCELL_LATCH,       // represents a latch
  UPF_HDLCELL_MEM          // represents a memory like ram, etc.
 }upfHdlCellKindE;

typedef enum
 {UPF_CLAMP_UNDEF,         // Undefined
  UPF_CLAMP_ZERO,          // -*_clamp 0
  UPF_CLAMP_ONE,           // -*_clamp 1
  UPF_CLAMP_ZEE,           // -*_clamp Z
  UPF_CLAMP_LATCH,         // -*_clamp latch
  UPF_CLAMP_ANY,           // -*_clamp any
  UPF_CLAMP_USER_VALUE     // -*_clamp value
 }upfIsolationClampE;

typedef enum
 {UPF_LS_LOW_TO_HIGH,      // -rule low_to_high
  UPF_LS_HIGH_TO_LOW,      // -rule high_to_low
  UPF_LS_BOTH              // -rule both
 }upfLevelShifterRuleE;

typedef enum
 {SELF,                    // -location self
  OTHER,                   // -location other
  PARENT,                  // -location parent
  AUTOMATIC,               // -location automatic
  FANOUT                   // -location fanout
 }upfLocationE;

typedef enum
 {
  UPF_REF_POWER,              // power function of supply set
  UPF_REF_GROUND,             // ground function of supply set
  UPF_REF_PWELL,              // pwell function of supply set
  UPF_REF_NWELL,              // nwell function of supply set
  UPF_REF_DEEPPWELL,          // deeppwell function of supply set
  UPF_REF_DEEPNWELL,          // deepnwell function of supply set
  UPF_REF_ISO_SIGNAL,         // reference to isolation control
                              //  signal in set_isolation
  UPF_REF_SAVE_SIGNAL,        // to save_signal in set_retention
  UPF_REF_RESTORE_SIGNAL,     // reference to restore_signal
                              //  in set_retention
  UPF_REF_GENERIC_CLOCK,      // reference to UPF_GENERIC_CLOCK
                              //  in set_retention
  UPF_REF_GENERIC_DATA,       // reference to UPF_GENERIC_DATA
                              //  in set_retention
  UPF_REF_GENERIC_ASYNC_LOAD, // reference to
                              //  UPF_GENERIC_ASYNC_LOAD
                              //  in set_retention
  UPF_REF_GENERIC_OUTPUT,      // reference to UPF_GENERIC_OUTPUT
                               //  in set_retention
  UPF_REF_USER_DEFINED         // some user defined ref handle
 }upfNamedRefKindE;

typedef enum
 {UPF_DIR_UNDEF,               // Undefined
  UPF_DIR_IN,                  // -direction in
  UPF_DIR_OUT,                 // -direction out
  UPF_DIR_INOUT                // -direction inout
 }upfPortDirE;

typedef enum
 {UNRESOLVED,                  // -resolve unresolved
  ONE_HOT,                     // -resolve one_hot
  PARALLEL,                    // -resolve parallel
  PARALLEL_ONE_HOT             // -resolve parallel_one_hot
 }upfResolveE;

typedef enum
 //The enumerated literals here map directly to values specified
 //  in -parameters option of set_retention command
 {RET_SUP_COR,
  NO_RET_SUP_COR,
  SAV_RES_COR,
  NO_SAV_RES_COR
 }upfRetentionParamE;

typedef enum
 {UPF_SENSE_HIGH,              // high
  UPF_SENSE_LOW,               // low
  UPF_SENSE_POSEDGE,           // posedge
  UPF_SENSE_NEGEDGE            // negedge
 }upfSignalSenseKindE;

typedef enum
 {CORRUPT                        // -simstate CORRUPT
                                 //   or -simstate NOT_NORMAL
  CORRUPT_ON_ACTIVITY,           // -simstate CORRUPT_ON_ACTIVITY
                                 //   or -simstate NOT_NORMAL
  CORRUPT_ON_CHANGE,             // -simstate CORRUPT_ON_CHANGE
                                 //   or -simstate NOT_NORMAL
  CORRUPT_STATE_ON_ACTIVITY,     // -simstate
                                 //  CORRUPT_STATE_ON_ACTIVITY
                                 //   or -simstate NOT_NORMAL
  CORRUPT_STATE_ON_CHANGE,       // -simstate
                                 //  CORRUPT_STATE_ON_CHANGE
                                 //   or -simstate NOT_NORMAL
  NORMAL,                        // -simstate NORMAL
 }upfSimstateE;

typedef enum
 {NOM,                         // when only nominal value is
                               //  specified in add_port_state
                               //  command
  DOUBLET,                     // when a doublet is specified in
                               //  add_port_state command
  TRIPLET                      // when a triplet is specified in
                               //  add_port_state command
 }upfVoltKindE;
```

```
//////////////////////////////////////
// HDL Access Functions
//////////////////////////////////////


//////////////////////////////////////
// Purpose: Get a handle to a given object from the pathname
function automatic upfHandleT upf_get_handle_by_name
  (upfStringT pathname,            // A string representing
                                   //  handle ID for an object
   upfHandleT relative_to = null); // An optional handle to the
                                   //  object from which the
                                   //  relative pathname is given

  // It shall be an error if :
  //   1) pathname is not a valid handle ID
  //   2) relative_to is not a valid UPF Handle

  // Returns the handle to the specified property or null
  //  if not found
endfunction:upf_get_handle_by_name

//////////////////////////////////////
// Purpose: Query propertys on a given object
function automatic upfHandleT upf_query_object_properties
  (upfHandleT      object_handle,  // UPF Handle of the given
                                   //  object
   upfPropertyIdE attr);           // The enumerated value
                                   //  corresponding to the given
                                   //  property on the object

  // It shall be an error if :
  //   1) object_handle is not a valid object
  //   2) attr is not a valid property on the given object
  //   3) attr is not a valid value defined in Table 21

  // Returns the handle to the specified property
  //  or null if not found
endfunction:upf_query_object_properties

//////////////////////////////////////
// Purpose: Get the next handle from the iterator
function automatic upfHandleT upf_iter_get_next
  (upfHandleT iter_handle);        // Handle to the iterator

  // It shall be an error if iter_handle is not
  //  a valid iterator handle.

  // Returns the handle to the next element
  //  in the iterator or 0 if no element is present.
endfunction:upf_iter_get_next

//////////////////////////////////////
// Immediate Read Access Functions
//////////////////////////////////////

// The following table provides the mapping of basic
// properties and the read access routine to get the value.
/////////////////////////////////////////////////
// Return Type           function names
//----------            -------------
// upfStringT            upf_get_value_str()
// upfIntegerT           upf_get_value_int()
// upfBooleanT           upf_get_value_int()
// All Enumerated Types  upf_get_value_int()
// upfRealT              upf_get_value_real()
/////////////////////////////////////////////////

//////////////////////////////////////
// Purpose: Get the string value from property handle
function automatic  upfStringT upf_get_value_str
  (upfHandleT attr);        // Handle to the property

  // Returns string value of the given
  //  property handle or NULL if error.
endfunction:upf_get_value_str

//////////////////////////////////////
// Purpose: Get the integer value from property handle
function automatic  upfIntegerT upf_get_value_int
  (upfHandleT attr);        // Handle to the property

  // Returns integer value of the given property handle.
endfunction:upf_get_value_int

//////////////////////////////////////
// Purpose: Get the real value from property handle
function automatic  upfRealT upf_get_value_real
  (upfHandleT attr);        // Handle to the property

  // Returns real value of the given property handle.
endfunction:upf_get_value_real

//////////////////////////////////////
// Purpose: Get the value of supply net
//          type from the property handle
function automatic upfSupplyTypeT  upf_get_label_upfSupplyTypeT
  (upfHandleT attr);        // Handle to the property

  // It shall be an error if property is not a valid
  //  property handle of upfSupplyTypeT

  // Returns upfSupplyTypeT value of the given property handle.
endfunction:upf_get_label_upfSupplyTypeT

//////////////////////////////////////
// Purpose: Get the supply value of a Net
function automatic upfSupplyTypeT get_supply_value
  (upfStringT name);        // A string representing
                            // pathname of supply net

  // It shall be an error if name is not a UPF created
  //  supply net/port or an HDL object of upfSupplyTypeT
  //  or upfSupplyObjT.

endfunction:get_supply_value

//////////////////////////////////////
// Purpose: Get the supply voltage of a net
function automatic upfRealT get_supply_voltage
  (upfSupplyTypeT arg);     // An HDL object of upfSupplyTypeT

  // the voltage in the supply_net_type struct is in uV
endfunction:get_supply_voltage
```

```
//////////////////////////////////////
// Purpose: Get the supply FULL_ON state
function automatic upfBooleanT get_supply_on_state
  (supply_net_type arg);    //-  An object of upfSupplyTypeT
  upfBooleanT     state = 0;

  // The get_supply_on_state returns the on/off state of the
  //  specified supply net. It returns 1 when supply net state is
  //  FULL_ON or PARTIAL_ON and set_partial_on_translation
  //  FULL_ON is called. It returns 0 in other cases.

endfunction:get_supply_on_state

//////////////////////////////////////
// Purpose: Get the state from a supply net
function automatic upfSupplyStateE get_supply_state
  (upfSupplyTypeT arg);     //  An object of upfSupplyTypeT

  // Return Value: Returns state of the supply net
endfunction:get_supply_state

////////////////////////////////////////////////////////////
// Immediate Write Access Functions
////////////////////////////////////////////////////////////
// The following table provides the mapping between basic
// property types and corresponding hdl function to write
// the value
////////////////////////////////////////////////////////////
// Type Name
//----------             -------------
// upfStringT            /
// upfIntegerT           upf_set_value_int()
// upfBooleanT           upf_set_value_int()
// All Enumerated Types  upf_set_value_int()
// upfRealT              /
// upfSupplyTypeT        upf_set_value_upfSupplyTypeT()
////////////////////////////////////////////////////////////

//////////////////////////////////////
// Purpose: Sets the integer value on the dynamic property
function automatic upfBooleanT upf_set_value_int
  (upfHandleT attr,         // Handle to the property
   upfIntegerT value);      // value

  // It shall be an error when:
  //   1) attr is a null handle
  //        if (attr = null) return '0';
  //   2) attr is not a valid dynamic property of upfIntegerT
  //        type.

  // Return Value: Returns 1 on success, 0 on failure
endfunction:upf_set_value_int

//////////////////////////////////////
// Purpose: Sets the upfSupplyTypeT value on the
//            dynamic property
function automatic upfBooleanT upf_set_value_upfSupplyTypeT
  (upfHandleT attr,         // Handle to the property
   supply_net_type value);  // value of type upfSupplyTypeT

  // It shall be an error when:
  //   1) attr is a null handle
  //        if (attr = null) return '0';
  //   2) attr is not a valid dynamic property of
  //        upfSupplyTypeT type.

  // Return Value: Returns 1 on success, 0 on failure
endfunction:upf_set_value_upfSupplyTypeT

//////////////////////////////////////
// Purpose: Change state of supply to FULL_ON and
//            specify voltage
// Return Value: Returns 1 on success, 0 on failure
function automatic upfBooleanT supply_on
  (upfStringT supply_name,         // A string representing pathname
                                //   of the root supply
   upfRealT value = 1.0);       // Voltage value in real

  // It shall be an error if :
  //   1) supply_name is not a UPF created supply net/port or
  //        an HDL object of upfSupplyTypeT or upfSupplyObjT
  //   2) supply_name already has a driver

endfunction:supply_on

//////////////////////////////////////
// Purpose: Change state of supply to OFF
// Return Value: Returns 1 on success, 0 on failure
function automatic upfBooleanT supply_off
  (upfStringT supply_name);  // A hierarchical path ID for a
                             //  root supply.

  // It shall be an error if :
  //   1) supply_name is not a UPF created supply net/port or
  //      an HDL object of upfSupplyTypeT or upfSupplyObjT
  //   2) supply_name already has a driver

endfunction:supply_off

//////////////////////////////////////
// Purpose: Change state of supply to PARTIAL_ON and
//          specify voltage
// Return Value: Returns 1 on success, 0 on failure
function automatic upfBooleanT supply_partial_on
  (upfStringT supply_name,        // A string representing pathname
                               //  of the root supply
   upfRealT   value = 1.0);    // Voltage value in real

  // It shall be an error if :
  //   1) supply_name is not a UPF created supply net/port or
  //      an HDL object of upfSupplyTypeT or upfSupplyObjT
  //   2) supply_name already has a driver

endfunction:supply_partial_on

//////////////////////////////////////
// Purpose: Assign the specified supply state to this object.
// Return Value: Returns 1 on success, 0 on failure
function automatic upfBooleanT set_supply_state
  (upfStringT object_name,  // the hierarchical path ID of a supply port,
                            //   net, or supply set function
   upfSupplyStateE supply_state);// one of OFF, PARTIAL_ON, FULL_ON,
                            //   UNDETERMINED

  // It shall be an error if :
  //   1) object_name is not a valid supply port, net, or
  //      supply set function name
  //   2) different values are assigned to the supply object
  //      in the same cycle
endfunction:set_supply_state

//////////////////////////////////////
// Purpose: Make the specified power state active for this object.
function automatic upfBooleanT set_power_state_by_handle
  (upfHandleT object,      // Handle to the UPF object
   upfHandleT power_state);// Handle of the power state present
                           //   on the object

  // It shall be an error if :
  //   1) object_name is not a valid name of a supply set, power
  //      domain, composite domain, group, model, or instance
  //        if (object = null) return '0';
  //   2) power_state is not the name of a power state of the
  //      specified object
  //      power state present on object
  //        if (power_state = null) return '0'
  //   3) different power states that are not related by refinement
  //      are made active for this object in the same cycle
  //        ...
  // Return Value: Returns 1 on success, 0 on failure
endfunction:set_power_state_by_handle

//////////////////////////////////////
// Purpose: Make the specified power state active for this object.
// Return Value: Returns 1 on success, 0 on failure
function automatic upfBooleanT set_power_state
  (upfStringT object_name, // Hierarchical path ID of object
                           //   having power state
   upfStringT power_state);// Relative path ID of power state
                           //   with respect to the object

  // It shall be an error if :
  //   1) object_name is not a valid name of a supply set, power
  //      domain, composite domain, group, model, or instance
  //   2) power_state is not the name of a power state of the
  //      specified object
  //   3) different power states that are not related by refinement
  //      are made active for this object in the same cycle

endfunction:set_power_state

//////////////////////////////////////
// Continuous Access Functions
//////////////////////////////////////

//////////////////////////////////////
// Purpose: Query properties on a given object
function automatic upfBooleanT upf_create_object_mirror
  (upfStringT src,   // A string representing hierarchical
                     //   path ID of the source object whose
                     //   value will be continuously monitored
   upfStringT dst);  // A string representing hierarchical
                     //   path ID of the destination object on
                     //   which the value will be transferred
                     //   from source object

  // It shall be an error if :
  //   1) src and dst represent objects that do not
  //        have Native HDL Representation (see 11.2.2.3)
  //   2) dst does not support write access
  //   3) dst already has a driver associated with it
  //   4) upf_create_object_mirror is called multiple
  //        times on same src/dst pair

  // Return Value: Returns 1 on success, 0 on failure
endfunction:upf_create_object_mirror

//////////////////////////////////////
// Utility Functions
//////////////////////////////////////

//////////////////////////////////////
// Purpose: Get the class id of the given object handle
function automatic upfClassIdE upf_query_object_type
  (upfHandleT handle);  // Handle to the object or property

  // It shall be an error if handle is not a valid
  //  object/property handle.

  // Return Value: Returns enumerated value representing class
  //               of the given object or property
endfunction:upf_query_object_type

//////////////////////////////////////
// Purpose: Check if object belongs to particular class
function automatic upfBooleanT upf_object_in_class
  (upfHandleT  handle,    // Handle to the object or property
   upfClassIdE class_id); // The enumerated value representing
                          //  the class of the object.

  // It shall be an error if :
  //   1) handle is not a valid object/property handle
  //   2) class_id is not a valid enumerated value defined
  //       in upfClassIdE.

  // Return Value: Returns 1 when handle belongs to class
  //                and 0 otherwise.
endfunction:upf_object_in_class


//////////////////////////////////////
// Purpose: Get the pathname of the given handle
function automatic upfStringT upf_query_object_pathname
  (upfHandleT object,              // Handle to a valid object
   upfHandleT relative_to = null); // An optional handle to the
                                   //  object from which the
                                   //  relative pathname is
                                   //  required

  // It shall be an error if :
  //   1) object is not a valid UPF Handle
  //   2) relative_to is not a valid UPF Handle
  //   3) relative_to is not in the ancestor hierarchy of
  //        the object

  // Return Value: Returns the hierarchical pathname
  //                of the given object
endfunction:upf_query_object_pathname

//////////////////////////////////////
// Purpose: Converts the voltage value in volts represented as
//          real value to micro volts represented as integer.
```

```
  function automatic
    upfIntegerT upf_convert_real_volts_to_int_uvolts
      (upfRealT volt);    // Voltage value in volts represented
                          //  as a real number

  endfunction:upf_convert_real_volts_to_int_uvolts

  //////////////////////////////////////
  // Purpose: Converts the voltage in micro volts represented
  //          as integer to volts represented as real value
  function automatic upfRealT upf_convert_int_uvolts_to_real_volts
    (upfIntegerT uvolt);  // Voltage value in micro volts
                          //   represented as a integer number

  endfunction:upf_convert_int_uvolts_to_real_volts

  //////////////////////////////////////
  // Pre-defined supply net resolution functions
  //////////////////////////////////////

  function automatic upfSupplyTypeT one_hot
    (upfSupplyTypeVectorT sources);

  endfunction:one_hot

  function automatic upfSupplyTypeT parallel
    (upfSupplyTypeVectorT sources);

  endfunction:parallel

  function automatic upfSupplyTypeT parallel_one_hot
    (upfSupplyTypeVectorT sources);


  endfunction:parallel_one_hot

endpackage:UPF
```

11.2.5 VHDL UPF package

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Use IEEE.numeric_bit.all;

------------------------------------------------------------------
-- Package Declaration
------------------------------------------------------------------

package UPF is

  --------------------------------------
  -- Basic Types
  --------------------------------------

  subtype upfStringT   is STRING;
  subtype upfIntegerT  is INTEGER;
  subtype upfBooleanT  is BIT;
  subtype upfRealT     is REAL;
  subtype upfHandleT   is INTEGER;
  subtype upfIteratorT is INTEGER;
```

```
type upfSupplyStateE is (
  OFF,          -- -off_state or reference to OFF for supply net/port
                --   state
  FULL_ON,      -- -on_state or reference to FULL_ON for supply
                --   net/port state
  PARTIAL_ON,  -- -on_partial_state or reference to PARTIAL_ON for
                --   supply net/port state
  UNDETERMINED -- -error_state or reference to UNDETERMINED for
                --   supply net/port state
);

type upfSupplyTypeT is record
  state         : upfSupplyStateE;
  voltage       : upfIntegerT; -- Voltage in microvolts
end record;

-- Legacy support: backward compatibility
subtype state           is upfSupplyStateE;
subtype supply_net_type is upfSupplyTypeT;

--------------------------------------
-- Native HDL Representation
--------------------------------------

type upfPowerStateObjT is record
  handle        : upfHandleT;
  is_active     : upfBooleanT;
end record;

type upfPdSsObjT is record
  handle        : upfHandleT;
  current_state : upfPowerStateObjT;
end record;

type upfBooleanObjT is record
  handle        : upfHandleT;
  current_value : upfBooleanT;
end record;

type upfSupplyObjT is record
  handle        : upfHandleT;
  current_value : upfSupplyTypeT;
end record;

--------------------------------------
-- HDL Type Mapping
--------------------------------------

subtype upfCompositeDomainT      is upfPdSsObjT;
subtype upfPowerDomainT          is upfPdSsObjT;
subtype upfSupplySetT            is upfPdSsObjT;

subtype upfPstStateT             is upfPowerStateObjT;
subtype upfPowerStateT           is upfPowerStateObjT;

subtype upfAckPortT              is upfBooleanObjT;
subtype upfExpressionT           is upfBooleanObjT;
subtype upfLogicNetT             is upfBooleanObjT;
subtype upfLogicPortT            is upfBooleanObjT;

subtype upfSupplyNetT            is upfSupplyObjT;
subtype upfSupplyPortT           is upfSupplyObjT;
subtype upfAttributeT            is upfHandleT;
subtype upfCellT                 is upfHandleT;
subtype upfExtentT               is upfHandleT;
subtype upfHdlMultiBitSliceT     is upfHandleT;
subtype upfHdlNetBitT            is upfHandleT;
subtype upfHdlNetMultiBitT       is upfHandleT;
subtype upfHdlPortBitT           is upfHandleT;
subtype upfHdlPortMultiBitT      is upfHandleT;
subtype upfHdlScopeT             is upfHandleT;
subtype upfIsolationStrategyT    is upfHandleT;
subtype upfLevelShifterStrategyT is upfHandleT;
subtype upfNamedRefT             is upfHandleT;
subtype upfPowerStateTableT      is upfHandleT;
subtype upfPowerStateTransitionT is upfHandleT;
subtype upfPowerSwitchStateT     is upfHandleT;
subtype upfPowerSwitchT          is upfHandleT;
subtype upfRepeaterStrategyT     is upfHandleT;
subtype upfRetentionStrategyT    is upfHandleT;
subtype upfSignalSenseT          is upfHandleT;
subtype upfSupplyPortStateT      is upfHandleT;

--------------------------------------
-- Enumerations
--------------------------------------

-- The classes in the information model will be identified in HDL by
-- the enumerated values defined in upfClassIdE enumerated type :
type upfClassIdE is (
  --Abstract Class Id               Class Name
  UPF_BASE,                         --upfBaseT
  UPF_BASE_HDL,                     --upfBaseHdlT
  UPF_BASE_NAMED,                   --upfBaseNamedT
  UPF_BASE_RELATIONSHIP,            --upfBaseRelationshipT
  UPF_BASE_UPF,                     --upfBaseUpfT
  UPF_BOUNDARY_STRATEGY,            --upfBoundaryStrategyT
  UPF_EXTENT_CLASS,                 --upfExtentClassT
  UPF_HDL_DECL,                     --upfHdlDeclT
  UPF_HDL_NET_CLASS,                --upfHdlNetClassT
  UPF_HDL_PORT_CLASS,               --upfHdlPortClassT
  UPF_NET_CLASS,                    --upfNetClassT
  UPF_NETWORK_CLASS,                --upfNetworkClassT
  UPF_PORT_CLASS,                   --upfPortClassT
  UPF_STATE_CLASS,                  --upfStateClassT
  UPF_STRATEGY,                     --upfStrategyT

  --Concrete Class Id               Class Name
  UPF_ACK_PORT,                     --upfAckPortT
  UPF_ATTRIBUTE,                    --upfAttributeT
  UPF_CELL,                         --upfCellT
  UPF_COMPOSITE_DOMAIN,             --upfCompositeDomainT
  UPF_EXPRESSION,                   --upfExpressionT
  UPF_EXTENT,                       --upfExtentT
  UPF_HDL_MULTI_BIT_SLICE,          --upfHdlMultiBitSliceT
  UPF_HDL_NET_BIT,                  --upfHdlNetBitT
  UPF_HDL_NET_MULTI_BIT,            --upfHdlNetMultiBitT
  UPF_HDL_PORT_BIT,                 --upfHdlPortBitT
  UPF_HDL_PORT_MULTI_BIT,           --upfHdlPortMultiBitT
  UPF_HDL_SCOPE,                    --upfHdlScopeT
  UPF_ISOLATION_STRATEGY,           --upfIsolationStrategyT
  UPF_LEVEL_SHIFTER_STRATEGY,       --upfLevelShifterStrategyT
  UPF_LOGIC_NET,                    --upfLogicNetT
  UPF_LOGIC_PORT,                   --upfLogicPortT
  UPF_NAMED_REF,                    --upfNamedRefT
  UPF_POWER_DOMAIN,                --upfPowerDomainT
  UPF_POWER_STATE,                 --upfPowerStateT
  UPF_POWER_STATE_TABLE,           --upfPowerStateTableT
  UPF_POWER_STATE_TRANSITION,      --upfPowerStateTransitionT
  UPF_POWER_SWITCH_STATE,          --upfPowerSwitchStateT
  UPF_POWER_SWITCH,                --upfPowerSwitchT
  UPF_PST_STATE,                   --upfPstStateT
  UPF_REPEATER_STRATEGY,           --upfRepeaterStrategyT
  UPF_RETENTION_STRATEGY,          --upfRetentionStrategyT
  UPF_SIGNAL_SENSE,                --upfSignalSenseT
  UPF_SUPPLY_NET,                  --upfSupplyNetT
  UPF_SUPPLY_PORT,                 --upfSupplyPortT
  UPF_SUPPLY_PORT_STATE,           --upfSupplyPortStateT
  UPF_SUPPLY_SET,                  --upfSupplySetT
  UPF_SUPPLY_TYPE,                 --upfSupplyTypeT
  -- Basic Property types
  UPF_BOOLEAN,                     --upfBooleanT
  UPF_STRING,                      --upfStringT
  UPF_INTEGER,                     --upfIntegerT
  UPF_REAL                         --upfRealT
);

-- The following table provides the mapping between the property
-- names and property IDs in HDL type.
type upfPropertyIdE is (
  --Property Id                    Class Name
  UPF_ACK_DELAY,                   --upfAckPortT
  UPF_ACK_PORTS,                   --upfPowerSwitchT
  UPF_APPLIES_TO,                  --upfBoundaryStrategyT
  UPF_ATTR_NAME,                   --upfAttributeT
  UPF_ATTR_VALUE,                  --upfAttributeT
  UPF_CELL_INFO,                   --upfBaseHdlT
  UPF_CELL_KIND,                   --upfCellT
  UPF_CELL_ORIGIN,                 --upfCellT
  UPF_CELLS,                       --upfExtentT
  UPF_CHILD_INSTANCES,             --upfHdlScopeT
  UPF_CLAMP_VALUES,                --upfIsolationStrategyT
  UPF_CONTROL_PORTS,               --upfPowerSwitchT
  UPF_CONTROL_SIGNAL,              --upfSignalSenseT
  UPF_CREATION_SCOPE,              --upfBaseUpfT
  UPF_CURRENT_STATE,               --upfPowerDomainT,
                                   --  upfCompositeDomainT,
                                   --  upfSupplySetT,
                                   --  upfPowerStateTableT
  UPF_CURRENT_VALUE,               --upfExpressionT, upfSupplyNetT,
                                   --  upfSupplyPortT, upfLogicNetT,
                                   --  upfLogicPortT, upfAckPortT
  UPF_EFFECTIVE_EXTENTS,           --upfExtentClassT
  UPF_EQUIVALENT_SETS,             --upfSupplySetT
  UPF_EXPR_OPERANDS,               --upfExpressionT
  UPF_EXPR_STRING,                 --upfExpressionT
  UPF_EXTENTS,                     --upfBaseHdlT
  UPF_FANIN_CONN,                  --upfNetClassT
  UPF_FANOUT_CONN,                 --upfNetClassT
  UPF_FILE,                        --upfBaseT
  UPF_FROM_STATES,                 --upfPowerStateTransitionT
  UPF_FUNCTIONS,                   --upfSupplySetT
  UPF_HDL_ATTRIBUTES,              --upfBaseHdlT
  UPF_HDL_CELL_KIND,               --upfCellT
  UPF_HDL_ELEMENT,                 --upfExtentT
  UPF_HDL_IMPLEMENTATION,          --upfNetworkClassT
  UPF_HDL_ITEMS,                   --upfHdlScopeT
  UPF_HDL_PORTS,                   --upfHdlScopeT
```

```
UPF_HDL_WIDTH,                  --upfHdlNetMultiBitT,
                                --  upfHdlPortMultiBitT
UPF_HICONN,                     --upfPortClassT
UPF_INPUT_SUPPLY_PORT,          --upfPowerSwitchStateT
UPF_INPUT_SUPPLY_PORTS,         --upfPowerSwitchT
UPF_IS_ACTIVE,                  --upfPowerStateT, upfPstStateT
UPF_IS_DIFF_SUPPLY_ONLY,        --upfIsolationStrategyT
UPF_IS_FORCE_ISOLATION,         --upfIsolationStrategyT
UPF_IS_FORCE_SHIFT,             --upfLevelShifterStrategyT
UPF_IS_ILLEGAL,                 --upfStateClassT
UPF_IS_NO_ISOLATION,            --upfIsolationStrategyT
UPF_IS_NO_RETENTION,            --upfRetentionStrategyT
UPF_IS_NO_SHIFT,                --upfLevelShifterStrategyT
UPF_IS_USE_EQUIVALENCE,         --upfBoundaryStrategyT
UPF_IS_USE_RETENTION_AS_PRIMARY,--upfRetentionStrategyT
UPF_ISOLATION_CONTROLS,         --upfIsolationStrategyT
UPF_ISOLATION_STRATEGIES,       --upfPowerDomainT
UPF_ITEMS,                      --upfHdlScopeT
UPF_LEVEL_SHIFT_RULE,           --upfLevelShifterStrategyT
UPF_LEVEL_SHIFTER_STRATEGIES,   --upfPowerDomainT
UPF_LINE,                       --upfBaseT
UPF_LOCATION,                   --upfBoundaryStrategyT
UPF_LOCONN,                     --upfPortClassT
UPF_LOGIC_EXPR,                 --upfPowerStateT
UPF_LOGIC_REFS,                 --upfStrategyT
UPF_LOWER_BOUNDARY,             --upfPowerDomainT
UPF_LSB,                        --upfHdlMultiBitSliceT
UPF_MODEL_NAME,                 --upfCellT
UPF_MSB,                        --upfHdlMultiBitSliceT
UPF_NAME,                       --upfBaseT
UPF_NAME_PREFIX,                --upfBoundaryStrategyT
UPF_NAME_SUFFIX,                --upfBoundaryStrategyT
UPF_NETWORK_ATTRIBUTES,         --upfNetworkClassT
UPF_NEXT_EXTENT,                --upfExtentT
UPF_NORMALIZED_BITS,            --upfHdlNetMultiBitT,
                                --  upfHdlPortMultiBitT
UPF_NORMALIZED_IDX,             --upfHdlNetBitT, upfHdlPortBitT
UPF_OBJECT,                     --upfExtentT
UPF_OUTPUT_SUPPLY_PORT,         --upfPowerSwitchT
UPF_PARENT,                     --upfBaseT
UPF_PD_STATE_TRANSITIONS,       --upfPowerDomainT,
                                --  upfCompositeDomainT
UPF_PD_STATES,                  --upfPowerDomainT,
                                --  upfCompositeDomainT
UPF_PORT_DIR,                   --upfHdlPortClassT, upfPortClassT
UPF_PST_HEADER,                 --upfPowerStateTableT
UPF_PST_STATES,                 --upfPowerStateTableT
UPF_REF_KIND,                   --upfNamedRefT
UPF_REF_OBJECT,                 --upfNamedRefT
UPF_REPEATER_STRATEGIES,        --upfPowerDomainT
UPF_RESOLVE_TYPE,               --upfSupplyNetT
UPF_RESTORE_CONDITION,          --upfRetentionStrategyT
UPF_RESTORE_SIGNAL,             --upfRetentionStrategyT
UPF_RETENTION_CONDITION,        --upfRetentionStrategyT
UPF_RETENTION_PARAMETERS,       --upfRetentionStrategyT
UPF_RETENTION_STRATEGIES,       --upfPowerDomainT
UPF_ROOT_DRIVER,                --upfNetworkClassT
UPF_SAVE_CONDITION,             --upfRetentionStrategyT
UPF_SAVE_SIGNAL,                --upfRetentionStrategyT
UPF_SIGNAL_SENSITIVITY,         --upfSignalSenseT
UPF_SIMSTATE,                   --upfPowerStateT
UPF_SINK_FILTER,                --upfBoundaryStrategyT
UPF_SLICE_BITS,                 --upfHdlMultiBitSliceT
```

```
  UPF_SMALLEST_ATOMIC_SLICE,      --upfHdlNetBitT, upfHdlPortBitT
  UPF_SOURCE_EXTENTS,             --upfCellT
  UPF_SOURCE_FILTER,              --upfBoundaryStrategyT
  UPF_SP_STATES,                  --upfSupplyPortT
  UPF_SS_STATES,                  --upfSupplySetT
  UPF_SS_TRANSITIONS,             --upfSupplySetT
  UPF_STATE,                      --upfSupplyTypeT
  UPF_SUBDOMAINS,                 --upfCompositeDomainT
  UPF_SUPPLY_EXPR,                --upfPowerStateT
  UPF_SUPPLY_SET_HANDLES,         --upfExtentClassT,
                                  --  upfCompositeDomainT
  UPF_SUPPLY_STATE,               --upfSupplyPortStateT
  UPF_SUPPLY_STATES,              --upfPstStateT
  UPF_SW_STATES,                  --upfPowerSwitchT
  UPF_SWITCH_EXPR,                --upfPowerSwitchStateT
  UPF_SWITCH_OUTPUT_STATE,        --upfPowerSwitchStateT
  UPF_THRESHOLD_VALUE,            --upfLevelShifterStrategyT
  UPF_TO_STATES,                  --upfPowerStateTransitionT
  UPF_UPPER_BOUNDARY,             --upfPowerDomainT
  UPF_USER_CLAMP_VALUES,          --upfIsolationStrategyT
  UPF_VOLTAGE,                    --upfSupplyTypeT
  UPF_VOLT_KIND,                  --upfSupplyPortStateT
  UPF_VOLT_MAX,                   --upfSupplyPortStateT
  UPF_VOLT_MIN,                   --upfSupplyPortStateT
  UPF_VOLT_NOM                    --upfSupplyPortStateT
);

type upfAppliesToFilterE is (
  UPF_FILTER_UNDEF,           -- Undefined
  UPF_FILTER_INPUTS,          -- -applies_to inputs
  UPF_FILTER_OUTPUTS,         -- -applies_to outputs
  UPF_FILTER_BOTH             -- -applies_to both
);

type upfCellKindE is (
  UPF_CELL_NONE,              -- not a cell
  UPF_CELL_ISOLATION,         -- represents isolation cell
  UPF_CELL_LEVEL_SHIFTER,     -- represents level shifter cell
  UPF_CELL_ISO_LS_COMBO,      -- represents isolation and level
                              --   shifter combo cell
  UPF_CELL_RETENTION,         -- represents retention cell
  UPF_CELL_SWITCH,            -- represents a switch cell
  UPF_CELL_REPEATER,          -- represents a repeater or buffer cell
  UPF_CELL_CORRUPT,           -- represents any standard cell which
                              --   can get corrupted
  UPF_CELL_MACRO              -- represents a macro cell or power
                              --   model
);

type upfCellOriginE is (
  UPF_ORIGIN_UNKNOWN,         -- When cell origin is not known
  UPF_ORIGIN_DESIGN,          -- When cell is present in design
                              --   itself
  UPF_ORIGIN_INSERTED,        -- When cell is inserted by UPF after
                              --   application of strategy (e.g.
                              --   isolation) and using default
                              --   model.
  UPF_ORIGIN_INSERTED_MAP,    -- When cell is inserted by UPF after
                              --   application of strategy and using
                              --   user specified model via map_*
                              --   and use_interface_cell commands
  UPF_ORIGIN_INFERRED         -- When cell is inferred by UPF at RTL.
                              --   This information will only be
                              --   present on cells which are set on
                              --   HDL Port or Nets group of objects.
);

type upfHdlCellKindE is (
  UPF_HDLCELL_NONE,           -- not a cell
  UPF_HDLCELL_COMB,           -- represents a combinatorial logic
  UPF_HDLCELL_FLOP,           -- represents a flip flop
  UPF_HDLCELL_LATCH,          -- represents a latch
  UPF_HDLCELL_MEM             -- represents a memory like ram, etc.
);

type upfIsolationClampE is (
  UPF_CLAMP_UNDEF,            -- Undefined
  UPF_CLAMP_ZERO,             -- -*_clamp 0
  UPF_CLAMP_ONE,              -- -*_clamp 1
  UPF_CLAMP_ZEE,              -- -*_clamp Z
  UPF_CLAMP_LATCH,            -- -*_clamp latch
  UPF_CLAMP_ANY,              -- -*_clamp any
  UPF_CLAMP_USER_VALUE        -- -*_clamp value
);

type upfLevelShifterRuleE is (
  UPF_LS_LOW_TO_HIGH,         -- -rule low_to_high
  UPF_LS_HIGH_TO_LOW,         -- -rule high_to_low
  UPF_LS_BOTH                 -- -rule both
);

type upfLocationE is (
  SELF,                       -- -location self
  OTHER,                      -- -location other
  PARENT,                     -- -location parent
  AUTOMATIC,                  -- -location automatic
  FANOUT                      -- -location fanout
);

type upfNamedRefKindE is (
  UPF_REF_POWER,              -- power function of supply set
  UPF_REF_GROUND,             -- ground function of supply set
  UPF_REF_PWELL,              -- pwell function of supply set
  UPF_REF_NWELL,              -- nwell function of supply set
  UPF_REF_DEEPPWELL,          -- deeppwell function of supply set
  UPF_REF_DEEPNWELL,          -- deepnwell function of supply set
  UPF_REF_ISO_SIGNAL,         -- reference to isolation control
                              --   signal in set_isolation
  UPF_REF_SAVE_SIGNAL,        -- to save_signal in set_retention
  UPF_REF_RESTORE_SIGNAL,     -- reference to restore_signal in
                              --   set_retention
  UPF_REF_GENERIC_CLOCK,      -- reference to UPF_GENERIC_CLOCK in
                              --   set_retention
  UPF_REF_GENERIC_DATA,       -- reference to UPF_GENERIC_DATA in
                              --   set_retention
  UPF_REF_GENERIC_ASYNC_LOAD, -- reference to UPF_GENERIC_ASYNC_LOAD
                              --   in set_retention
  UPF_REF_GENERIC_OUTPUT,     -- reference to UPF_GENERIC_OUTPUT in
                              --   set_retention
  UPF_REF_USER_DEFINED        -- some user defined ref handle
);

type upfPortDirE is (
  UPF_DIR_UNDEF,              -- Undefined
  UPF_DIR_IN,                 -- -direction in
  UPF_DIR_OUT,                -- -direction out
  UPF_DIR_INOUT               -- -direction inout
);

type upfResolveE is (
  UNRESOLVED,                 -- -resolve unresolved
  ONE_HOT,                    -- -resolve one_hot
  PARALLEL,                   -- -resolve parallel
  PARALLEL_ONE_HOT            -- -resolve parallel_one_hot
);

type upfRetentionParamE is (
  RET_SUP_COR,
  NO_RET_SUP_COR,
  SAV_RES_COR,
  NO_SAV_RES_COR
);

type upfSignalSenseKindE is (
  UPF_SENSE_HIGH,             -- high
  UPF_SENSE_LOW,              -- low
  UPF_SENSE_POSEDGE,          -- posedge
  UPF_SENSE_NEGEDGE           -- negedge
);

type upfSimstateE is (
  CORRUPT                       -- -simstate CORRUPT or -simstate
                                --   NOT_NORMAL
  CORRUPT_ON_ACTIVITY,          -- -simstate CORRUPT_ON_ACTIVITY or
                                --   -simstate NOT_NORMAL
  CORRUPT_ON_CHANGE,            -- -simstate CORRUPT_ON_CHANGE or
                                --   -simstate NOT_NORMAL
  CORRUPT_STATE_ON_ACTIVITY,    -- -simstate CORRUPT_STATE_ON_ACTIVITY
                                --   or -simstate NOT_NORMAL
  CORRUPT_STATE_ON_CHANGE,      -- -simstate CORRUPT_STATE_ON_CHANGE
                                --   or -simstate NOT_NORMAL
  NORMAL,                       -- -simstate NORMAL
);

type upfVoltKindE is (
  NOM,                        -- when only nominal value is specified
                              --   in add_port_state command
  DOUBLET,                    -- when a doublet is specified in
                              --   add_port_state command
  TRIPLET                     -- when a triplet is specified in
                              --   add_port_state command
);

--------------------------------------
-- HDL Access Functions
--------------------------------------

-- Purpose: Get a handle to a given object from the pathname
impure function upf_get_handle_by_name (
  -- A string representing handle ID for an object
  pathname    : upfStringT;
  -- An optional handle to the object from which the relative
  --   pathname is given
  relative_to : upfHandleT := 0)
  -- Return Value: Returns the handle to the specified property or
  --              null if not found
return upfHandleT;
```

```
--------------------------------------

-- Purpose: Query property on a given object
impure function upf_query_object_properties (
  -- UPF Handle of the given object
  object_handle : upfHandleT;
  -- The enumerated value corresponding to the given property on the
  --   object
  prop          : upfPropertyIdE)
  -- Return Value: Returns the handle to the specified property or
  --               null if not found
return upfHandleT;

--------------------------------------

-- Purpose: Get the next handle from the iterator
impure function upf_iter_get_next (
  -- Handle to the iterator
  iter_handle : upfIteratorT)
  -- Return Value: Returns the handle to the next element in the
  --               iterator or 0 if no element is present.
return upfHandleT;

--------------------------------------
-- Immediate Read Access Functions
--------------------------------------

-- The following table provides the mapping of basic property and the
-- read access routine to get the value.
-------------------------------------------------
-- Type Name             VHDL
-------------------------------------------------
-- upfStringT            upf_get_value_str()
-- upfIntegerT           upf_get_value_int()
-- upfBooleanT           upf_get_value_int()
-- All Enumerated Types  upf_get_value_int()
-- upfRealT              upf_get_value_real()
-------------------------------------------------

-- Purpose: Get the string value from property handle
impure function upf_get_value_str (
  -- Handle to the property
  prop : upfHandleT)
  -- Return Value: Returns string value of the given property handle
  --               or empty string if error.
return upfStringT;

--------------------------------------

-- Purpose: Get the integer value from property handle
impure function upf_get_value_int (
  -- Handle to the property
  prop : upfHandleT)
  -- Return Value: Returns integer value of the given property
  --               handle.
return upfIntegerT;

--------------------------------------

-- Purpose: Get the real value from property handle
impure function upf_get_value_real (
  -- Handle to the property
  prop : upfHandleT)
  -- Return Value: Returns real value of the given property handle.
return upfRealT;

--------------------------------------

-- Purpose: Get the value of supply net type from the property handle
impure function upf_get_label_upfSupplyTypeT (
  -- Handle to the property
  prop : upfHandleT)
  -- Return Value: Returns upfSupplyTypeT for given property handle.
return upfSupplyTypeT;

--------------------------------------

-- Purpose: Get the supply value of a Net
impure function get_supply_value (
  -- A string representing pathname of supply net
  name : upfStringT)
  -- Return Value Returns upfSupplyTypeT for given supply net.
return upfSupplyTypeT;

--------------------------------------

-- Purpose: Get the supply voltage of a net
impure function get_supply_voltage (
  -- An HDL object of upfSupplyTypeT
  arg : upfSupplyTypeT)
  -- Return Value: Returns supply voltage of given upfSupplyTypeT
return upfRealT;

--------------------------------------

-- Purpose: Get the supply FULL_ON state
impure function get_supply_on_state (
  --  An object of upfSupplyTypeT
  arg : upfSupplyTypeT)
  -- Return Value: Returns 1 when supply net state is FULL_ON or
  --               PARTIAL_ON and set_partial_on_translation FULL_ON
  --               is called. It returns 0 in other cases.
return upfBooleanT;

--------------------------------------

-- Purpose: Get the state from a supply net
impure function get_supply_state (
  -- An object of upfSupplyTypeT
  arg : upfSupplyTypeT)
  -- Return Value: Returns state of the supply net
return upfSupplyStateE;

--------------------------------------
-- Immediate Write Access Functions
--------------------------------------

-- The following table provides the mapping between basic property
-- types and corresponding hdl function to write the value
------------------------------------------------------------
-- Type Name             VHDL
------------------------------------------------------------
-- upfStringT            -
-- upfIntegerT           upf_set_value_int()
-- upfBooleanT           upf_set_value_int()
-- All Enumerated Types  upf_set_value_int()
-- upfRealT              -
-- upfSupplyTypeT        upf_set_value_upfSupplyTypeT()
------------------------------------------------------------

-- Purpose: Sets the integer value on the dynamic property
impure function upf_set_value_int (
  -- Handle to the property
  prop  : upfHandleT;
  -- Value
  value : upfIntegerT)
  -- Return Value: Returns 1 on success, 0 on failure
return upfBooleanT;

--------------------------------------

-- Purpose: Sets the upfSupplyTypeT value on the dynamic property
impure function upf_set_value_upfSupplyTypeT (
  -- Handle to the property
  prop  : upfHandleT;
  -- Value of type upfSupplyTypeT
  value : upfSupplyTypeT)
  -- Return Value: Returns 1 on success, 0 on failure
return upfBooleanT;

--------------------------------------

-- Purpose: Change state of supply to FULL_ON and specify voltage
impure function supply_on (
  -- A string representing pathname of the root supply
  supply_name : upfStringT;
  -- Voltage value in real
  value       : upfRealT := 1.0)
  -- Return Value: Returns 1 on success, 0 on failure
return upfBooleanT;

--------------------------------------

-- Purpose: Change state of supply to OFF
impure function supply_off (
  -- A hierarchical path ID for a root supply.
  supply_name : upfStringT)
  -- Return Value: Returns 1 on success, 0 on failure
return upfBooleanT;

--------------------------------------

-- Purpose: Change state of supply to PARTIAL_ON and specify voltage
impure function supply_partial_on (
  -- A string representing pathname of the root supply
  supply_name : upfStringT;
  -- Voltage value in real
  value    : upfRealT := 1.0)
  -- Return Value: Returns 1 on success, 0 on failure
return upfBooleanT;

--------------------------------------

-- Purpose: Assign the specified supply state to this object.
impure function set_supply_state (
  -- the hierarchical path ID of a supply port, net or supply set function
  object_name : upfStringT;
  -- one of OFF, PARTIAL_ON, FULL_ON, UNDETERMINED
  supply_state : upfSupplyStateE)
  -- Return Value: Returns 1 on success, 0 on failure
return upfBooleanT;

--------------------------------------

-- Purpose: Make the specified power state active for this object.
impure function set_power_state_by_handle (
  -- Handle to the UPF object
  object      : upfHandleT;
  -- Handle of the power state present on the object
  power_state : upfHandleT)
  -- Return Value: Returns 1 on success, 0 on failure
return upfBooleanT;

--------------------------------------

-- Purpose: Make the specified power state active for this object.
impure function set_power_state (
  -- the hierarchical path ID of a supply set, power domain,
  --   composite domain, group, model, or instance
  object_name : upfStringT;
  -- the simple name of a power state of that object
  power_state : upfStringT)
  -- Return Value: Returns 1 on success, 0 on failure
return upfBooleanT;

--------------------------------------
-- Continuous Access Functions
--------------------------------------

-- Purpose: Create a continuous monitor that monitors the dynamic
--          property on the given object
impure function upf_create_object_mirror (
  -- A string representing hierarchical path ID of the source object
  --   whose value will be continuously monitored
  src : UpfStringT;
  -- A string representing hierarchical path ID of the destination
  --   object on which the value will be transferred from source
  --   object
  dst : UpfStringT)
  -- Return Value: Returns 1 when mirroring is successful or 0
  --               otherwise
return upfBooleanT;

--------------------------------------
-- Utility Functions
--------------------------------------

-- Purpose: Get the class id of the given object handle
impure function upf_query_object_type (
  -- Handle to the object or property
  handle : upfHandleT)
  -- Return Value: Returns enumerated value representing class of the
  --               given object or property
return upfClassIdE;

--------------------------------------

-- Purpose Check if object belongs to particular class
impure function upf_object_in_class (
  -- Handle to the object or property
  handle   : upfHandleT;
  -- The enumerated value representing the class of the object.
  class_id : upfClassIdE)
```

```
    -- Return Value: Returns 1 when handle belongs to class and 0
    --               otherwise.
  return upfBooleanT;

  --------------------------------------

  -- Purpose: Get the pathname of the given handle
  impure function upf_query_object_pathname (
    -- Handle to a valid object
    object      : upfHandleT;
    -- An optional handle to the object from which the relative
    --   pathname is required
    relative_to : upfHandleT := 0)
    -- Return Value: Returns the hierarchical pathname of the given
    --               object
  return upfStringT;

  --------------------------------------

  -- Purpose: Converts the voltage value in volts represented as real
  --          value to micro volts represented as integer.
  impure function upf_convert_real_volts_to_int_uvolts (
    -- Voltage value in volts represented as a real number
    volt      : upfRealT)
    -- Return Value: Returns the converted value of voltage in real
    --               converted to integer value in micro volts
  return upfIntegerT;

  --------------------------------------

  -- Purpose: Converts the voltage in micro volts represented as
  --          integer to volts represented as real value
  impure function upf_convert_int_uvolts_to_real_volts (
    -- Voltage value in micro volts represented as a integer number
    uvolt      : upfIntegerT)
    -- Return Value: Returns the converted value of voltage in micro
    --               volts represented as integer to volts represented
    --               as real value
  return upfRealT;

  --------------------------------------
  -- Pre-defined supply net resolution functions
  --------------------------------------

  type upfSupplyTypeVectorT is array (INTEGER range <>)
    of upfSupplyTypeT;

  impure function one_hot (
    sources: upfSupplyTypeVectorT)
  return upfSupplyTypeT;

  impure function parallel (
    sources: upfSupplyTypeVectorT)
  return upfSupplyTypeT;

  impure function parallel_one_hot (
    sources: upfSupplyTypeVectorT)
  return upfSupplyTypeT;

end package UPF;


------------------------------------------------------------------

-- EOF
```

Annex A

(informative)

Bibliography

Bibliographical references are resources that provide additional or helpful material but do not need to be understood or used to implement this standard. Reference to these resources is made for informational use only.

[B1] IEEE Standards Dictionary Online.[17]

[B2] IEEE Std 1666™, IEEE Standard for Standard SystemC Language Reference Manual.

[B3] ISO/IEC 8859-1, Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin Alphabet No. 1.[18]

[B4] Liberty library format usage.[19]

[B5] Tcl language syntax summary.[20]

[B6] Tcl language usage.[21]

[17] Available at http://www.ieee.org/publications_standards/publications/subscriptions/prod/standards_dictionary.html.
[18] ISO/IEC publications are available from the ISO Central Secretariat (http://www.iso.org/). ISO publications are also available in the United States from the American National Standards Institute (http://www.ansi.org/).
[19] Available at https://www.opensourceliberty.org.
[20] Available at http://www.tcl.tk/man/tcl8.4/TclCmd.
[21] Available at http://sourceforge.net/projects/tcl/.

Annex B

(normative)

Value conversion tables

The predefined value conversion tables (VCTs) are as follows.

B.1 VHDL_SL2UPF

```
create_hdl2upf_vct VHDL_SL2UPF
-hdl_type vhdl
-table { {'U' UNDETERMINED}
              {'X' UNDETERMINED}
              {'0' OFF}
              {'1' FULL_ON}
              {'Z' UNDETERMINED}
              {'L' OFF}
              {'H' FULL_ON}
              {'W' UNDETERMINED}
              {'-' UNDETERMINED}}
```

B.2 UPF2VHDL_SL

```
create_upf2hdl_vct UPF2VHDL_SL
-hdl_type vhdl
-table {{UNDETERMINED 'X'}
             {PARTIAL_ON 'X'}
             {FULL_ON '1'}
             {OFF '0'}}
```

B.3 VHDL_SL2UPF_GNDZERO

```
create_hdl2upf_vct VHDL_SL2UPF_GNDZERO
-hdl_type vhdl
-table { {'U' UNDETERMINED}
              {'X' UNDETERMINED}
              {'0' FULL_ON}
              {'1' OFF}
              {'Z' UNDETERMINED}
              {'L' FULL_ON}
              {'H' OFF}
              {'W' UNDETERMINED}
              {'-' UNDETERMINED}}
```

B.4 UPF_GNDZERO2VHDL_SL

```
create_upf2hdl_vct UPF_GNDZERO2VHDL_SL
-hdl_type vhdl
-table {{UNDETERMINED 'X'}
             {PARTIAL_ON 'X'}
             {OFF '1'}
             {FULL_ON '0'}}
```

B.5 SV_LOGIC2UPF

```
create_hdl2upf_vct SV_LOGIC2UPF
-hdl_type sv
-table {{X UNDETERMINED}
            {Z UNDETERMINED}
            {1 FULL_ON}
            {0 OFF }}
```

B.6 UPF2SV_LOGIC

```
create_upf2hdl_vct UPF2SV_LOGIC
-hdl_type sv
-table {{UNDETERMINED X}
            {PARTIAL_ON X}
            {FULL_ON 1}
            {OFF 0}}
```

B.7 SV_LOGIC2UPF_GNDZERO

```
create_hdl2upf_vct SV_LOGIC2UPF_GNDZERO
-hdl_type sv
-table {{X UNDETERMINED}
              {0 FULL_ON}
              {1 OFF}
              {Z UNDETERMINED}}
```

B.8 UPF_GNDZERO2SV_LOGIC

```
create_upf2hdl_vct UPF_GNDZERO2SV_LOGIC
-hdl_type sv
-table {{UNDETERMINED X}
             {PARTIAL_ON X}
             {OFF 1}
             {FULL_ON 0}}
```

B.9 VHDL_TIED_HI

```
create_upf2hdl_vct VHDL_TIED_HI
-hdl_type vhdl
-table {{UNDETERMINED 'X'}
            {FULL_ON '1'}
            {PARTIAL_ON 'X'}
            {OFF 'X'}}
```

B.10 SV_TIED_HI

```
create_upf2hdl_vct SV_TIED_HI
-hdl_type sv
-table {{UNDETERMINED X}
            {FULL_ON 1}
            {PARTIAL_ON X}
            {OFF X}}
```

B.11 VHDL_TIED_LO

```
create_upf2hdl_vct VHDL_TIED_LO
-hdl_type vhdl
-table {{UNDETERMINED 'X'}
            {FULL_ON '0'}
            {PARTIAL_ON '0'}
            {OFF 'X'}}
```

B.12 SV_TIED_LO

```
create_upf2hdl_vct SV_TIED_LO
-hdl_type sv
-table {{UNDETERMINED X}
            {FULL_ON 0}
            {PARTIAL_ON X}
            {OFF X}}
```

Annex C

(informative)

UPF query examples

This annex lists a few sample Tool Command Language (Tcl) procs that perform some high-level queries and are built upon basic Unified Power Format (UPF) queries.

C.1 Utility procs

C.1.1 Introduction

The utility procs are some useful procs that are used by more complex procs to perform a specific functionality.

C.1.2 Get strategy from port

```
proc get_port_strategy_extent {port strategy} {
    set extents [upf_query_object_properties $port \
        -property upf_extents];
    foreach extent $extents {
        set upf [upf_query_object_properties $extent \
            -property UPF_OBJECT];
        if {[upf_query_object_type $upf] == $strategy} {
            return $extent
        }
    }
    return ""
}
```

C.1.3 Get list of effective extents from UPF object

```
proc query_effective_extent_list {extent} {
    set result "";
    # Check for empty arg
    if {$extent == ""} {
        return $result;
    }
    # Check if incorrect object is passed
    if {[upf_query_object_type $extent] != "upfExtentT"} {
        return $result;
    }
    # Traverse to next extent
    set result [concat $result
                  [query_effective_extent_list \
                    [upf_query_object_properties $extent \
                     -property upf_next_extent]]];
    # Add the current extent to the list
    lappend result $extent;
    return $result;
}
```

C.2 High-level procs

C.2.1 Check whether port has isolation cell

```
proc is_port_isolated {port} {
    if {[get_port_strategy_extent $port \
        upfIsolationStrategyT] } {
        return "true"
    }
    return "false"
}
```

Usage

```
    is_port_isolated mid/port1
    is_port_isolated /top/dut_i/port1
```

Output

```
    false
    true
```

C.2.2 get strategy name corresponding to isolation cell

```
proc get_port_iso_strat_name {port} {
    set extent [get_port_strategy_extent $port \
        upfIsolationStrategyT];
    if {$extent != ""} {
        set upf [upf_query_object_properties $extent \
            -property upf_object];
        return [upf_query_object_properties $upf \
            -property upf_name];
    }
    return ""
}
```

Usage

```
    get_port_iso_strat_name mid/port1
```

Output

```
    iso
```

C.2.3 Check isolation clamp value matches with given value

```
proc check_isolation_clamp {port reset_val} {
    set extent [get_port_strategy_extent $port \
        upfIsolationStrategyT];
    if {$extent != ""} {
        set upf [upf_query_object_properties $extent \
            -property upf_object];
        set clampv [upf_query_object_properties $extent \
            -property upf_clamp_values];
        if {[lindex $clampv 0] == $reset_val} {
            return 1;
        }
    }
    return 0;
}
```

Usage

```
    check_isolation_clamp top/dut_i/port 1
```

Output

```
    1
```

C.2.4 Print effective element list

```
proc print_effective_element_list {upf} {
    if {[upf_object_in_class $upf \
        -class upfExtentClassT] != 1} {
        return;
    }
    set extent_head [upf_query_object_properties $upf \
        -property upf_effective_extents];
    set extents [query_effective_extent_list $extent_head]
    foreach extent $extents {
        set element [upf_query_object_properties $extent \
            -property upf_hdl_element];
        puts [upf_query_object_pathname $element];
    }
    return;
}
```

Usage

```
    print_effective_element_list /top/dut_i/pd
```

Output

```
    /top/dut_i
    /top/dut_i/mid
```

C.2.5 Print isolation info from PD

```
proc print_iso_info {upf} {
    if {[upf_object_in_class $upf \
        -class upfPowerDomainT] != 1} {
        return;
    }
    set pd_iso [upf_query_object_properties $upf \
        -property upf_isolation_strategies];
    foreach iso $pd_iso {
        # Print name of strategy
        puts [upf_query_object_properties $iso \
            -property upf_name];
        # Print clamp value
        puts [upf_query_object_properties $iso \
            -property upf_clamp_values];
        # Print Control info
        set ctrlsense lindex \
            [upf_query_object_properties $iso \
            -property upf_isolation_controls] 0;
        puts $ctrlsense;
        # Print Control signal ID
        puts [upf_query_object_properties $ctrlsense \
            -property upf_control_signal];
        # Print Control signal sensitivity
        puts [upf_query_object_properties $ctrlsense \
            -property upf_signal_sensitivity];
    }
    return;
}
```

Usage

```
    print_effective_element_list PD_Proc
```

Output

```
    ISOproc
    0
    #UPFSIGSENSE1#
    /Sub/pISO
    upf_sense_high
```

Annex D

(informative)

Replacing deprecated and legacy commands and options

This annex shows the commands and command options that have been categorized as deprecated or legacy since the last version of this standard, and recommendations for replacing them (where applicable).

Legacy constructs (commands and/or options) have not had their syntax and/or semantics updated to be consistent with other commands in this version of the standard, so their descriptions may contain significant obsolete information and their semantics may not be interoperable with the latest Unified Power Format (UPF) concepts. For recommendations on how to use current constructs to replace legacy and deprecated ones, see D.2.

D.1 Deprecated and legacy constructs

D.1.1 Introduction

The following subclauses shows any constructs that have been categorized as deprecated or legacy constructs (see also 6.2). For recommendations on replacing them, see Table D.1.

D.1.2 Deprecated constructs

This subclause lists the deprecated commands and options.

There are currently no deprecated constructs.

D.1.3 Legacy constructs

D.1.3.1 Overview

Subclause D.1.3 lists the legacy commands and options.

D.1.3.2 add_port_state (see also 6.4)

add_port_state *port_name*
  {**-state {***name* <*nom* | *min max* | *min nom max* | **off**>**}**}*

D.1.3.3 add_pst_state (see also 6.6)

add_pst_state *state_name*
  -pst *table_name*
  -state *supply_states*

D.1.3.4 create_pst (see also 6.23)

create_pst *table_name*
    -supplies *supply_list*

D.1.3.5 describe_state_transition (see also 6.28)

describe_state_transition *transition_name* **-object** *object_name*
[**-from** *from_list* **-to** *to_list*]
[**-paired** **{{***from_state to_state***}***}] [**-legal** | **-illegal**]

D.1.3.6 load_upf_protected (see also 6.33)

load_upf_protected *upf_file_name*
    [**-hide_globals**] [**-scope** *instance_name_list*]
    [**-params** *param_list*]

D.1.3.7 set_domain_supply_net (see also 6.42)

set_domain_supply_net *domain_name*
    -primary_power_net *supply_net_name*
    -primary_ground_net *supply_net_name*

D.1.3.8 set_isolation (see also 6.44)

set_isolation *strategy_name*
    ...
    [**-isolation_power_net** *net_name*] [**-isolation_ground_net** *net_name*] (These are legacy options.)

D.1.3.9 set_retention (see also 6.49)

set_retention *isolation_name*
    ...
    [**-retention_power_net** *net_name*] [**-retention_ground_net** *net_name*] (These are legacy options.)

D.2 Recommendations for replacing deprecated and legacy constructs

Table D.1 shows how to use current constructs to replace deprecated and/or legacy constructs.

Table D.1—Recommended commands and options for replacing deprecated and legacy constructs

Command	Options	Recommended command	Recommended options	Reasons for the recommendation
add_port_state	*port_name* **-state** {*name <options>*}	**add_power_state**	*object_name* **-supply_expr** *boolean_expression*	**add_power_state** is intended to replace the whole of the PST commands
add_pst_state	*state_name* **-pst** *table_name* **-state** *supply_states*	**add_power_state**	**-state** *state_name* N/A **-supply_expr** {*boolean_expression*}	**add_power_state** is intended to replace the whole of the PST commands
create_pst	*table_name* **-supplies** *supply_list*	**add_power_state**	**-state** *state_name* N/A **-supply_expr** {*boolean_expression*}	**add_power_state** is intended to replace the whole of the PST commands
describe_state_transition	*transition_name* **-object** *object_name* [**-from** *from_list* **-to** *to_list*] [**-paired** {{*from_state to_state*}*}] [**-legal** | **-illegal**]	**add_state_transition**	*object_name* **-transition** *transition_name* **-from** *from_list* **-to** *to_list* **-paired** {{*from_state to_state*}*} **-legal** | *-illegal*	**add_state_transition** is intended to replace **describe_state_transition**
load_upf_protected	*upf_file_name* [**-hide_globals**] [**-scope** *instance_name_list*] [**-params** *param_list*]	**load_upf**	*upf_file_name* [**-hide_globals**] [**-scope** *instance_name_list*] [**-parameters** {{*parameter_name* [*parameter_value*]}*}]	Simplification of **load_upf_protected** and **load_upf**
set_domain_supply_net	*domain_name* **-primary_power_net** *net* **-primary_ground_net** *net*	**associate_supply_set**	*supply_set* **-handle** *supply_set_handle* (for both)	Superseded by a a more abstract concept
set_isolation	**-isolation_power_net** *net* **-isolation_ground_net** *net*	**set_isolation**	**-isolation_supply** *set* (for both)	Superseded by a more abstract concept
set_retention	**-retention_power_net** *net* **-retention_ground_net** *net*	**set_retention**	**-retention_supply** *set* (for both)	Superseded by a more abstract concept

Annex E

(informative)

Low-power design methodology

The purpose of this annex is two-fold. First, various design flows with a recommended use model of Unified Power Format (UPF) are illustrated. Second, a simple design example is used to demonstrate how these various power intent aware design flows can be built.

E.1 Simple System on Chip (SoC) example design

E.1.1 Introduction

Consider a simple design shown in Figure E.1. This design has the module name `soc`, which contains glue logic at the top, and other intellectual property (IP) blocks—MPCore (mpcore), display controller (display), power control unit (pcu) and memory controller (mem_controller). The MPCore and display controller IPs have been designed and verified as independent IPs and integrated in the SoC.

The MPCore IP is a dual central processing unit (CPU) multi-processor IP that consists of L2 cache random access memory (RAM)—`l2tagram` and `l2dataram` that are hard IPs. The two instances of CPU, named cpu0 and cpu1 are instances of the same module cpu. The CPU consists of L1 cache RAM—`l1tagram` and `l1dataram` that are hard IPs and other logic blocks within.

The display controller IP has a PHY with internal regulator. The power control unit controls the power for the SoC.

E.1.2 Functional power states of the design

The following is the detailed description of the power states of the example design.

The memories support power gating with internal switches. The memory contents can be retained in retention state of the memory when the memory is power gated. The memory supports the following power states—ON, OFF, and RET. In the ON state, the memory leakage power can be reduced by setting to a light-sleep (LS) state through control pins of the memory.

The CPU supports power-gating and voltage-scaling. The CPU is in power domain PDCPU. In power down state, the state of the CPU can be retained. The CPU can be in one of these states: ON, OFF, or RET. When the CPU is in RET state, the L1 memories shall be in RET state. When the CPU is in OFF state, the L1 memories shall be in OFF state. When the CPU is in ON state the L1 memories can be in any of its legal states.

PDCPU state	L1 memory state
ON	ON, RET, OFF, LS
RET	RET
OFF	OFF

The MPCORE supports power-gating and voltage-scaling. The MPCORE is in power domain PDMPCORE, can be either ON or OFF state. In ON state, cpu0, cpu1, and L2 memories can be in any of its legal states. In OFF state, cpu0, cpu1, and L2 memories will be in OFF state.

PDMPCORE state	L2 memory state	L1 memory state
ON	ON, RET, OFF	ON, RET, OFF, LS
OFF	OFF	OFF

The CPU and MPCORE operate on same supply VCPU that has a nominal voltage of 0.7 V and scales from 0.63 V to 0.77 V based on the SoC performance and power requirements decided during its operation by higher-level firmware.

The display controller PHY is a hard IP that has an internal regulator. The input to the regulator is VDDI that is 1.8 V. The output of the regulator can either be LOW (0.6 V) or OFF (0 V). The PHY and display controller are powered by the output supply of the regulator. The display controller is in power domain PDDISP. Functionally the display controller can be in either ON or OFF state.

PDDISP state	PHY state
ON	ON
OFF	OFF

The power control unit is an always-on module that is powered by VSOC supply. The power control unit is in power domain PDAON.

The SoC top supports power gating and is powered by VSOC supply. SoC top is set to power domain PDSOC. The SoC can be either in ON or OFF power state. In the ON power state, the underlying IPs can be in any of their legal states. In the OFF state, the MPCORE and display will be in OFF state.

The VSOC supply has a nominal voltage of 0.8 V. VSOC and VCPU supplies are correlated supplies (see 6.39).

PDSOC	PDSOC	PDAON	PDMPCORE state	PDDISP
ON	ON	ON	ON/OFF	ON/OFF
SLEEP	OFF	ON	OFF	OFF

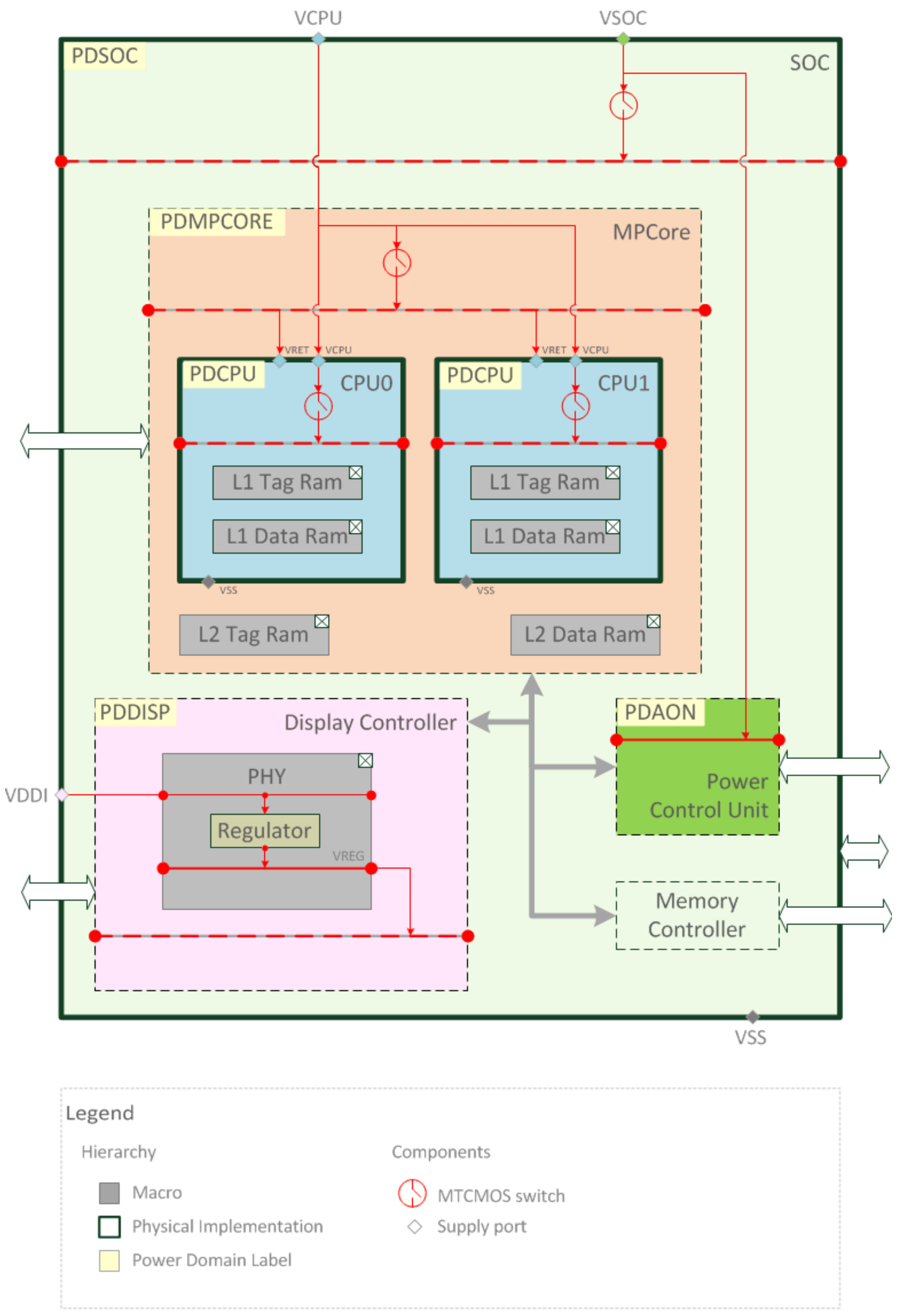


Figure E.1—Simple SoC design

E.1.3 Successive refinement UPF

The SOC power intent is specified using successive refinement UPF as described in 4.9. The CPU, MPCORE, and display controller IPs have constraint UPF available along with the hardware description language (HDL). In the context of the SoC, the configuration and implementation UPF are created.

To meet various engineering challenges in ASIC implementation, it could be decided to harden the CPU and instantiate the same hard macro twice in the MPCORE as cpu0 and cpu1. This engineering decision has no influence on how the verification of the SoC has been done. To be able to implement the CPU as a hard macro, an implementation UPF for the CPU is required. The CPU will have constraint, configuration, and implementation UPF to implement it as a hard macro.

The MPCore and display controller will have constraint and configuration UPF. The SoC top will have constraint, configuration, and implementation UPF.

In summary, an IP will typically have constraint UPF, a soft macro will have configuration UPF, and any block implemented separately must have an implementation UPF.

E.2 Design, verification, and implementation flow

E.2.1 Overview

Figure E.2 illustrates a typical UPF design flow for a SoC like the one shown in Figure E.1.

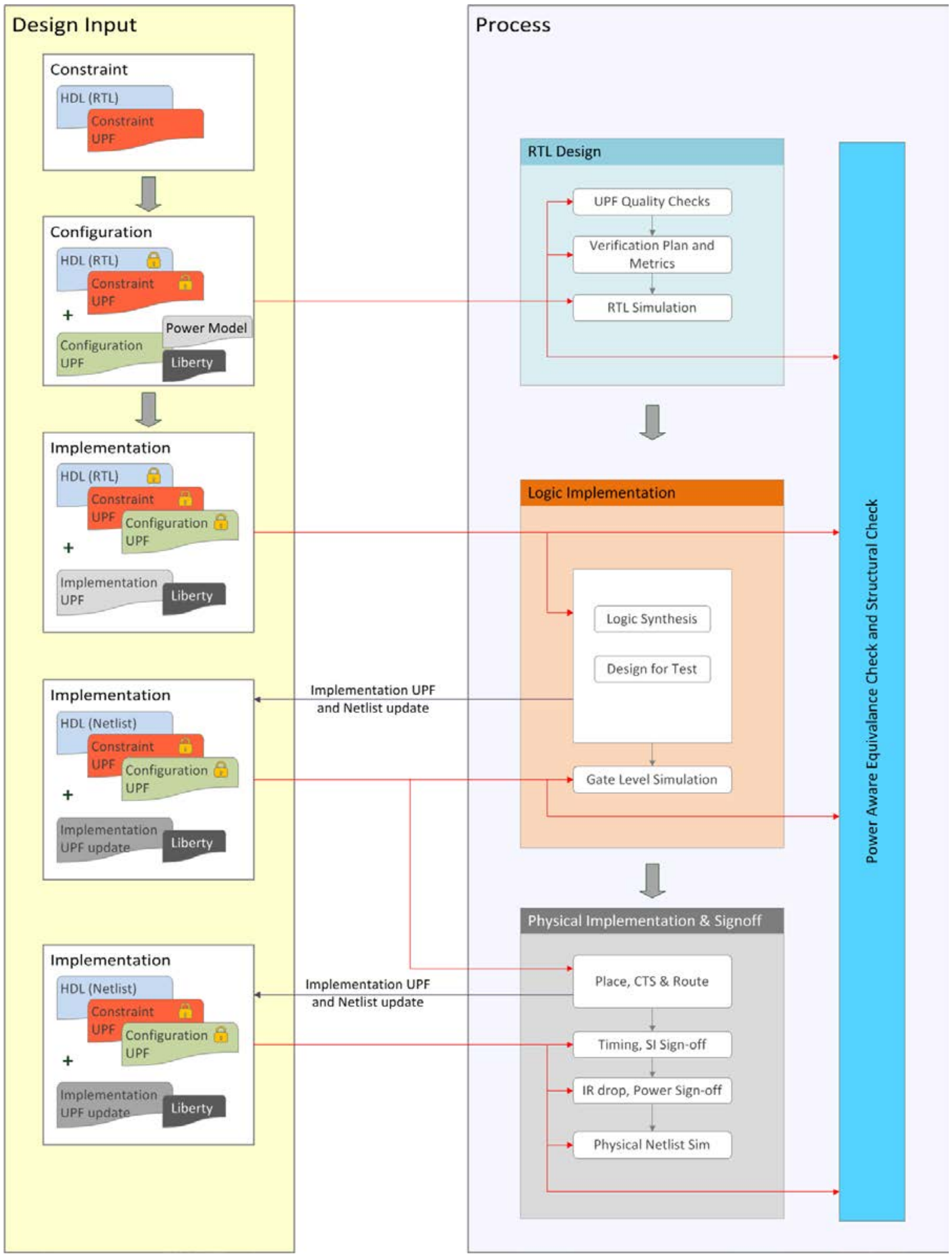


Figure E.2—Typical UPF design flow

For each of the three design stages shown in Figure E.2, the design example in Figure E.1 illustrates how the UPF can be created, used, and passed on to the later stages of the design flow. It starts with RTL design, followed by logic implementation, and then physical implementation.

E.2.2 RTL design stage

The configuration UPF is created at this stage, which includes the UPF power models and/or Liberty models for the hard IPs instantiated within the soft IP. The configuration UPF must satisfy the constraint UPF for the IP blocks.

The hard IPs can use one of these methods to specify the power intent of the hard IPs in the design:

a) UPF macro model that completely specifies the power intent of the hard IP which is descriptive rather than directive.

b) Liberty model in conjunction with a UPF macro model that supplements the Liberty model.

The UPF example shows the use of these two methods to specify power intent.

E.2.3 Logic implementation

The logic implementation stage includes logic synthesis, Design for Test (DFT) synthesis, and gate-level simulation. The following information is typically required in addition to the power intent specified in the RTL stage:

a) The supply ports and supply net definitions, supply net associations with the supply set functions, and supply state of the supply sets; the supply connections for the hard IPs are specified.

b) The isolation supply and location of isolation cells based on the supply availability in each domain.

c) If level-shifters are needed based on the supply voltages, it is specified in the implementation UPF.

d) If designers have some preferences for specific library cells to be used for state retention, isolation, and level-shifting strategies, specify them in the implementation UPF.

DFT synthesis typically creates some new ports and connections in the design that can create new domain crossings that are not covered by the original power intent. Designers need to make sure either of the following occur:

— All newly created ports are covered by existing strategies, which is possible if the strategy was written using path-based strategies (without using **-elements** to specify the exact port name).

— A new isolation strategy is added to cover the new crossing before the physical implementation stage.

E.2.4 Physical implementation

Physical implementation includes all the steps from power planning, placement, routing, power-switch insertion, physical optimization, and sign-off to generating the final physical netlist and layout. The following information is typically required in addition to the power intent specified for the logic implementation stage:

a) Power-switch definitions

b) Other physical implementation constraints, such as the requirements for repeaters (see 6.48)

E.3 Power intent of the example design

E.3.1 Introduction

The UPF for the SoC is constructed bottom up. The constraint UPF for the IPs are assumed to be available as part of the IP. The power model for the hard IP is represented based on the approach adopted for simulation and implementation.

E.3.2 Power model for memory hard IP

The approach described here uses UPF power model in conjunction with the Liberty model of the memory hard IP to represent the power intent of the memory. The Liberty model defines the memory as a macro.

Power intent details that are to be supplemented to the Liberty model to completely specify the power intent of the memory macro are as follows:

a) In a macro that has internal switch, and hence internal power/ground pin, some input/output pins of the macro have related power/ground pin as the internal power/supply pin. Since the internal power/ground pin defined in the Liberty is not available in UPF, the related power/ground attribute specified in Liberty is unusable though that reflects the actual implementation of the macro. Accessing the internal power/ground pin of the macro in the UPF and specifying the power states based on the internal power/ground pin will be done in the power model.

b) Liberty does not specify the power states of the macro to be able to perform power intent checks or power-aware simulations with macro set to different functional states it can be in. So the power states of the memory will be specified in the power model.

The L1 cache RAMs of the CPU use MEMSRAM_1024X32 memory modules. The SystemVerilog module declaration of the memory is as follows:

```
module MEMSRAM_1024X32 (
  `ifdef PG_PINS
    // Core supply port of the memory
    (* UPF_pg_type = "primary_power" *)  input VDDCE;
    // Periphery supply port of the memory
    (* UPF_pg_type = "primary_power" *)  input VDDPE;
    // Common ground supply port of the memory
    (* UPF_pg_type = "primary_ground" *) input VSSE;
  `endif
  output [33:0] Q;
  output        PRDYN;
  input         CLK, CEN, WEN;
  input  [9:0]  A;
  input  [33:0] D;
  input         PGEN; // switch control input
  input         RET;  // retention control input
);
```

Extract from the Liberty model of the memory that shows the key information required to represent the power intent of the memory is shown below:

```
library(MEMSRAM_1024X32_tt_0p72v_0c) {
  pg_pin(VDDCE) {
    voltage_name : VDDCE;
    pg_type : primary_power;
    direction : input;
  }

  pg_pin(VDDPE) {
    voltage_name : VDDPE;
    pg_type : primary_power;
    direction : input;
  }

  pg_pin(VSSE) {
    voltage_name : VSSE;
    pg_type : primary_ground;
    direction : input;
  }

  # Internal PG Pin
  pg_pin(VDDPI) {
    voltage_name : VDDPE;
    pg_type : internal_power;
    direction : internal;
    switch_function : "PGEN";
    pg_function : VDDPE;
  }

  # Power/State Control Pin
  pin(PGEN) {
    direction : input;
    always_on : true;
    switch_pin : true;
    related_power_pin : "VDDPE";
    related_ground_pin : "VSSE";
  }

  pin(RET) {
    direction : input;
    always_on : true;
    related_power_pin : "VDDPE";
    related_ground_pin : "VSSE";
  }

  # Example Input
  bus(D) {
    direction : input;
    related_power_pin : "VDDPI";
    related_ground_pin : "VSSE";
  }

  # Example Output
  bus(Q) {
    direction : output;
    related_power_pin : "VDDPE";
    related_ground_pin : "VSSE";
    power_down_function : "!VDDCE + !VDDPE + VSSE";
  }
}
```

The power model that supplements the Liberty model is show below. The power intent described by the UPF power model for the hard IP is descriptive rather than directive for implementation. For simulation, the power intent is directive. For example, if the power model specifies a switch policy, the policy is directive for simulation and ignored by the implementation tool as the switch within the hard IP has already been implemented.

```
# A memory power model named memPwrModel is created for the memory
# model MEMSRAM_1024X32

begin_power_model memPwrModel -for MEMSRAM_1024X32

# Since the hard IP is a macro as defined in the Liberty, the
# design attribute UPF_is_hard_macro is set on the model.
set_design_attributes -models MEMSRAM_1024X32 -is_hard_macro TRUE

# VDDPI is internal supply net specified in the Liberty that is
# switched off when memory is in OFF or RET state
# Access internal_power pin VDDPI and create a supply set with the
# internal power pin and ground.
create_supply_net VDDPI

create_supply_set ss_vddpi \
  -function {power  VDDPI} \
  -function {ground VSSE}

# Create supply set that bundles the VDDPE and VSSE of memory
create_supply_set ss_vddpe \
  -function {power  VDDPE} \
  -function {ground VSSE}

# Create supply set that bundles the VDDCE and VSSE of memory
create_supply_set ss_vddce \
  -function {power  VDDCE} \
  -function {ground VSSE}

# Create power switch to model the internal switch of the memory
create_power_switch sw_vddp \
  -input_supply_port  {sw_in     VDDPE} \
  -output_supply_port {sw_out    ss_vddpi.power} \
  -control_port       {sw_ctrl   PGEN} \
  -on_state           {on_state  sw_in {!sw_ctrl}} \
  -off_state          {off_state {sw_ctrl}} \
}

# Having defined a supply set ss_vddpi (with VDDPI as power and
# VSSE as ground), predefined power states ON and OFF of the supply set
# can be updated appropriately.
# This is effectively setting power states on VDDPI
add_power_state -supply ss_vddpi -update \
  -state {ON  -logic_expr {PGEN == 1}} \
  -supply_expr {power=={FULL_ON 0.6} && ground=={FULL_ON 0}}} \
  -state {OFF -logic_expr {PGEN == 0}} \
  -supply_expr {power==OFF           && ground=={FULL_ON 0}}}

# With the power states of the supply set and the state of control
# pins of macro, the power states of the macro can be defined.
add_power_state -model MEMSRAM_1024X32 \
  -state {ON  -logic_expr {ss_vddpi==ON           }} \
  -state {RET -logic_expr {ss_vddpi==OFF && RET==1}} \
  -state {OFF -logic_expr {ss_vddpi==OFF && RET==0}} \
  -complete

# Refine power state ON to create a power state LS such that
# in LS state the instance is in ON state and an additional condition
# is met
add_power_state -model MEMSRAM_1024X32 -update \
  -state {ON.LS -logic_expr {MEMSRAM_1024X32==ON && RET==1}}

end_power_model
```

The following key points are to be noted:

a) No power domain has been created in the power model. The hard IP will be in the parent power domain in which it is instantiated.

b) All objects specified in the power model are scoped to the model.

c) No driver/receiver supply attributes are defined for the ports on the interface of the hard macro. The related_power_pin and related _ground_pin attribures in the Liberty model defines the internal environment of the hard macro. The macro will be instantiated in context of the parent domain, and therefore the external environment of the macro is inferred from the actual driver/receiver in the parent domain.

E.3.3 Power model for PHY

The approach described here uses a standalone UPF power model to completely specify the power intent of the PHY. The power model must be self-contained and complete to specify the power intent of the PHY.

```
begin_power_model phyPwrModel for PHY3TX2RX0P8V

# Since the hard IP is a hard macro, the design attribute UPF_is_hard_macro
# is set on the model. This attribute is automatically imported from the
# Liberty model when a Liberty model is read in for the macro.
set_design_attributes -models PHY3TX2RX0P8V_1024X32 -is_hard_macro TRUE

# To ensure the model is self-contained, a power domain is created for the
# PHY and the objects are set to the power domain
create_power_domain PDPHY -elements {.} \
  -supply {ss_regin} \
  -supply {ss_regout}

# Supply set handle ss_regin has input supply VDDI and ground VSS
# associated to power and ground functions
# VDDI is the primary power and VSS is the primary ground of the PHY
create_supply_set PDPHY.ss_regin -update \
  -function {power  VDDI} \
  -function {ground VSS}

# Supply set handle ss_regout has output supply VREG and ground VSS
# associated to power and ground functions
create_supply_set PDPHY.ss_regout -update \
  -function {power  VREG} \
  -function {ground VSS}

# Supply set ss_regin has only 1 state named HIGH as input voltage
# is fixed at 1.8 V
add_power_state -supply PDPHY.ss_regin \
  -state {HIGH -supply_expr {power == {FULL_ON 1.8}}}

# Supply set ss_regout has two state LOW and REG_OFF
# The state of ss_regout supply is controlled by regCtrl input
# In LOW state, the supply is 0.8 V
# In REG_OFF state, the supply is OFF
add_power_state -supply PDPHY.ss_regout \
  -state {LOW     -logic_expr  {regCtrl==1} \
                  -supply_expr {power=={FULL_ON 0.8}}} \
  -state {REG_OFF -logic_expr  {regCtrl==0} \
                  -supply_expr {power==OFF}}

# Power domain PDPHY has two states, ON and OFF
# The state of the power domain is a function of the states of the supplies
# In ON state, the ss_regout is in LOW state
# In OFF state, the ss_regout is in REG_OFF state
add_power_state -domain PDPHY \
  -state {ON  -logic_expr \
    {PDPHY.ss_regin==HIGH && PDPHY.ss_regout==LOW}} \
  -state {OFF -logic_expr \
    {PDPHY.ss_regin==HIGH && PDPHY.ss_regout==REG_OFF}} \

# Terminal Boundary Conditions
# Since the power model must be self-contained, the boundary
# conditions are specified for the model. The hard IP is one that has
# already been implemented and so the conditions as seen from outside
# the macro are specified.
set_port_attributes -model {.} -ports \
  [find_objects . -pattern * -object_type port -direction in] \
  -exclude_ports {regCtrl} -receiver_supply PDPHY.ss_regout

set_port_attributes -model {.} -ports \
  {regCtrl} -receiver_supply PDPHY.ss_regin

set_port_attributes -model {.} -ports \
  [find_objects . -pattern * -object_type port -direction out] \
  -driver_supply PDPHY.ss_regout

end_power_model
```

E.3.4 UPF for CPU

E.3.4.1 Introduction

The CPU UPF loads the constraint, configuration, and implementation UPF for CPU. Since CPU is hardened in implementation, an implementation UPF is required for this hierarchy. The UPF must be self-contained and complete.

```
set_design_top cpu
load_upf cpu_constraints.upf
load_upf cpu_configuration.upf
load_upf cpu_implementation.upf
```

E.3.4.2 Constraint UPF

The constraint UPF for CPU is provided with the HDL of the IP.

```
# Variable Declarations
# List of CPU output ports to be clamped 1 when CPU is OFF.
# This list shall be used to define port attributes on the ports that
# shall determine the clamp value for isolation policy.
set cpuClamp1 [list \
  commtx_o \
  afreadym_o \
  ncommirq_o \
]

# Power Domains
# CPU consists of only one power domain PDCPU. The current scope is the
# domain top.
# Two supply set handles primary and ret are defined in PDCPU
# primary is the supply as defined in the standard
# ret is the supply used as the back-up supply for the retention FFs
create_power_domain PDCPU -elements {.} \
  -supply {primary} \
  -supply {ret}

# Port Attributes for Isolation
# By defining port attributes on the output ports, the clamp values
# are set.
# All the output ports except the list specified by $cpuClamp1 are
# set to clamp value 0
# Ports specified by $cpuClamp1 are set to clamp value 1
set_port_attributes -model {.} -ports \
  [find_objects . -pattern * -object_type port -direction out] \
  -exclude_ports ${cpuClamp1}  -clamp_value 0
set_port_attributes -model {.} -ports "$cpuClamp1" -clamp_value 1

# Retention Elements
# If the CPU is set to retention, then the list of instances specified
# in the cpuRetList must be retained.
# This retention list shall be used in the retention policy specified
# in the configuration UPF
set_retention_elements cpuRetList \
  -elements {u_cpu_noram}

# Power State
# PDCPU: The PDCPU supports three power states ON, RET, OFF
# In the ON state, supply sets primary and ret are ON
# In the RET state, supply set ret is ON and primary is OFF
# In the OFF state, both supply sets are OFF
# Note:
# The ON and OFF states of the supply sets are the deferred power
# states defined by the standard.
add_power_state -domain PDCPU \
  -state {ON  -logic_expr {PDCPU.ret==ON  && PDCPU.primary==ON }} \
  -state {RET -logic_expr {PDCPU.ret==ON  && PDCPU.primary==OFF}} \
  -state {OFF -logic_expr {PDCPU.ret==OFF && PDCPU.primary==OFF}}
```

E.3.4.3 Configuration UPF

The configuration UPF for CPU takes into consideration that the CPU is hardened in implementation and the low-power control signals (such as isolation, retention control) cannot be connected from the top power control unit. All the control ports required for CPU are created at the CPU logical boundary.

The control ports that are to be defined at the CPU logical boundary are:

a) lp_lSleep—switch control for the domain logic power switch
b) lp_rSleep—switch control for the memory power switch
c) lp_lReady—switch control acknowledge from the domain logic power switch
d) lp_rReady—switch control acknowledge from the memory power switch
e) lp_lRet—logic retention control for the domain
f) lp_rRet—memory retention control

When the CPU is switched OFF or set in RET state, the outputs of CPU are isolated. The CPU is implemented as a hard macro, and in this example it is an implementation choice to have the isolation outside of the CPU hard macro and so no isolation policy is specified in the configuration UPF of CPU.

```
# Low-Power Control Ports
# The logic ports required for switch control, retention control and
# switch control acknowledge are created.
# It is allowed to use the logic_port in place of logic_net and so
# logic_net is not explicitly created.
create_logic_port lp_lSleep
create_logic_port lp_rSleep
create_logic_port lp_lReady
create_logic_port lp_rReady
create_logic_port lp_lRet
create_logic_port lp_rRet

# Retention Strategy
# Retention policy is created with the retention list specified in
# constraint UPF. PDCPU.ret supply handle is used as retention supply
# as specified in the constraint UPF. Control port lp_lRet is used as
# retention control signal.
set_retention retCpu \
  -domain           PDCPU \
  -retention_supply PDCPU.ret \
  -save_signal      {lp_lRet negedge} \
  -restore_signal   {lp_lRet posedge} \
  -elements         cpuRetList

# Memory-Related Configuration
# List of memory instances in CPU
set l1MemInstances [list \
  u_l1tag_ram \
  u_l1data_ram \
]

# The memory hard IPs instantiated in the CPU support power gating
# and retention. The memory has two primary power ports, VDDPE and VDDCE.
# To make supply connection simple, two new supply set handles mem_vddc
# and mem_vddp are updated to PDCPU. mem_vddc will be associated to
# VDDCE of all the memories and mem_vddp will be associated to VDDPE
# of all the memories in the CPU.
create_power_domain PDCPU -update  \
  -supply {mem_vddc} \
  -supply {mem_vddp}

# Apply power model to each macro instance.
# UPF power model is overlaid over the Liberty model of the memory.
# In applying power model, the supply sets defined in the memory
# power model are associated to the supply set handles defined in the
# power domain.
foreach instance $l1MemInstances {
  apply_power_model memPwrModel -elements $instance \
    -supply_map {{$instance/ss_vddpe PDCPU.mem_vddp} \
                 {$instance/ss_vddce PDCPU.mem_vddc}}
}

# When CPU is in OFF state, all memories shall be in OFF state.
# When CPU is in RET state, all memories shall be in RET state.
# When CPU is in ON state, memories can be in one of ON, RET, OFF
# state but all memories shall be in the same state.
# To enforce power state over a collection of lower level instances,
# create a group and set power state of the group based on the
# power state of the lower level instances.
create_power_state_group L1MEMS

# Create power states for the group created.
# Power state of the group is such that each of the instance
# in the group are in the same power state as the group itself.
add_power_state -group L1MEMS \
  -state {ON  -logic_expr {u_l1tag_ram==ON  && u_l1data_ram==ON }} \
  -state {RET -logic_expr {u_l1tag_ram==RET && u_l1data_ram==RET}} \
  -state {OFF -logic_expr {u_l1tag_ram==OFF && u_l1data_ram==OFF}}

# Update power state of the parent domain with the power state of
# the group. When CPU is in ON state, the L1MEMS can be in one of ON,
# RET or OFF.
# Use of logic_expr {L1MEMS==ON || L1MEMS==RET || L1MEMS==OFF} would
# result in an indefinite state that cannot be used to update higher
# level states. To avoid indefinite state, the different ON states
# are implemented as refinement of PDCPU ON state.
# The hierarchical state names ON.L1ON, ON.L1RET, and ON.L1OFF implicitly
# include the term PDCPU==ON.
add_power_state PDCPU -domain -update \
  -state {ON.L1ON  -logic_expr {L1MEMS==ON }} \
  -state {ON.L1RET -logic_expr {L1MEMS==RET}} \
  -state {ON.L1OFF -logic_expr {L1MEMS==OFF}} \
  -state {RET      -logic_expr {L1MEMS==RET}} \
  -state {OFF      -logic_expr {L1MEMS==OFF}}

# Power State logic_expr Update for Supply Sets
# The logic expression that defines the ON and OFF state of the
# supply set is updated.
# ON and OFF state are deferred power states that are now being
# updated. Since the pre-defined deferred power states are being
# updated -update is required.
add_power_state -supply PDCPU.primary -update \
  -state {ON  -logic_expr {lp_lSleep == 1}}\
  -state {OFF -logic_expr {lp_lSleep == 0}}
```

E.3.4.4 Implementation UPF

The CPU is hardened in the implementation and so an implementation UPF is required. The implementation UPF defines the terminal boundary conditions for the CPU. The terminal boundary conditions define the external environment conditions that are assumed when the CPU is hardened. The external environment conditions defined for hardening may not reflect the exact context in which the CPU hardened macro will be used in the MPCORE. In defining the external conditions for the CPU implementations, the MPCORE context isn't available and hence the boundary conditions are defined in terms of CPU context.

```
# Supply Ports, Supply Nets
# VCPU is the supply voltage for CPU
create_supply_port -direction in VCPU
create_supply_port -direction in VRET
create_supply_port -direction in VSS

create_supply_net VCPU
create_supply_net VRET
create_supply_net VSS
create_supply_net VCPU_sw

connect_supply_net VCPU -ports VCPU
connect_supply_net VRET -ports VRET
connect_supply_net VSS  -ports VSS

# Create supply set handle aon in the PDCPU domain.
create_power_domain PDCPU -update  \
  -supply {aon}
```

```
# Associate supply nets to supply set functions
create_supply_set PDCPU.aon \
  -function {power VCPU} \
  -function {ground VSS}

create_supply_set PDCPU.primary -update \
  -function {power VCPU_sw} \
  -function {ground VSS}

create_supply_set PDCPU.ret -update \
  -function {power VRET} \
  -function {ground VSS}

# Power Switches
create_power_switch sw_CPU \
  -input_supply_port  {sw_in     PDCPU.aon.power} \
  -output_supply_port {sw_out    PDCPU.primary.power} \
  -control_port       {sw_ctrl   lp_lSleep} \
  -on_state           {on_state  sw_in {sw_ctrl}} \
  -off_state          {off_state {!sw_ctrl}} \
  -supply_set         PDCPU.aon \
  -domain PDCPU

# Associate supply nets to memory supply set functions.
# This association connects all the memory VDDCE, VDDPE supply ports
# to VCPU supply port, and memory VSSE supply port to VSS.
# Explicit connection of supply ports using connect_supply_net can
# also be done.
# The standard allows both supply association and supply net
# connection to specified together.
create_supply_set PDCPU.mem_vddp -update \
  -function {power VCPU} \
  -function {ground VSS}

create_supply_set PDCPU.mem_vddc -update \
  -function {power VCPU} \
  -function {ground VSS}

# Power State supply_expr Update for Supply Sets
add_power_state -supply PDCPU.primary -update \
  -state {ON  {-supply_expr \
              {power=={FULL_ON 0.7} && ground=={FULL_ON 0}}}} \
  -state {OFF {-supply_expr \
              {power==OFF           && ground=={FULL_ON 0}}}}

add_power_state -supply PDCPU.ret -update \
  -state {ON  {-supply_expr \
              {power=={FULL_ON 0.7} && ground=={FULL_ON 0}}}} \
  -state {OFF {-supply_expr \
              {power==OFF           && ground=={FULL_ON 0}}}}

add_power_state -supply PDCPU.aon -update \
  -state {ON  {-supply_expr \
              {power=={FULL_ON 0.7} && ground=={FULL_ON 0}}}}

# The VRET voltage is derived from the VCPU voltage within the SOC.
# But at the CPU implementation, this derived supply information
# need to be specified as supplies VCPU and VRET are correlated.
set_correlated -sets PDCPU.primary PDCPU.ret

# VCPU, VRET supplies support voltage scaling from 0.63 to 0.77
# With 0.7 V being nominal, the scaling factor {0.63/0.7 0.77/0.7}
set_variation -supply {PDCPU.primary PDCPU.ret PDCPU.aon} -range {0.9 1.1}
```

```
# No level-shifters are required in implementation of PDCPU.
# The supplies input to CPU are VCPU and VRET that have same voltage
# conditions and are correlated.

# Terminal boundary model for CPU hardened macro
# Note that tools will ignore the driver/receiver attributes
# when the actual driver and receiver for the ports are available
# in the design.

# The external environment conditions are required for the implementation
# of CPU as a soft macro
# The external environment conditions are defined based on the
# supply sets available in PDCPU
set_port_attributes -model {.} -ports \
  [find_objects . -pattern * -object_type port -direction in] \
  -exclude_ports {lp_lSleep lp_rSleep lp_lRet lp_rRet} \
  -driver_supply PDCPU.primary

set_port_attributes -model {.} -ports \
  {lp_lSleep lp_rSleep lp_lRet lp_rRet} -driver_supply PDCPU.aon

set_port_attributes -model {.} -ports \
  [find_objects . -pattern * -object_type port -direction out] \
  -receiver_supply PDCPU.primary

# When the UPF for CPU is loaded in MPCORE implementation context,
# the internals of PDCPU are not available as the CPU has been
# hardened and treated as a leaf cell. In the higher implementation
# context, the CPU internal environment attributes are required.
set_port_attributes -model {.} -ports \
  [find_objects . -pattern * -object_type port -direction out] \
  -driver_supply PDCPU.primary

set_port_attributes -model {.} -ports \
  [find_objects . -pattern * -object_type port -direction in] \
  -exclude_ports {lp_lSleep lp_rSleep lp_lRet lp_rRet} \
  -receiver_supply PDCPU.primary

set_port_attributes -model {.} -ports \
  {lp_lSleep lp_rSleep lp_lRet lp_rRet} \
  -receiver_supply PDCPU.aon
```

E.3.5 UPF for MPCORE

E.3.5.1 Introduction

The MPCORE UPF loads the constraint, configuration UPF for mpcore. Since MPCORE is a soft IP, no implementation UPF is required. The UPF should be complete from a simulation context.

```
set_design_top mpcore
source mpcore_constraints.upf
source mpcore_configuration.upf
```

E.3.5.2 Constraint UPF

The MPCORE is a soft IP and has a constraint UPF provided with the HDL of the IP.

```
# Variable Declarations
# List of MPCORE output ports to be clamped 1 when MPCORE is OFF.
# This list shall be used to define port attributes on the ports that
# shall determine the clamp value for isolation policy.
set coreClamp1 [list \
  IRQn \
  FIQn \
  L2ACCEPTn \
]

# Power Domains
# MPCORE consists of only one power domain PDMPCORE. The current scope
# is the domain top.
# Two supply set handles, primary and ret, are defined in PDMPCORE:
# primary is the supply as defined in the standard;
# ret is the supply used as the back-up supply for the retention FFs.

create_power_domain PDMPCORE -elements {.} \
  -supply {primary} \
  -supply {ret}

# Port Attributes for Isolation
# All the output ports except the list specified by $coreClamp1 are
# set to clamp value 0.
# Ports specified by $coreClamp1 are set to clamp value 1.

set_port_attributes -model {.} -ports \
  [find_objects . -pattern * -object_type port -direction out] \
  -exclude_ports ${coreClamp1}  -clamp_value 0
set_port_attributes -model {.} -ports "$coreClamp1" -clamp_value 1

# Load UPF for Lower Level IPs
# The UPF of CPU that is hardened is loaded.
# Note:
# If UPF_is_soft_macro attribute is set for the cpu, only the higher
# power domain is loaded ignoring all internals of the UPF. The terminal
# boundary conditions defined in the CPU UPF are used to model the
# internals of the CPU.
# If UPF_is_soft_macro attribute is not set for the cpu, the cpu UPF
# is used to completely model the power intent of cpu.
# The constraints for MPCORE does not have information whether the cpu
# is a soft_macro or not.
load_upf cpu.upf -scope u_cpu0
load_upf cpu.upf -scope u_cpu1

# Retention Elements
# The list of instances that can be retained if the MPCORE
# is put into retention mode is specified in the coreRetList.
# This retention list shall be used in the retention policy specified
# in the configuration UPF.
set_retention_elements coreRetList \
  -elements {u_core_noram}

# Power State
# PDMPCORE: The PDMPCORE supports three power states: ON, RET, OFF.
# In the ON state, supply sets primary and ret are ON.
# In the RET state, supply set ret is ON and primary is OFF.
# In the OFF state, both supply sets are OFF.
# Note:
# The ON and OFF states of the supply sets are the deferred power
# states defined by the standard.
add_power_state -domain PDMPCORE \
-state {ON  -logic_expr{PDMPCORE.ret==ON  && PDMPCORE.primary==ON }}\
-state {RET -logic_expr{PDMPCORE.ret==ON  && PDMPCORE.primary==OFF}}\
-state {OFF -logic_expr{PDMPCORE.ret==OFF && PDMPCORE.primary==OFF}}

# In the RET and OFF power state, the cpu0 and cpu1 shall be in OFF
# state. Update PDMPCORE state with state of PDCPU.
add_power_state -domain PDMPCORE -update \
  -state {RET  -logic_expr {cpu0/PDCPU==OFF}} \
  -state {OFF  -logic_expr {cpu0/PDCPU==OFF}}

add_power_state -domain PDMPCORE -update \
  -state {RET  -logic_expr {cpu1/PDCPU==OFF}} \
  -state {OFF  -logic_expr {cpu1/PDCPU==OFF}}
```

E.3.5.3 Configuration UPF

The MPCORE is a soft IP and the UPF shall be complete in verification context to enable standalone verification of MPCORE. To enable verification of MPCORE, the configuration UPF should have the low power (LP) control ports.

```
# LP Control Ports
create_logic_port lp_lSleep
create_logic_port lp_rSleep
create_logic_port lp_lReady
create_logic_port lp_rReady

create_logic_port lp_lSleepCpu0
create_logic_port lp_lSleepCpu1
create_logic_port lp_rSleepCpu0
create_logic_port lp_rSleepCpu1
create_logic_port lp_lRetCpu0
create_logic_port lp_lRetCpu1
create_logic_port lp_rRetCpu0
create_logic_port lp_rRetCpu1
create_logic_port lp_lReadyCpu0
create_logic_port lp_lReadyCpu1

create_logic_port lp_rReadyCpu0
create_logic_port lp_rReadyCpu1
create_logic_port lp_isoCpu0
create_logic_port lp_isoCpu1

# Connect lower level controls
# The control nets to the lower level CPU macro are connected
connect_logic_net lp_lSleepCpu0 -ports u_cpu0/lp_lSleep
connect_logic_net lp_lSleepCpu1 -ports u_cpu1/lp_lSleep
connect_logic_net lp_rSleepCpu0 -ports u_cpu0/lp_rSleep
connect_logic_net lp_rSleepCpu1 -ports u_cpu1/lp_rSleep
connect_logic_net lp_lReadyCpu0 -ports u_cpu0/lp_lReady
connect_logic_net lp_lReadyCpu1 -ports u_cpu1/lp_lReady
connect_logic_net lp_rReadyCpu0 -ports u_cpu0/lp_rReady
connect_logic_net lp_rReadyCpu1 -ports u_cpu1/lp_rReady
connect_logic_net lp_lRetCpu0   -ports u_cpu0/lp_lRet
connect_logic_net lp_lRetCpu1   -ports u_cpu1/lp_lRet
connect_logic_net lp_rRetCpu0   -ports u_cpu0/lp_rRet
connect_logic_net lp_rRetCpu1   -ports u_cpu1/lp_rRet

# Design attribute UPF_is_soft_macro is set on cpu0 and cpu1 instance to
# enable the correct terminal boundary condition as the CPU has been
# hardened in the implementation.
# In the verification context, if the actual drivers and receivers
# across the port are available, the port attributes defined in the
# terminal boundary shall be ignored.
set_design_attributes -model {cpu} -is_soft_macro TRUE

# Retention Strategy
# From the SoC functional power states, MPCORE is configured to NOT
# use RET state defined in constraint UPF for the logic.
# The memories can be in retention state when MPCORE is in ON state.
# The RET state of PDMPCORE is defined as illegal in this
# configuration of the MPCORE.
add_power_state -domain PDMPCORE -update \
  -state {RET -illegal}

# The PDMPCORE.ret supply set is associated to PDMPCORE.primary since the
# ON and OFF power states of PDMPCORE power domain have been defined as
# a function of PDMPCORE.ret in addition to PDMPCORE.primary
associate_supply_set { PDMPCORE.primary PDMPCORE.ret }

# Memory Related Configuration
# List of memory instances in MPCORE
set l2MemInstances [list \
  u_l2tag_ram \
  u_l2data_ram \
]

# Two new supply set handles mem_vddc and mem_vddp are updated to
# PDMPCORE. mem_vddc will be associated to
# VDDCE of all the memories and mem_vddp will be associated to VDDPE
# of all the memories in the CPU.
create_power_domain PDCPU -update  \
  -supply {mem_vddc} \
  -supply {mem_vddp}

# Similar to the L1 memories in CPU configuration UPF, the power
# model for L2 memory is applied.
# Create a group of macros that should be in the same state
# at any given time.

# Apply power model to each macro instance
foreach instance $l2MemInstances {
  apply_power_model memPwrModel2 -elements $instance \
    -supply_map {{$instance/ss_vddpe PDMPCORE.mem_vddp} \
                 {$instance/ss_vddce PDMPCORE.mem_vddc}}
}

# When MPCORE is in OFF state, all memories shall be in OFF state.
# When MPCORECPU is in RET state, all memories shall be in RET state.
# When MPCORE is in ON state, memories can be in ON or OFF state but
# all memories shall be in the same state.
# To enforce power state over a collection of lower level instances,
# create a group and set power state of the group based on the
# power state of the lower level instances.
create_power_state_group L2MEMS

# Create power states for the group created.
# Power state of the group is such that each of the instance
# in the group are in the same power state as the group itself.
add_power_state -group L2MEMS \
  -state {ON  -logic_expr {u_l2tag_ram==ON  && u_l2data_ram==ON }} \
  -state {OFF -logic_expr {u_l2tag_ram==OFF && u_l2data_ram==OFF}}

# Update power state of PDMPCORE with the power state of the group
# when MPCORE is in ON state, L2MEMS can be in ON/OFF state.
# It is illegal for L2MEMS to be in NOT OFF state when MPCORE is OFF.
```

```
add_power_state -domain PDMPCORE -update \
  -state {ERR1 -logic_expr {PDMPCORE==OFF && L2MEMS!=OFF} -illegal}

# Power State logic_expr Update for Supply Sets
# The logic expression that defines the ON and OFF state of the
# supply set is updated.
# ON and OFF state are deferred power states that are now being
# updated. Since the deferred power states are being updated -update
# is required.
add_power_state -supply PDMPCORE.primary -update \
  -state {ON  -logic_expr {lp_lSleep==1}}\
  -state {OFF -logic_expr {lp_lSleep==0}}

# Isolation Policy for Path That Have CPU as Source
# The port attributes specified in the CPU constraint UPF determine
# the clamp value for the ports.
set_isolation isoCpu0ToMpcore \
   -domain PDMPCORE \
   -applies_to inputs \
   -source u_cpu0/PDCPU.primary \
   -isolation_signal lp_isoCpu0 \
   -isolation_sense low

set_isolation isoCpu1ToMpcore \
   -domain PDMPCORE \
   -applies_to inputs \
   -source u_cpu1/PDCPU.primary \
   -isolation_signal lp_isoCpu1 \
   -isolation_sense low
```

E.3.6 UPF for display controller

E.3.6.1 Introduction

The display core is a soft IP and has its constraint and configuration UPF. The display UPF should be complete from a simulation context.

```
set_design_top display

source display_constraints.upf
source display_configuration.upf
```

E.3.6.2 Constraint UPF

```
# Variable Declarations
# List of display output ports to be clamped high when display is OFF.
set dispClamp1 [list \
  ready \
]

# Power Domains
# Display IP consists of one power domain PDDISP.
# Two supply set handles primary and aon are defined.
# primary is the supply as defined in the standard.
# aon is the unswitched supply in the display context.
create_power_domain PDDISP -elements {.} \
  -supply {primary} \
  -supply {aon}
```

```
# Port Attributes for Isolation
# All the output ports except the list specified by $dispClamp1 are
# set to clamp value 0.
# Ports specified by $dispClamp1 are set to clamp value 1.
set_port_attributes -ports \
  [find_objects . -pattern * -object_type port -direction out] \
  -exclude_elements $dispClamp1 -clamp_value 0
set_port_attributes -model -ports $dispClamp1 -clamp_value 1

# Note:
# In the display constraints, the power intent details of PHY is unknown.
# The power intent of PHY is handled in the configuration UPF.

# Power State PDDISP
# supply aon cannot be in OFF state.
# The deferred power state OFF is set as illegal for aon supply set.
# PDDISP supports ON and OFF states.
add_power_state -supply PDDISP.aon -update \
  -state {OFF -illegal}

add_power_state -domain PDDISP \
  -state {ON  -logic_expr {PDDISP.aon==ON && PDDISP.primary==ON }} \
  -state {OFF -logic_expr {PDDISP.aon==ON && PDDISP.primary==OFF}}
```

E.3.6.3 Configuration UPF

```
# LP Control Ports in RTL
create_logic_port lp_regCtrl

# PHY Related Configuration
# Load the power model of PHY.
# The supplies defined in the parent power model are associated to
# the supplies defined in the power model.
apply_power_model phyPwrModel -elements u_phy \
  -supply_map {{PDREG.ss_regin  PDDISP.aon} \
               {PDREG.ss_regout PDDISP.primary}}

# Connect control net to the PHY hard macro.
connect_logic_net lp_regCtrl -ports u_phy/regCtrl

# Update power state of PDDISP with state of PDPHY.
# PDPHY will be ON when PDDISP is ON.
# PDPHY will be OFF when PDDISP is OFF.
add_power_state -domain PDDISP -update \
  -state {ON  -logic_expr {u_phy/PDPHY==ON }} \
  -state {OFF -logic_expr {u_phy/PDPHY==OFF}}
```

E.3.7 UPF for SoC

E.3.7.1 Introduction

The SoC is the top-level module and has the constraint, configuration, and implementation UPF. In the context of the SoC, the constraint and configuration UPF can be merged into configuration as the SoC does not require the constraints and configuration to be separate.

```
set_design_top soc
source soc_constraints.upf
source soc_configuration.upf
source soc_implementation.upf
```

E.3.7.2 Constraint UPF

```
# Variable Declarations
# List of ports that have sink in the PDAON
set socAonInputs [list \
  SYSPLLCLK \
  PORESETn \
  TDI \
  nTRST \
]

# Power Domains
# SoC has two power domains: PDSOC and PDAON.
# primary is the supply as defined in the standard.
# ret is the supply used as the back-up supply for the retention FFs.
create_power_domain PDSOC \
  -elements {.} \
  -supply {primary} \
  -supply {aon}

create_power_domain PDAON \
  -elements u_pcu  \
  -supply {primary}

# Load Lower Level UPF of the Design
# The lower level UPFs may include soft IPs, hard macro, and soft macro.
# In this example, there are two soft IPs. Each of the soft IPs in this
# example includes soft marco and/or hard macro that have their own UPF.
load_upf mpcore.upf  -scope u_mpcore
load_upf display.upf -scope u_display

# Power States - AON
# The pre-defined deferred state OFF for supply set primary is defined
# as illegal as this supply cannot be OFF.
add_power_state -supply PDAON.primary -update \
  -state {OFF  -illegal}

add_power_state -domain PDAON \
  -state {ON  -logic_expr {PDAON.primary==ON}} \

# Power States - SOC
# The pre-defined deferred state OFF for supply set aon is defined
# as illegal as this supply cannot be OFF.
add_power_state -supply PDSOC.aon -update \
  -state {OFF -illegal}

add_power_state -domain PDSOC \
  -state {RUN   -logic_expr {PDSOC.aon==ON && PDSOC.primary==ON } \
  -state {SLEEP -logic_expr {PDSOC.aon==ON && PDSOC.primary==OFF}

# When PDSOC is in RUN or SLEEP state, PDAON must be ON.
add_power_state -domain PDSOC -update \
  -state {RUN   -logic_expr {PDAON==ON}} \
  -state {SLEEP -logic_expr {PDAON==ON}}

# When PDSOC is in SLEEP state, PDMPCORE and PDDISP must be OFF.
add_power_state -domain PDSOC -update \
  -state {SLEEP -logic_expr {u_mpcore/PDMPCORE==OFF && \
                             u_display/PDDISP==OFF}}
```

E.3.7.3 Configuration UPF

The SoC configuration defines the complete configuration of the SoC and the connections to the power controls signals for isolation, retention, and power switch.

```
# LP Control nets to make connections from power control unit to
# hard/soft macros
create_logic_net lp_lSleep
create_logic_net lp_lSleepCpu0
create_logic_net lp_lSleepCpu1
create_logic_net lp_lSleepDisp
create_logic_net lp_rSleepCpu0
create_logic_net lp_rSleepCpu1
create_logic_net lp_lRetCpu0
create_logic_net lp_lRetCpu1
create_logic_net lp_rRetCpu0
create_logic_net lp_rRetCpu1
create_logic_net lp_isoCpu0
create_logic_net lp_isoCpu1
create_logic_net lp_isoDisp
create_logic_net lp_isoAon
create_logic_net lp_lSleepL2
create_logic_net lp_rSleepL2
create_logic_net lp_rReadyL2
create_logic_net lp_rRetL2

# Connection of the power control ports of macros to the power control
# unit are explicitly done.
connect_logic_net lp_lSleepCpu0 \
  -ports u_pcu/LSLEEPCPU0 u_mpcore/lp_lSleepCpu0
connect_logic_net lp_lSleepCpu1 \
  -ports u_pcu/LSLEEPCPU1 u_mpcore/lp_lSleepCpu1
connect_logic_net lp_rSleepCpu0 \
  -ports u_pcu/RSLEEPCPU0 u_mpcore/lp_rSleepCpu0
connect_logic_net lp_rSleepCpu1 \
  -ports u_pcu/RSLEEPCPU1 u_mpcore/lp_rSleepCpu1
connect_logic_net lp_lRetCpu0   \
  -ports u_pcu/LRETCPU0 u_mpcore/lp_lRetCpu0
connect_logic_net lp_lRetCpu1   \
  -ports u_pcu/LRETCPU1 u_mpcore/lp_lRetCpu1
connect_logic_net lp_rRetCpu0   \
  -ports u_pcu/RRETCPU0 u_mpcore/lp_rRetCpu0
connect_logic_net lp_rRetCpu1   \
  -ports u_pcu/RRETCPU1 u_mpcore/lp_rRetCpu1
connect_logic_net lp_isoCpu0    \
  -ports u_pcu/ISOCPU0 u_mpcore/lp_isoCpu0
connect_logic_net lp_isoCpu1    \
  -ports u_pcu/ISOCPU1 u_mpcore/lp_isoCpu1
connect_logic_net lp_lSleepL2   \
  -ports u_pcu/LSLEEPL2 u_mpcore/lp_lSleep
connect_logic_net lp_rSleepL2   \
  -ports u_pcu/RSLEEPL2 u_mpcore/lp_rSleep
connect_logic_net lp_rReadyL2   \
  -ports u_pcu/RREADYL2 u_mpcore/lp_rReady
connect_logic_net lp_rRetL2     \
  -ports u_pcu/RRETL2 u_mpcore/lp_rRet
connect_logic_net lp_lSleep     -ports u_pcu/LSLEEP
connect_logic_net lp_lSleepDisp -ports u_pcu/LSLEEPDISP
connect_logic_net lp_isoDisp    -ports u_pcu/ISODISP
connect_logic_net lp_rSleepL2   -ports u_pcu/RSLEEPL2
connect_logic_net lp_rRetL2     -ports u_pcu/RRETL2
connect_logic_net lp_isoAon     -ports u_pcu/ISOAON
```

```
# Isolation Strategy
# AON - Isolation for all inputs of PDAON that have sink in PDAON
set_isolation isoAonIn0 \
  -domain PDAON \
  -applies_to inputs \
  -sink PDAON.primary \
  -clamp_value 0 \
  -isolation_signal ln_isoAon \
  -isolation_sense high \

# Display - Isolation for Display Outputs
set_isolation isoDispOut \
  -domain PDDISP \
  -source PDDISP.primary \
  -sink PDSOC.primary \
  -isolation_signal lp_lIsoDisp \
  -isolation_sense low

# MPCORE - Isolation for mpcore Outputs
set_isolation isoMpcoreOut \
  -domain u_mpcore/PDMPCORE \
  -source u_mpcore/PDMPCORE.primary \
  -sink PDSOC.primary \
  -isolation_signal lp_lIsoMpcore \
  -isolation_sense low

# Power State
add_power_state -supply PDSOC.primary -update \
  -state {ON  -logic_expr {ln_lSleep==0}} \
  -state {OFF -logic_expr {ln_lSleep==1}}
```

E.3.7.4 Implementation UPF

```
# Supply Ports, Supply Nets
create_supply_port -direction in VSOC
create_supply_port -direction in VCPU
create_supply_port -direction in VDDI
create_supply_port -direction in VSS

create_supply_net  VSOC
create_supply_net  VCPU
create_supply_net  VDDI
create_supply_net  VSS
create_supply_net  VSOC_sw
create_supply_net  VMP_sw

connect_supply_net VSOC -ports {VSOC}
connect_supply_net VCPU -ports {VCPU}
connect_supply_net VDDI -ports {VDDI}
connect_supply_net VSS  -ports {VSS}

# Power Switches
# VSOC
create_power_switch sw_VSOC \
  -input_supply_port  {sw_in     VSOC} \
  -output_supply_port {sw_out    VSOC_sw} \
  -control_port       {sw_ctrl   ln_lSleep} \
  -on_state           {on_state  sw_in {sw_ctrl}} \
  -off_state          {off_state {!sw_ctrl}} \
  -supply_set         PDVSOC.aon \
  -domain PDVSOC
# MPCORE
```

```
create_power_switch sw_MPCORE \
  -input_supply_port  {sw_in     VCPU} \
  -output_supply_port {sw_out    VMP_sw} \
  -control_port       {sw_ctrl   ln_1SleepMpcore} \
  -on_state           {on_state  sw_in {sw_ctrl}} \
  -off_state          {off_state {!sw_ctrl}} \
  -supply_set         u_mpcore/PDMPCORE.aon \
  -domain PDVSOC

# aon Supply for MPCORE
create_power_domain u_mpcore/PDMPCORE -update  \
  -supply {aon}

# Associate Supply Nets to Supply Set Functions
create_supply_set u_mpcore/PDMPCORE.aon \
  -function {power VCPU} \
  -function {ground VSS}

create_supply_set u_mpcore/PDMPCORE.primary -update \
  -function {power VMP_sw} \
  -function {ground VSS}

create_supply_set PDVSOC.primary -update \
  -function {power  VSOC_sw} \
  -function {ground VSS}

create_supply_set PDVSOC.aon -update \
  -function {power  VSOC} \
  -function {ground VSS}

create_supply_set PDAON.primary -update \
  -function {power  VSOC} \
  -function {ground VSS}

# Connect Power Supplies of Lower Level Macros
connect_supply_net VCPU   -ports u_mpcore/u_cpu0/VCPU
connect_supply_net VCPU   -ports u_mpcore/u_cpu1/VCPU

connect_supply_net VSS    -ports u_mpcore/u_cpu0/VSS
connect_supply_net VSS    -ports u_mpcore/u_cpu1/VSS

connect_supply_net VMP_sw -ports u_mpcore/u_cpu0/VRET
connect_supply_net VMP_sw -ports u_mpcore/u_cpu1/VRET

create_supply_set u_mpcore/PDMPCORE.mem_vddp -update \
  -function {power VCPU} \
  -function {ground VSS}

create_supply_set u_mpcore/PDMPCORE.mem_vddc -update \
  -function {power VCPU} \
  -function {ground VSS}

connect_supply_net VDDI -ports u_display/u_phy/VDDI
connect_supply_net VSS  -ports u_display/u_phy/VSS

# Update Isolation Policy with -location and –isolation Supply Set
# AON
set_isolation isoAonIn0 \
  -domain PDAON \
  -isolation_supply PDAON.primary \
  -location self

# Display
```

```
set_isolation isoDispOut \
  -domain PDDISP \
  -isolation_supply PDSOC.primary \
  -location parent

# MPCORE
set_isolation isoMpcoreOut \
  -domain u_mpcore/PDMPCORE \
  -isolation_supply PDSOC.primary \
  -location parent

# Level Shifting Policy for Paths from SoC to CPU
set_level_shifter lsMpIn \
  -domain PDSOC \
  -source PDSOC.primary \
  -sink u_mpcore/PDMPCORE.primary \
  -location self

set_level_shifter lsCpu0In \
  -domain PDSOC \
  -source PDSOC.primary \
  -sink u_mpcore/u_cpu0/PDCPU.primary \
  -location self

set_level_shifter lsCpu1In \
  -domain PDSOC \
  -source PDSOC.primary \
  -sink u_mpcore/u_cpu1/PDCPU.primary \
  -location self

# Level Shifting Policy for Paths to SoC from cpu/mpcore
set_level_shifter lsMpOut \
  -domain PDSOC \
  -source u_mpcore/PDMPCORE.primary \
  -sink PDSOC.primary \
  -location self

set_level_shifter lsCpu0In \
  -domain PDSOC \
  -source u_mpcore/u_cpu0/PDCPU.primary \
  -sink PDSOC.primary \
  -location self

set_level_shifter lsCpu1In \
  -domain PDSOC \
  -source u_mpcore/u_cpu1/PDCPU.primary \
  -sink PDSOC.primary \
  -location self

# Power State Supply_expr Update for Supply Sets
# AON
add_power_state -supply PDAON.primary -update \
  -state {ON -supply_expr \
             {power=={FULL_ON 0.8} && ground=={FULL_ON 0}}}

# VSOC
add_power_state -supply PDVSOC.primary -update \
  -state {ON  -supply_expr \
              {power=={FULL_ON 0.8} && ground=={FULL_ON 0}}} \
  -state {OFF -supply_expr \
              {power==OFF           && ground=={FULL_ON 0}}}

add_power_state -supply PDVSOC.aon -update \
  -state {ON  -supply_expr \
              {power=={FULL_ON 0.8} && ground=={FULL_ON 0}}}

# PDDISP
add_power_state -supply PDDISP.primary -update \
  -state {ON  -supply_expr \
              {power=={FULL_ON 0.8} && ground=={FULL_ON 0}}} \
  -state {OFF -supply_expr \
              {power==OFF           && ground=={FULL_ON 0}}}

# PDMPCORE
add_power_state -supply u_mpcore/PDMPCORE.primary -update \
  -state {ON  -supply_expr \
              {power=={FULL_ON 0.7} && ground=={FULL_ON 0}}} \
  -state {OFF -supply_expr \
              {power==OFF           && ground=={FULL_ON 0}}}

set_variation -supply {u_mpcore/PDMPCORE.primary} -range {0.9 1.1}

set_port_attributes -ports $socAonInputs \
  -driver_supply PDSOC.aon
```

Annex F

(informative)

Power-management cell definitions in UPF and Liberty

F.1 Introduction

F.1.1 Overview

This annex describes how the information specified in each power-management cell command (see Clause 7) can be used by the corresponding power intent commands in Clause 6. In addition, it also describes the mapping between each command and option to the Liberty attributes. Unless otherwise stated, the referenced Liberty attributes are based on the Liberty 2009.06 release (see Liberty library format usage [B4]). For designers who prefer to use the Liberty approach to describe power-management cell attributes, the mapping tables in this annex can be used to understand what the required information is in Liberty to enable a UPF flow.

F.1.2 Liberty attribute mapping

If a UPF option has a corresponding Liberty attribute, the following type of mapping table (see Table F.1) is used:

Table F.1—Sample Liberty attribute mapping

Name	**Group**	**Type**	**Value**
pg_type	pg_pin	string	primary_ground

Where the column *Name* lists the corresponding Liberty attribute name; the column *Group* indicates the name of the group statement in which this attribute is specified; the column *Type* indicates the attribute type such as a string, Boolean, integer, or floating point; and the column *Value* indicates the corresponding attribute value.

If a UPF option has no corresponding Liberty attribute, this is indicated explicitly.

F.1.3 Potential conflicts with library command definitions

These mappings are based on the syntax from the actual library command definitions (see Clause 7), which are replicated in this annex as a convenience. In the event of a conflict between this material and the syntax shown in Clause 7, the syntax listing for Clause 7 shall prevail.

F.2 define_always_on_cell

define_always_on_cell [from 7.2]
    -cells *cell_list*
    -power *pin*

-ground *pin*
[**-power_switchable** *pin*] [**-ground_switchable** *pin*]
[**-isolated_pins** *list_of_pin_lists*][**-enable** *expression_list*]

The Liberty mappings for this command are as follows:

a) Table F.2 indicates the Liberty attribute mapping for all cells identified by the **-cells** option of this command.

Table F.2—Liberty attribute mapping for -cells

Name	**Group**	**Type**	**Value**
always_on	cell	Boolean	true

b) Table F.3 indicates the Liberty attribute mapping for the **-power** argument.

Table F.3—Liberty attribute mapping for -power

Name	**Group**	**Type**	**Value**
pg_type	pg_pin	string	backup_power primary_power

1) If this option is specified with **-power_switchable**, the corresponding *pg_type* is **backup_power**. During implementation, this pin is connected to the ground net specified by users.

2) If this option is not specified with **-power_switchable**, the corresponding *pg_type* is **primary_power**. During implementation, this pin is connected to the ground net of the primary supply set of the power domain in which the cell is located.

c) Table F.4 indicates the Liberty attribute mapping for the **-ground** argument.

Table F.4—Liberty attribute mapping for -ground

Name	**Group**	**Type**	**Value**
pg_type	pg_pin	string	backup_ground primary_ground

1) If this option is specified with **-ground_switchable**, the corresponding *pg_type* is **backup_ground**. During implementation, this pin is connected to the ground net specified by users.

2) If this option is not specified with **-ground_switchable**, the corresponding *pg_type* is **primary_ground**. During implementation, this pin is connected to the ground net of the primary supply set of the power domain in which the cell is located.

d) Table F.5 indicates the Liberty attribute mapping for the **-power_switchable** argument.

Table F.5—Liberty attribute mapping for -power_switchable

Name	Group	Type	Value
pg_type	pg_pin	string	primary_power

1) During implementation, this pin is connected to the power net of the primary supply set of the power domain in which the cell is located.

e) Table F.6 indicates the Liberty attribute mapping for the -ground_switchable argument.

Table F.6—Liberty attribute mapping for -ground_switchable

Name	Group	Type	Value
pg_type	pg_pin	string	primary_ground

1) During implementation, this pin is connected to the ground net of the primary supply set of the power domain in which the cell is located.

f) **-isolated_pins** has no corresponding Liberty attribute.

g) **-enable** has no corresponding Liberty attribute.

F.3 define_diode_clamp

define_diode_clamp [from 7.3]
  -cells *cell_list*
  -data_pins *pin_list*
  [**-type** <**power** | **ground** | **both**>]
  [**-power** *pin*] [**-ground** *pin*]

The Liberty mappings for this command are as follows:

a) Table F.7 indicates the Liberty attribute mapping for all cells identified by the **-cells** option of this command.

Table F.7—Liberty attribute mapping for -cells

Name	Group	Type	Value
antenna_diode_type	cell	Boolean	true

b) **-data_pins** has no corresponding Liberty attribute.

c) **-type** has no corresponding Liberty attribute.

d) Table F.8 indicates the Liberty attribute mapping for the **-power** argument.

Table F.8—Liberty attribute mapping for -power

Name	Group	Type	Value
pg_type	pg_pin	string	primary_power

e) Table F.9 indicates the Liberty attribute mapping for the **-ground** argument.

Table F.9—Liberty attribute mapping for -ground

Name	Group	Type	Value
pg_type	pg_pin	string	primary_ground

F.4 define_isolation_cell

define_isolation_cell [from 7.4]
  -cells *cell_list*
  [**-power** *power_pin*]
  [**-ground** *power_pin*]
  {**-enable** *pin* [**-clamp_cell** <**high** | **low**>]
  | **-pin_groups** **{**{*input_pin output_pin* [*enable_pin*]**}***}**
  | **-no_enable** <**high** | **low** | **hold**>}
  [**-always_on_pins** *pin_list*]
  [**-aux_enables** *ordered_pin_list*]
  [**-power_switchable** *power_pin*] [**-ground_switchable** *ground_pin*]
  [**-valid_location** <**source** | **sink** | **on** | **off** | **any**>]
  [**-non_dedicated**]

The Liberty mappings for this command are as follows:

a) Table F.10 indicates the Liberty attribute mapping for all cells identified by the **-cells** option of this command.

Table F.10—Liberty attribute mapping for -cells

Name	Group	Type	Value
is_isolation_cell	cell	Boolean	true

b) Table F.11 and Table F.12 indicate the Liberty attribute mapping for the **-power** argument.

Table F.11—Liberty attribute mapping for -power and -power_switchable

Name	Group	Type	Value
pg_type	pg_pin	string	backup_power

1) This mapping takes place when the cell is also specified with the **-power_switchable** option. In this case, tools shall connect the pin to the power net of the isolation supply set specified or implied by the corresponding isolation strategy.

Table F.12—Liberty attribute mapping for -power

Name	Group	Type	Value
pg_type	pg_pin	string	primary_power

2) This mapping takes place when the cell is not specified with the **-power_switchable** option. In this case, tools shall connect the pin to power net of the primary supply set of the power domain in which the cell is located.

c) Table F.13 and Table F.14 indicate the Liberty attribute mapping for the **-ground** argument.

Table F.13—Liberty attribute mapping for -ground and -ground_switchable

Name	Group	Type	Value
pg_type	pg_pin	string	backup_ground

1) This mapping takes place when the cell is also specified with the **-ground_switchable** option. In this case, tools shall connect the pin to the ground net of the isolation supply set specified or implied by the corresponding isolation strategy.

Table F.14—Liberty attribute mapping for -ground

Name	Group	Type	Value
pg_type	pg_pin	string	primary_ground

2) This mapping takes place when the cell is not specified with the **-ground_switchable** option. In this case, tools shall connect the pin to ground net of the primary supply set of the power domain in which the cell is located.

d) Table F.15 indicates the Liberty attribute mapping for the **-enable** argument.

Table F.15—Liberty attribute mapping for -enable

Name	Group	Type	Value
isolation_cell_enable_pin	pin	Boolean	true

1) Tools need to connect the enable pin to the isolation signal specified in the corresponding isolation strategy.

e) **-clamp_cell** has no corresponding Liberty attribute.

1) For a clamp high cell, tools can presume the following connections unless they are specified explicitly:

i) Connect the data pin to the net or pin targeted for isolation;

ii) Connect the enable pin to the isolation signal specified in the corresponding isolation strategy;

iii) Connect the power pin of the cell to the power net of the isolation supply set specified or implied by the corresponding isolation strategy.

2) For a clamp low cell, tools can presume the following connections unless they are specified explicitly:

i) Connect the data pin to the net or pin targeted for isolation;

ii) Connect the enable pin to the isolation signal specified in the corresponding isolation strategy;

iii) Connect the ground pin of the cell to the ground net of the isolation supply set specified or implied by the corresponding isolation strategy.

f) For **-pin_groups**, the corresponding modeling of a multi-bit isolation cell is the `bundle` group in Liberty. Within the bundle group, standard pin attributes can be used for the isolation data pin and enable pin.

g) **-no_enable** has no corresponding Liberty attribute.

h) Table F.16 indicates the Liberty attribute mapping for the **-always_on_pins** argument.

Table F.16—Liberty attribute mapping for -always_on_pins

Name	**Group**	**Type**	**Value**
always_on	pin	Boolean	true

i) **-aux_enables** has no corresponding Liberty attribute.

This option models isolation cells with more than one enable pins. The index `0` is reserved for the isolation enable pin specified by the **-enable** option. The pins listed in this option start with index `1`. To use such cells for isolation, the corresponding strategy needs to be specified with a signal list in the **-isolation_signal** option. The elements in the list are ordered with the index starting with `0`. The signals in the list shall be connected to the pins of the cells with the same index.

j) Table F.17 and Table F.18 indicates the Liberty attribute mapping for the **-power_switchable** and **-ground_swithcable** arguments, respectively.

Table F.17—Liberty attribute mapping for -power_switchable

Name	**Group**	**Type**	**Value**
pg_type	pg_pin	string	primary_power

1) Tools need to connect the pin to the power net of the primary supply set of the power domain in which the cell is located.

Table F.18—Liberty attribute mapping for -ground_switchable

Name	**Group**	**Type**	**Value**
pg_type	pg_pin	string	primary_ground

2) Tools need to connect the pin to the ground net of the primary supply set of the power domain in which the cell is located.

k) **-valid_location** has no corresponding Liberty attribute.

1) Verification tools need to ensure the implementation of the isolation strategy places the isolation cells in the correct location based on this definition.

l) **-non_dedicated** has no corresponding Liberty attribute.**define_level_shifter_cell**

define_level_shifter_cell [from 7.5]
-cells *cell_list*
[**-input_voltage_range** {*voltage_ranges*}] [**-output_voltage_range** {*voltage_ranges*}]
[**-ground_input_voltage_range** {*voltage_ranges*}]
[**-ground_output_voltage_range** {*voltage_ranges*}]
[**-direction** <**low_to_high** | **high_to_low** | **both**>]
[**-input_power_pin** *power_pin*]
[**-output_power_pin** *power_pin*]
[**-input_ground_pin** *ground_pin*]
[**-output_ground_pin** *ground_pin*]
[**-ground** *ground_pin*] [**-power** *power_pin*]
[**-enable** *pin* | **-pin_groups** {{*input_pin output_pin* [*enable_pin*]}*}]
[**-valid_location** <**source** | **sink** | **either** | **any**>]
[**-bypass_enable** *expression*] [**-multi_stage** *integer*]

The Liberty mappings for this command are as follows:

a) Table F.19 indicates the Liberty attribute mapping for all cells identified by the **-cells** option of this command.

Table F.19—Liberty attribute mapping for -cells

Name	**Group**	**Type**	**Value**
is_level_shifter	cell	Boolean	true

b) **-input_voltage_range** has no corresponding Liberty attribute.

The syntax of this attribute is different from the Liberty attribute `input_voltage_range`, which specifies only two values to indicate the voltage lower bound and upper bound.

c) **-output_voltage_range** has no corresponding Liberty attribute.

The syntax of this attribute is different from the Liberty attribute `output_voltage_range`, which specifies only two values to indicate the voltage lower bound and upper bound.

d) **-ground_input_voltage_range** has no corresponding Liberty attribute.

The syntax of this attribute is different from the Liberty attribute `input_voltage_range`, which specifies only two values to indicate the voltage lower bound and upper bound.

e) **-ground_output_voltage_range** has no corresponding Liberty attribute.

The syntax of this attribute is different from the Liberty attribute `output_voltage_range`, which specifies only two values to indicate the voltage lower bound and upper bound.

f) **-direction** has no corresponding Liberty attribute.

g) Table F.20 indicates the Liberty attribute mapping for the **-input_power_pin** argument.

Table F.20—Liberty attribute mapping for -input_power_pin

Name	Group	Type	Value
pg_type	pg_pin	string	primary_power

1) Tools need to connect the pin to the power net of the input supply set in the corresponding level-shifter strategy [identified by the **-input_supply** of **set_level_shifter** (see 6.45)] or the power net of the driving cell of the level-shifter, unless the connection is specified explicitly.

h) Table F.21 indicates the Liberty attribute mapping for the **-output_power_pin** argument.

Table F.21—Liberty attribute mapping for -output_power_pin

Name	Group	Type	Value
pg_type	pg_pin	string	primary_power

Tools need to connect the pin to the power net of the output supply set in the corresponding level-shifter strategy [identified by the **-output_supply** of **set_level_shifter** (see 6.45)] or the power net of the load cell of the level-shifter, unless the connection is specified explicitly.

i) Table F.22 indicates the Liberty attribute mapping for the **-input_ground_pin** argument.

Table F.22—Liberty attribute mapping for -input_ground_pin

Name	Group	Type	Value
pg_type	pg_pin	string	primary_ground

Tools need to connect the pin to the ground net of the input supply set in the corresponding level-shifter strategy [identified by the **-input_supply** of **set_level_shifter** (see 6.45)] or the ground net of the driving cell of the level-shifter, unless the connection is specified explicitly.

j) Table F.23 indicates the Liberty attribute mapping for the **-output_ground_pin** argument.

Table F.23—Liberty attribute mapping for -output_ground_pin

Name	Group	Type	Value
pg_type	pg_pin	string	primary_ground

Tools need to connect the pin to the ground net of the output supply set in the corresponding level-shifter strategy [identified by the **-output_supply** of **set_level_shifter** (see 6.45)] or the ground net of the load cell of the level-shifter, unless the connection is specified explicitly.

k) Table F.24 indicates the Liberty attribute mapping for the **-ground** argument.

Table F.24—Liberty attribute mapping for -ground

Name	Group	Type	Value
pg_type	pg_pin	string	primary_ground

1) Tools need to connect the pin to ground net of the primary supply set of the power domain in which the cell is located.

l) Table F.25 indicates the Liberty attribute mapping for the **-power** argument.

Table F.25—Liberty attribute mapping for -power

Name	**Group**	**Type**	**Value**
pg_type	pg_pin	string	primary_power

1) Tools need to connect the pin to power net of the primary supply set of the power domain in which the cell is located.

m) Table F.26 indicates the Liberty attribute mapping for the **-enable** argument.

Table F.26—Liberty attribute mapping for -enable

Name	**Group**	**Type**	**Value**
level_shifter_enable_pin	pin	Boolean	true

n) For **-pin_groups**, the corresponding modeling of a multi-bit isolation cell is the `bundle` group in Liberty. Within the bundle group, standard pin attributes can be used for the isolation data pin and enable pin.

o) **-valid_location** has no corresponding Liberty attribute.

Verification tools need to ensure the implementation of the level-shifter strategy places the level-shifter in the correct location based on this definition.

p) **-bypass_enable** has no corresponding Liberty attribute.

The polarity of the bypass enable pin can be derived from the Liberty attribute `level_shifter_data_pin` and the function of the output pin.

q) **-multi_stage** has no corresponding Liberty attribute.

F.6 define_power_switch_cell

define_power_switch_cell [from 7.6]
  -cells *cell_list*
  -type <**footer** | **header**>
  -stage_1_enable *expression* [**-stage_1_output** *expression*]
  {**-power_switchable** *power_pin* **-power** *power_pin*
  | **-ground_switchable** *ground_pin* **-ground** *ground_pin*]}
  [**-stage_2_enable** *expression* [**-stage_2_output** *expression*]]
  [**-always_on_pins** *ordered_pin_list*]
  [**-gate_bias_pin** *power_pin*]

The Liberty mappings for this command are as follows:

a) Table F.27 indicates the Liberty attribute mapping for all cells identified by the **-cells** option of this command.

Table F.27—Liberty attribute mapping for -cells

Name	Group	Type	Value
switch_cell_type	cell	Boolean	coarse_grain

b) For **-type**, if a cell has a `pg_pin` with `pg_type internal_power` in the Liberty definition, then the cell is a `header` cell; if a cell has a `pg_pin` with `pg_type internal_ground`, then the cell is a `footer` cell.

c) **-stage_1_enable** (**-stage_2_enable**) has no corresponding Liberty attribute(s).

1) The Liberty pin attribute does not differentiate the function between the two enables, so two user attributes are created here. However, the Liberty pin attribute `switch_function` can be used to describe the switch function on the switched `pg_pin`, which has `pg_type` of either `internal_power` or `internal_ground`.

2) Tools need to connect the pins to the switch-enable signal specified in the **-control_port** option of the corresponding **create_power_switch** command (see 6.21).

d) Table F.28 indicates the Liberty attribute mapping for the **-power_switchable** argument.

Table F.28—Liberty attribute mapping for -power_switchable

Name	Group	Type	Value
pg_type	pg_pin	string	internal_power

Tools need to connect the pin to the supply net specified by the **-output_supply_port** option of the corresponding **create_power_switch** (see 6.21) command.

e) Table F.29 indicates the Liberty attribute mapping for the **-power** argument.

Table F.29—Liberty attribute mapping for -power

Name	Group	Type	Value
pg_type	pg_pin	string	primary_power

Tools need to connect the pin to the supply net specified by the **-input_supply_port** option of the corresponding **create_power_switch** (see 6.21) command.

f) Table F.30 indicates the Liberty attribute mapping for the **-ground_switchable** argument.

Table F.30—Liberty attribute mapping for -ground_switchable

Name	Group	Type	Value
pg_type	pg_pin	string	internal_ground

1) Tools need to connect the pin to the supply net specified by the **-output_supply_port** option of the corresponding **create_power_switch** (see 6.21) command.

g) Table F.31 indicates the Liberty attribute mapping for the **-ground** argument.

Table F.31—Liberty attribute mapping for -ground

Name	Group	Type	Value
pg_type	pg_pin	string	primary_ground

1) Tools need to connect the pin to the supply net specified by the **-input_supply_port** option of the corresponding **create_power_switch** (see 6.21) command.

h) For **-stage_1_output (-stage_2_output)**, the corresponding output pin can be automatically identified, based on the `pin` function and the `stage_1_enable` and `stage_2_enable` attributes.

Tools need to connect the pins to the switch-enable signal specified in the **-ack_port** option of the corresponding **create_power_switch** command (see 6.21).

i) Table F.32 indicates the Liberty attribute mapping for the **-always_on_pins** argument.

Table F.32—Liberty attribute mapping for -always_on_pins

Name	Group	Type	Value
always_on	pin	Boolean	true

j) Table F.33 indicates the Liberty attribute mapping for the **-gate_bias_pin** argument.

Table F.33—Liberty attribute mapping for -gate_bias_pin

Name	Group	Type	Value
user_pg_type	pg_pin	string	gate_bias

F.7 define_retention_cell

define_retention_cell [from 7.7]
  -cells *cell_list*
  -power *power_pin*
  -ground *ground_pin*
  [**-cell_type** *string*]
  [**-always_on_pins** *pin_list*]
  [**-restore_function** {{**pin** <**high** | **low** | **posedge** | **negedge**}}]
  [**-save_function** {{**pin** <**high** | **low** | **posedge** | **negedge**}}]
  [**-restore_check** *expression*] [**-save_check** *expression*]
  [**-retention_check** *expression*] [**-hold_check** *pin_list*]
  [**-always_on_components** *component_list*]
  [**-power_switchable** *power_pin*] [**-ground_switchable** *ground_pin*]

The Liberty mappings for this command are as follows:

a) Table F.34 indicates the Liberty attribute mapping for all cells identified by the **-cells** option of this command.

Table F.34—Liberty attribute mapping for -cells

Name	Group	Type	Value
retention_cell	cell	string	cell_type

The `cell_type` is the same string specified in the option **-cell_type** (see Table F.39).

b) Table F.35 and Table F.36 indicate the Liberty attribute mapping for the **-power** argument.

Table F.35—Liberty attribute mapping for -power and -power_switchable

Name	Group	Type	Value
pg_type	pg_pin	string	backup_power

1) This mapping takes place when the cell is also specified with the **-power_switchable** option. In this case, tools shall connect the pin to the power net of the retention supply set specified or implied by the corresponding retention strategy.

Table F.36—Liberty attribute mapping for -power

Name	Group	Type	Value
pg_type	pg_pin	string	primary_power

This mapping takes place when the cell is not specified with the **-power_switchable** option. In this case, tools shall connect the pin to power net of the primary supply set of the power domain in which the cell is located.

c) Table F.37 and Table F.38 indicate the Liberty attribute mapping for the **-ground** argument.

Table F.37—Liberty attribute mapping for -ground and -ground_switchable

Name	Group	Type	Value
pg_type	pg_pin	string	backup_ground

1) This mapping takes place when the cell is also specified with the **-ground_switchable** option. In this case, tools shall connect the pin to the ground net of the retention supply set specified or implied by the corresponding retention strategy.

Table F.38—Liberty attribute mapping for -ground

Name	Group	Type	Value
pg_type	pg_pin	string	primary_ground

2) This mapping takes place when the cell is not specified with the **-ground_switchable** option. In this case, tools shall connect the pin to ground net of the primary supply set of the power domain in which the cell is located.

d) Table F.39 indicates the Liberty attribute mapping for the **-cell_type** argument.

Table F.39—Liberty attribute mapping for -cell_type

Name	Group	Type	Value
retention_cell	cell	string	user_string

e) Table F.40 indicates the Liberty attribute mapping for the **-always_on_pins** argument.

Table F.40—Liberty attribute mapping for -always_on_pins

Name	Group	Type	Value
always_on	pin	Boolean	true

c) Table F.41 indicates the Liberty attribute mapping for the **-restore_function** argument.

Table F.41—Liberty attribute mapping for -restore_function

Name	Group	Type	Value
retention_pin	pin	string	restore | save_restore

1) The pin shall be specified by the `retention_pin` attribute in Liberty. If the cell has only one retention pin, then the corresponding attribute value is `save_restore`; otherwise the corresponding value is `restore`.

2) Table F.42 indicates the Liberty attribute mapping for the retention control pin functionality.

Table F.42—Liberty attribute mapping for -retention_action

Name	Group	Type	Value
restore_action	pin	complex	<L | H | R | F>

i) The pin shall also be specified by the `retention_pin` attribute in Liberty.

ii) The mapping of the Liberty value to the UPF value is:

L: low

H: high

R: posedge

F: negedge

iii) Tools need to connect the pin to the signal specified in the **-restore_signal** option of the **set_retention** command (see 6.49). The polarity or edge-sensitivity specification of the two options shall be identical.

d) Table F.43 indicates the Liberty attribute mapping for the **-save_function** argument.

Table F.43—Liberty attribute mapping for -save_function

Name	Group	Type	Value
retention_pin	pin	string	save | save_restore

1) The pin shall be specified by the `retention_pin` attribute in Liberty. If the cell has only one retention pin, then the corresponding attribute value is `save_restore`; otherwise the corresponding value is `save`.

2) Table F.44 indicates the Liberty attribute mapping for the retention control pin functionality.

Table F.44—Liberty attribute mapping for -retention_action

Name	Group	Type	Value
save_action	pin	complex	<L | H | R | F>

i) The pin shall also be specified by the `retention_pin` attribute in Liberty.

ii) The mapping of the Liberty value to the UPF value is:

L: low

H: high

R: posedge

F: negedge

iii) Tools need to connect the pin to the signal specified in the **-save_signal** option of the **set_retention** command (see 6.49). The polarity or edge-sensitivity specification of the two options shall be identical.

e) **-restore_check** has no corresponding Liberty attribute.

f) **-save_check** has no corresponding Liberty attribute.

g) **-retention_check** has no corresponding Liberty attribute.

h) **-hold_check** has no corresponding Liberty attribute.

i) **-always_on_components** has no corresponding Liberty attribute.

j) Table F.45 indicates the Liberty attribute mapping for the **-power_switchable** argument.

Table F.45—Liberty attribute mapping for -power_switchable

Name	Group	Type	Value
pg_type	pg_pin	string	primary_power

k) Tools need to connect the pin to power net of the primary supply set of the power domain in which the cell is located.

l) Table F.46 indicates the Liberty attribute mapping for the **-ground_switchable** argument.

Table F.46—Liberty attribute mapping for -ground_switchable

Name	**Group**	**Type**	**Value**
pg_type	pg_pin	string	primary_ground

m) Tools need to connect the pin to the ground net of the primary supply set of the power domain in which the cell is located.

Annex G

(informative)

Power-management cell modeling examples

This annex shows examples for how to model various types of power-management cell in power intent commands in Clause 6 and corresponding Liberty syntax. The information provides mapping between each command to the Liberty syntax. Unless otherwise stated, the referenced Liberty attributes are based on the Liberty 2014.09 release (see Liberty library format usage [B4]).

G.1 Modeling always-on cells

G.1.1 Types of always-on cells

An *always-on cell* is simply a library cell with more than one set of power and ground pins that can remain functional even when the supply to the rail-connected power or ground pin is switched off, as long as the non-switchable power or ground remains on. An always-on cell shall have at least a non-switchable power or a non-switchable ground pin defined.

Although a cell is called always-on, it does not mean the cell can never be powered off. When the supply to the non-switchable power or ground of such cell is switched off, the cell becomes non-functional. In other words, the term *always-on* actually means relatively always-on.

Any logic function can be implemented in the form of an always-on cell, such as an always-on buffer, always-on inverter, always-on `AND` gate, or even always-on flop. In the following subclauses, several different types of always-on cells are used as examples to describe how to use the **define_always_on_cell** command (see 7.2):

- Modeling a power-switched always-on buffer
- Modeling a ground-switched always-on buffer
- Modeling a power- and ground-switched always-on buffer
- Modeling a power-switched always-on flop with internal isolation

G.1.2 Modeling a power-switched always-on buffer

To model a power-switched always-on buffer, use the **define_always_on_cell** command (see 7.2) with the following options:

> **define_always_on_cell**
>   **-cells** *cells*
>   **-power** *pin* **-power_switchable** *pin* **-ground** *pin*

In Figure G.1, a type of power-switched always-on buffer is shown. The cell's rail connection `VSW` is not used by the cell. The actual power of the cell comes from `VDD`, which needs to be routed separately. The following command models this type of cell:

```
define_always_on_cell
    -cells LP_Buf_Pow
    -power VDD -power_switchable VSW -ground VSS
```

The same command can also be used to describe any other type of power-switched always-on cells, such as an inverter, AND gate, etc.

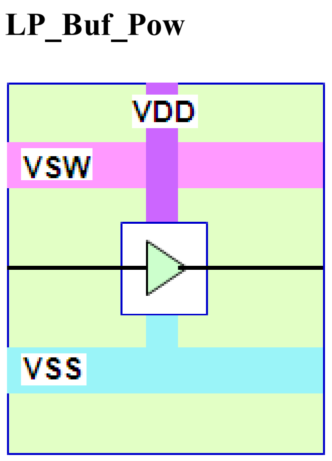


Figure G.1—Power-switched always-on buffer

Liberty model

```
library(mylib) {

  voltage_map(VDD, 1.0);  /* backup power */
  voltage_map(VSW, 1.0);  /* primary power */
  voltage_map(VSS, 0.0);  /* primary ground */

  cell(LP_Buf_Pow) {
    always_on : true;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : backup_power;
    }
    pg_pin(VSW) {
      voltage_name : VSW;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
    }
    pin(Y) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      function : "A";
      power_down_function : "!VDD + VSS";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.1.3 Modeling a ground-switched always-on buffer

To model a ground-switched always-on buffer, use the **define_always_on_cell** command (see 7.2) with the following options:

> **define_always_on_cell**
>   **-cells** *cells*
>   **-power** *pin* **-ground_switchable** *pin* **-ground** *pin*

In Figure G.2, a type of ground-switched always-on buffer is shown. The cell's rail connection `GSW` is not used by the cell. The actual ground of the cell comes from `VSS`, which needs to be routed separately. The following command models this type of cell:

```
define_always_on_cell
    -cells LP_Buf_Gnd
    -ground VSS -power VDD -ground_switchable GSW
```

The same command can also be used to describe any other type of ground-switched always-on cells, such as an inverter, `AND` gate, etc.

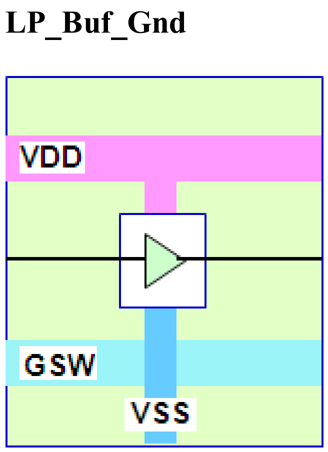


Figure G.2—Ground-switched always-on buffer

Liberty model

```
library(mylib) {

  voltage_map(VDD, 1.0);  /* primary power */
  voltage_map(GSW, 0.0);  /* primary ground */
  voltage_map(VSS, 0.0);  /* backup ground */

  cell(LP_Buf_Gnd) {
    always_on : true;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(GSW) {
      voltage_name : GSW;
      pg_type : primary_ground;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : backup_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
    }
    pin(Y) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      function : "A";
      power_down_function : "!VDD + VSS";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.1.4 Modeling a power- and ground-switched always-on buffer

To model a power- and ground-switched always-on buffer, use the **define_always_on_cell** command (see 7.2) with the following options:

> **define_always_on_cell**
> **-cells** *cells*
> **-power_switchable** *pin* **-ground_switchable** *pin*
> **-power** *pin* **-ground** *pin*

In Figure G.3, a type of power- and ground-switched always-on buffer is shown. The cell has both power and ground rail connections, VSW and GSW, respectively, but they are not used by the cell. The actual power and ground pins the cell come from VDD and VSS, which need to be routed separately. The following command models this type of cell:

```
define_always_on_cell
   -cells LP_Buf_Pow_Gnd
   -power VDD -ground VSS
   -power_switchable VSW -ground_switchable GSW
```

The same command can also be used to describe any other type of power- and ground-switched always-on cells such as an inverter, AND gate, etc.

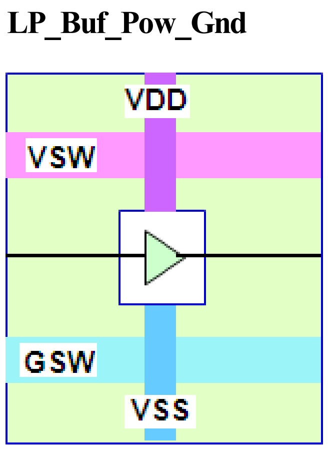


Figure G.3—Power- and ground-switched always-on buffer

Liberty model

```
library(mylib) {

  voltage_map(VDD, 1.0);  /* Backup power */
  voltage_map(VSW, 1.0);  /* Primary power */
  voltage_map(GSW, 0.0);  /* Primary ground */
  voltage_map(VSS, 0.0);  /* Backup ground */

  cell(LP_Buf_Pow_Gnd) {
    always_on : true;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : backup_power;
    }
    pg_pin(VSW) {
      voltage_name : VSW;
      pg_type : primary_power;
    }
    pg_pin(GSW) {
      voltage_name : GSW;
      pg_type : primary_ground;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : backup_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
    }
    pin(Y) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      function : "A";
      power_down_function : "!VDD + VSS";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.1.5 Modeling a power-switched always-on flop with internal isolation

To model a power-switched always-on cell with internal isolation at some input pins, use the **define_always_on_cell** command (see 7.2) with the following options:

define_always_on_cell
  -cells *cells*
  -power *pin* **-power_switchable** *pin* **-ground** *pin*
  -isolated_pins *list_of_pin_lists* [**-enable** *expression_list*]

The always-on flip-flop cell in Figure G.4 has internal isolation at input pins `SE` and `SI` with the other input pin `ISO` as the control. The following command models this type of cell:

```
define_always_on_cell
   -cells LP_ff
   -power VDD -power_switchable VSW -ground VSS \
   -isolated_pins { {SE SI} } -enable {!Iso}
```

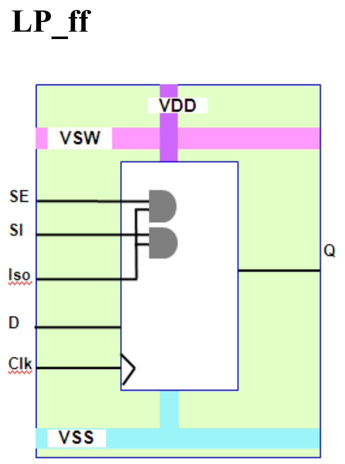


Figure G.4—Power-switched always-on flop with input isolation on pins SE and SI

Liberty model

```
library(mylib) {

  voltage_map(VDD, 1.0);  /* Backup Power */
  voltage_map(VSW, 1.0);  /* Primary Power */
  voltage_map(VSS, 0.0);  /* Primary Ground */

  cell(LP_ff) {
    always_on : true;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : backup_power;
    }
    pg_pin(VSW) {
      voltage_name : VSW;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    ff (IQ,IQN) {
      clocked_on : "Clk";
      next_state : "(D + (!SE * Iso) + (SI * Iso))"; /* assumed function in the
absence of full cell schematic */
    }
    pin(D) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
    }
    pin(SE) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      is_isolated : true;
      isolation_enable_condition : "!Iso";
    }
    pin(SI) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      is_isolated : true;
      isolation_enable_condition : "!Iso";
    }
    pin(Iso) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
    }
    pin(Clk) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
    }
    pin(Q) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      function : "IQ";
      power_down_function : "!VDD + VSS";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.2 Modeling cells with internal diodes

Cells with input pins connected to diodes need to be properly modeled to avoid electrical failure in a design with power-management. To model such cells, use the **define_diode_clamp** command (see 7.3) with the following options:

define_diode_clamp
    -cells *cell_list*
    -data_pins *pin_list*
    [**-type** <**power** | **ground** | **both**>]
    [**-power** *pin*] [**-ground** *pin*]

To describe the different type of diode connected pins shown in Figure G.5, use the following commands:

```
define_diode_clamp -cells cellA -data_pins in1 -type power -power VDD1
define_diode_clamp -cells cellB -data_pins in1 -type ground -ground VSS2
define_diode_clamp -cells cellC -data_pins in1 -type both \
   -power VDD1 -ground VSS2
define_diode_clamp -cells cellD -data_pins in1 -type power -power VDD
```

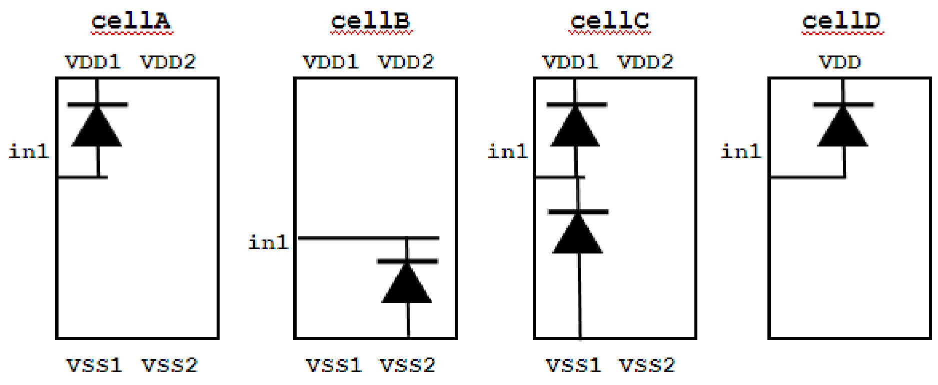


Figure G.5—Cells with different type of internal diodes

Liberty model

```
library (mylib) {

  voltage_map (VDD,  1.0);
  voltage_map (VDD1, 1.0);
  voltage_map (VSS2, 0.0);

/* An example of a power_diode cell */
  cell (cellA) {
    antenna_diode_type : power;
    pg_pin (VDD1) {
      voltage_name : VDD1;
      pg_type : primary_power;
    } /* end pg_pin group */
    pin (in1) {
      antenna_diode_related_power_pins : VDD1;
      direction : input;
    } /* end pin group */
  }/* end cell group */

/* An example of a ground_diode cell */
  cell (cellB) {
    antenna_diode_type : ground;
    pg_pin (VSS2) {
      voltage_name : VSS2;
      pg_type : primary_ground;
    }
    pin (in1) {
      antenna_diode_related_ground_pins : VSS2;
      direction : input;
    }
  }/* end cell group */

/* An example of a power_ground diode cell */
  cell (cellC) {
    antenna_diode_type : power_and_ground;
    pg_pin (VDD1) {
      voltage_name : VDD1;
      pg_type : primary_power;
    }
    pg_pin (VSS2) {
      voltage_name : VSS2;
      pg_type : primary_ground;
    }
    pin (in1) {
      antenna_diode_related_power_pins : VDD1;
      antenna_diode_related_ground_pins : VSS2;
      direction : input;
    } /* end pin group */
  } /* end cell group */

/* An example of a power_diode cell */
  cell (cellD) {
    antenna_diode_type : power;
    pg_pin (VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pin (in1) {
      antenna_diode_related_power_pins : VDD;
     direction : input;
    } /* end pin group */
  } /* end cell group */
} /* end library group */
```

G.3 Modeling isolation cells

G.3.1 Types of isolation cells

Isolation logic is required when the leaf-drivers and leaf-loads of a net are in power domains that are not on and off at the same time, or because it is part of the design intent. The following is a list of the most typical isolation cells:

- Isolation cell to be placed in the unswitched domain
- Isolation cell to be used in a ground-switchable domain
- Isolation cell to be used in a power-switchable domain
- Isolation cells to be used in a power- or ground-switchable domain
- Isolation cells without follow pins that can be placed in any domain
- Isolation cells without always-on power pins that can be placed in a switchable power domain
- Isolation cells without an enable pin
- Isolation clamp cell
- Isolation level-shifter combo cell

All types of isolation cells are defined using the **define_isolation_cell** command (see 7.4). The following subclauses indicate which command options to use for each type.

G.3.2 Modeling an isolation cell to be placed in the unswitched domain

To model an isolation cell to be placed in an unswitched domain, use the **define_isolation_cell** command (see 7.4) with the following options:

define_isolation_cell
  -cells *cell_list*

-power *power_pin* **-ground** *ground_pin*
-valid_location on
{**-enable** *pin* | **-no_enable** <**high** | **low** | **hold**>}

Figure G.6 shows an `AND` cell that can be used for isolation purposes.

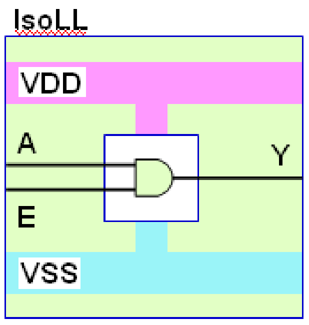


Figure G.6—Dedicated isolation cell in unswitched domain

Liberty model

```
library(mylib) {

  voltage_map(VDD, 1.0);   /* primary power */
  voltage_map(VSS, 0.0);   /* primary ground */

  cell(IsoLL) {
    is_isolation_cell : true;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      isolation_cell_data_pin : true;
    } /* end pin group */
    pin(E) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      isolation_cell_enable_pin : true;
    } /* end pin group */
    pin(Y) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      function : "A * E";
      power_down_function : "!VDD + VSS";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

The following command models the isolation cell in Figure G.6:

```
define_isolation_cell \
   -cells IsoLL \
   -power VDD -ground VSS \
   -enable E \
   -valid_location on
```

NOTE—To use the cell in regular logic, add the **-non_dedicated** option. Non-dedicated cells are typically only placed in the unswitched domain (i.e., `-valid_location on`).

G.3.3 Modeling an isolation cell for ground-switchable domain

To model an isolation cell to be used in a ground-switchable domain, use the **define_isolation_cell** command (see 7.4) with the following options:

define_isolation_cell
  -cells *cell_list*
  {**-enable** *pin* | **-no_enable** <**high** | **low** | **hold**>}
  -ground_switchable *ground_pin*
  -power *power_pin* **-ground** *ground_pin*
  [**-valid_location** <**source** | **sink** | **on** | **off**>]
  [**-always_on_pins** *pin_list*]

Figure G.7 shows an `AND` cell that has the path from power to ground cut off on the ground side. This `AND` cell can only be used for isolation.

The following command models the isolation cell in Figure G.7, which can be placed at the output of a ground-switchable domain:

```
define_isolation_cell \
   -cells IsoLL \
   -ground_switchable GSW \
   -power VDD -ground VSS \
   -enable E \
   -valid_location source
```

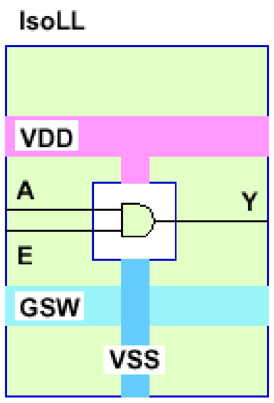


Figure G.7—Isolation cell with ground-switchable pin

Liberty model

```
library(mylib) {

  voltage_map(VDD, 1.0);    /* primary power */
  voltage_map(GSW, 0.0);    /* primary ground */
  voltage_map(VSS, 0.0);    /* backup ground */

  cell(IsoLL) {
    is_isolation_cell : true;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(GSW) {
      voltage_name : GSW;
      pg_type : primary_ground;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : backup_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : GSW;
      isolation_cell_data_pin : true;
    }
    pin(E) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      isolation_cell_enable_pin : true;
    }
    pin(Y) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      function : "A * E";
      clamp_0_function : "!E";
      power_down_function : "!VDD + VSS";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.3.4 Modeling an isolation cell for power-switchable domain

To model an isolation cell to be used in a power-switchable domain, use the **define_isolation_cell** command (see 7.4) with the following options:

define_isolation_cell
  -cells *cell_list*
  {**-enable** *pin* | **-no_enable** <**high** | **low** | **hold**>}
  -power_switchable *power_pin*
  -power *power_pin* **-ground** *ground_pin*
  [**-valid_location** <**source** | **sink** | **on** | **off**>]

Figure G.8 shows an AND cell that has the path from power to ground cut off on the power side. This AND cell can only be used for isolation.

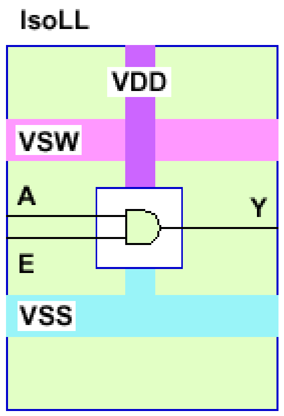


Figure G.8—Isolation cell with power-switchable pin

The following command models the isolation cell in Figure G.8:

```
define_isolation_cell \
    -cells IsoLL \
    -power_switchable VSW \
    -power VDD -ground VSS \
    -enable E \
    -valid_location source
```

Such a cell would be a good candidate for an isolation strategy like the following, assuming `PSW` is a switchable domain.

```
set_isolation myIso -domain PSW -applies_to outputs \
    -isolation_signal iso -isolation_sense high \
    -clamp_value low -location self
```

```
Liberty model:
library(mylib) {

  voltage_map(VDD, 1.0);  /* backup power */
  voltage_map(VSW, 1.0);  /* primary power */
  voltage_map(VSS, 0.0);  /* primary ground */

  cell(IsoLL) {
    is_isolation_cell : true;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : backup_power;
    }
    pg_pin(VSW) {
      voltage_name : VSW;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : VSW;
      related_ground_pin : VSS;
      isolation_cell_data_pin : true;
    }
    pin(E) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      isolation_cell_enable_pin : true;
    }
    pin(Z) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      function : "A * E";
      clamp_0_function : "!E";
      power_down_function : "!VDD + VSS";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.3.5 Modeling an isolation cell for power- and ground-switchable domains

To model an isolation cell to be used in a power- and ground-switchable domain, use the **define_isolation_cell** command (see 7.4) with the following options:

> **define_isolation_cell**
> **-cells** *cell_list*
> {**-enable** *pin* | **-no_enable** <**high** | **low** | **hold**>}
> **-power_switchable** *power_pin* **-ground_switchable** *ground_pin*
> **-power** *power_pin* **-ground** *ground_pin*
> [**-valid_location** <**source** | **sink** | **on** | **off**>]
> [**-always_on_pins** *pin_list*]

Figure G.9 shows an AND cell that has the path from power to ground cut off on the power and ground sides. This AND cell can only be used for isolation.

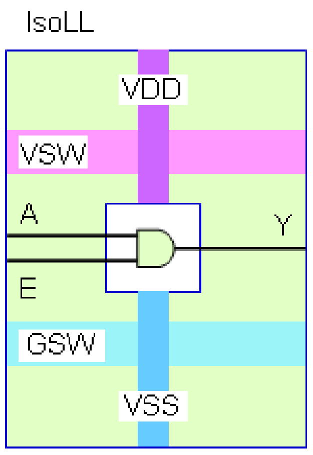


Figure G.9—Dedicated power- and ground-switchable isolation cell

The following command models the isolation cell in Figure G.9:

```
define_isolation_cell \
    -cells IsoLL \
    -power_switchable VSW -ground_switchable GSW \
    -power VDD -ground VSS \
      -enable E \
      -valid_location source

Liberty Model:
library(mylib) {

  voltage_map(VDD, 1.0);  /* backup power */
  voltage_map(VSW, 1.0);  /* primary power */
  voltage_map(VSS, 0.0);  /* backup ground */
  voltage_map(GSW, 0.0);  /* primary ground */

  cell(IsoLL) {
    is_isolation_cell : true;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : backup_power;
    }
    pg_pin(VSW) {
      voltage_name : VSW;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : backup_ground;
    }
    pg_pin(GSW) {
      voltage_name : GSW;
      pg_type : primary_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : VSW;
      related_ground_pin : GSW;
      isolation_cell_data_pin : true;
    }
    pin(E) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      isolation_cell_enable_pin : true;
    }
    pin(Z) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      function : "A * E";
      clamp_0_function : "!E";
      power_down_function : "!VDD + VSS";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.3.6 Modeling an isolation cell that can be placed in any domain

To model an isolation cell to be used in any domain, which typically does not have the power or ground rail connection, use the **define_isolation_cell** command (see 7.4) with the following options:

define_isolation_cell
  -cells *cell_list*
  {**-enable** *pin* | **-no_enable** <**high** | **low** | **hold**>}
  -power *power_pin* **-ground** *ground_pin*

-valid_location any
[**-always_on_pins** *pin_list*]

Liberty model

```
library(mylib) {

  voltage_map(VDD, 1.0);  /* primary power */
  voltage_map(VSS, 0.0);  /* primary ground */

  cell(isolation_cell_in_any_domain) {
    is_isolation_cell : true;

    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      isolation_cell_data_pin : true;
    }
    pin(E) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      isolation_cell_enable_pin : true;
    }
    pin(Y) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      function : "A * E";
      clamp_0_function : "!E";
      power_down_function : "!VDD + VSS";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.3.7 Modeling an isolation cell without always-on power pins that can be placed in a switchable power domain

In some cases, a regular single rail can also be placed at the output of a switchable domain and used for isolation. For example, for a 2-input `NOR` type cell, the output will be pull-down to the ground or logic zero as long as one of the inputs is logic one irrespective of the voltages at the power pins. As a result, such a cell can be placed within a power-gated domain to isolate the domain outputs to logic zero. To model such a cell, use the following command and options:

define_isolation_cell
-cells *cell_list*
-enable *pin*
-power_switchable *power_pin* **-ground** *ground_pin*
-valid_location off

Similarly, for a 2-input `NAND` type cell, the output will be driven to logic one as long as one of the inputs is logic zero, irrespective of the connection at the ground pins. As a result, such a cell can be placed within a ground-gated domain to isolate the domain outputs to logic one. To model such a cell, use the following command and options:

define_isolation_cell
  -cells *cell_list*
  -enable *pin*
  -power *power_pin* **-ground_switchable** *ground_pin*
  -valid_location off

Example

```
       define_isolation_cell \
       -cells NOR_ISO \
       -power_switchable VDD -ground VSS \
       -enable iso \
       -valid_location off

   Liberty Model:
library(mylib) {

  voltage_map(VDD, 1.0);  /* primary power */
  voltage_map(VSS, 0.0);  /* primary ground */

  cell(IsoLL) {
    is_isolation_cell : true;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
      permit_power_down : true;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      isolation_cell_data_pin : true;
    }
    pin(iso) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      isolation_cell_enable_pin : true;
      alive_during_partial_power_down : true;
    }
    pin(Y) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      alive_during_partial_power_down : true;
      function : "!(A + iso)";
      power_down_function : "VSS";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.3.8 Modeling an isolation cell without enable pin

There are special isolation cells that do not have an enable pin, but still can clamp output to a logic value when the primary power supply is switched off. Such a cell looks like a buffer, but its functionality is different when the switchable power is on and off. These cells are useful to buffer a net that typically requires always-on buffers, e.g., the retention control pin of all retention flops. The advantage of using such a cell versus an always-on buffer is it consumes much less power. To model such a cell, use the **define_isolation_cell** command (see 7.4) with the following options:

> **define_isolation_cell**
>   **-cells** *cell_list*
>   **-no_enable** <**high** | **low** | **hold**>
>   [**-power_switchable** *power_pin*] [**-ground_switchable** *ground_pin*]
>   [**-power** *power_pin*] [**-ground** *ground_pin*]
>   [**-valid_location** <**source** | **sink** | **on** | **off**>]
>   [**-always_on_pins** *pin_list*]

Example

```
    define_isolation_cell \
        -cells IsoLL \
        -power VDD -ground VSS \
        -no_enable low\
        -valid_location sink

Liberty Model:
library(mylib) {

  voltage_map(VDD, 1.0);  /* primary power */
  voltage_map(VSS, 0.0);  /* primary ground */

  cell(IsoLL) {
    is_isolation_cell : true;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
      permit_power_down : true;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      alive_during_partial_power_down : true;
      isolation_cell_data_pin : true;
    }
    pin(Y) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      alive_during_partial_power_down : true;
      function : "A";
      power_down_function : "VSS";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.3.9 Modeling an isolation clamp cell

An *isolation clamp high cell* is a simple PMOS transistor with the gate input being used as the enable pin. When its driver is switched off by a ground switch and the enable pin has value `0`, the connected net can be clamped to a logic high value as shown in Figure G.10.

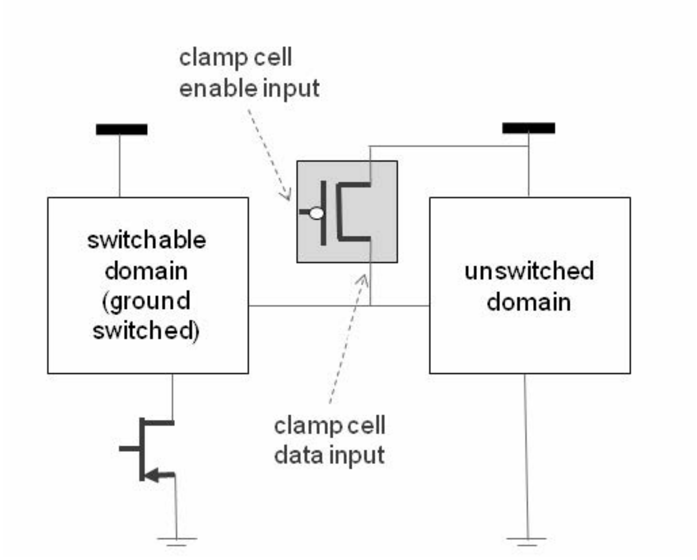


Figure G.10—Isolation clamp high cell

To model an isolation clamp high cell, use the **define_isolation_cell** command (see 7.4) with the following options:

define_isolation_cell
  -cells *cell_list*
  -enable *pin* **-clamp_cell high -power** *power_pin*
  -valid_location on

Liberty model

```
library(mylib) {

  voltage_map( VDD, 1.0);  /* primary power */
  voltage_map( VSS, 0.0);  /* primary ground */

  cell(clamp_high_isolation_cell) {
    is_isolation_cell : true;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      isolation_cell_data_pin : true;
    } /* end pin group */
    pin(iso_en) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      isolation_cell_enable_pin : true;
    }
    pin(Y) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      function : "1";
      clamp_1_function : "!iso_en";
      power_down_function : "!VDD + VSS";
    }/* end pin group*/
  } /*end cell group*/
} /* end library group */
```

An *isolation clamp low cell* is a simple NMOS transistor with the gate input being used as the enable pin. When its driver is switched off by a power switch and the enable pin has value 1, the connected net can be clamped to a logic low value as shown in Figure G.11.

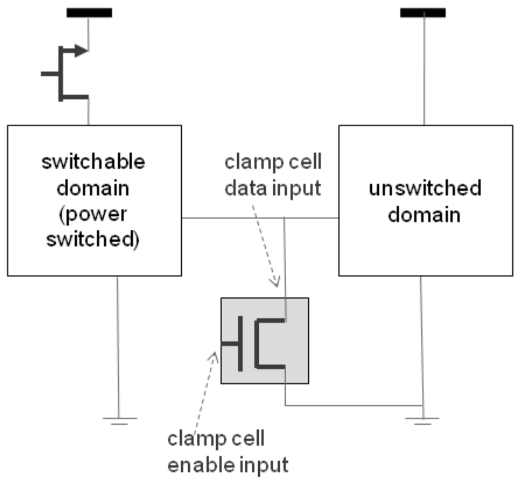


Figure G.11—Isolation clamp low cell

To model an isolation clamp low cell, use the **define_isolation_cell** command (see 7.4) with the following options:

define_isolation_cell
-cells *cell_list*
-enable *pin* **-clamp_cell low -ground** *ground_pin*
-valid_location on

Due to its special connectivity requirement, to apply such a power or ground clamp cell for a specific isolation strategy, use the **-port_map** option of the **use_interface_cell** command (see 6.55). In terms of power and ground net connection, if it is a clamp low cell, only the isolation ground net specified in **-isolation_supply** is used; if it is a clamp high cell, only the isolation power net specified in **-isolation_supply** is used.

Liberty model

```
library(mylib) {
  voltage_map( VDD, 1.0);  /* primary power */
  voltage_map( VSS, 0.0);  /* primary ground */

cell(clamp_low_isolation_cell) {
  is_isolation_cell : true;
  pg_pin(VDD) {
    voltage_name : VDD;
    pg_type : primary_power;
  }
  pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
  }
  pin(A) {
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    isolation_cell_data_pin : true;
  }

  pin(iso_en) {
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    isolation_cell_enable_pin : true;
  }

  pin(Y) {
    direction : output;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    function : "0";
    clamp_0_function : "iso_en";
    power_down_function : "!VDD + VSS";
   } /* end pin group*/
  } /*end cell group*/
} /* end library */
```

G.3.10 Modeling an isolation cell with multiple enable pins

Some isolation cells have an enable pin that is related to the non-switchable supply of the cell and additional enable pins that are related to the switchable supply. The switchable enable pin can be used to synchronize the isolation logic right before the non-switchable enable pin is activated or deactivated. To model an isolation cell with multiple enable pins, use the **define_isolation_cell** command (see 7.4) with the following options:

> **define_isolation_cell**
>   **-cells** *cell_list*
>   **-aux_enables** *pin_list* **-enable** *pin* [**-clamp** <**high** | **low**>]
>   [**-power_switchable** *power_pin*] [**-ground_switchable** *ground_pin*]
>   [**-power** *power_pin*] [**-ground** *ground_pin*]
>   [**-valid_location** <**source** | **sink** | **on** | **off** | **any**>]

To specify an isolation strategy that targets these types of isolation cells, use the **set_isolation** command with the **-isolation_signal** option (see 6.44) by assigning a list of signals to the option. In this list, the first signal is the one to drive the enable pin and the rest of the signals drive the auxiliary enable pin specified in the **-aux_enables** option in the same order.

Figure G.12 shows two examples of cells with multiple enable pins. The `iso` enable pin is related to the non-switchable supply `vddc`, while the `en` enable pin is related to the switchable supply `vdd`.

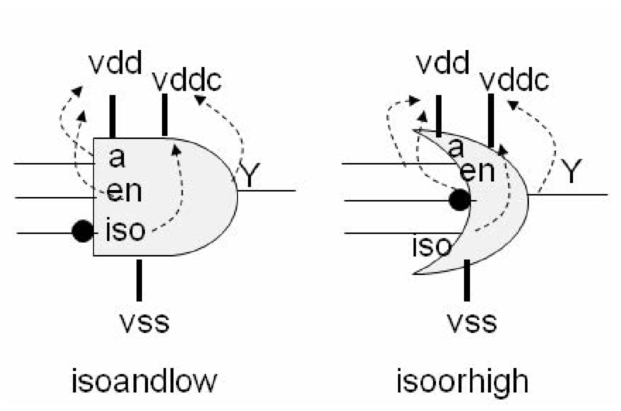


Figure G.12—Isolation cells with multiple enable pins

The following command models the `isoandlow` and `isoorhigh` cells in Figure G.12:

```
define_isolation_cell \
    -cells {isoandlow isoorhigh} \
    -aux_enables en \
    -power_switchable vdd \
    -power vddc -ground vss \
    -enable iso
```

The following commands show the isolation strategies that target the `isoandlow` and `isoorhigh` cells in Figure G.12:

```
set_isolation iso1 -domain PD1 -source PD1 \
    -isolation_signal { iso_drvr en_drvr} \
    -isolation_sense { high low } \
    -clamp_value 0

set_isolation iso2 -domain PD2 -source PD2 \
    -isolation_signal { iso_drvr en_drvr} \
    -isolation_sense { high high } \
    -clamp_value 1
```

```
Liberty model:
library(mylib) {

  voltage_map(vdd,  1.0);  /* primary power  */
  voltage_map(vddc, 1.0);  /* backup power   */
  voltage_map(vss,  0.0);  /* primary ground */

  cell(isoandlo) {
    is_isolation_cell : true;
    pg_pin(vdd) {
      voltage_name : vdd;
      pg_type : primary_power;
    }
    pg_pin(vddc) {
      voltage_name : vddc;
      pg_type : backup_power;
    }
    pg_pin(vss) {
      voltage_name : vss;
      pg_type : primary_ground;
    }
    pin(a) {
      direction : input;
      related_power_pin : vdd;
      related_ground_pin : vss;
      isolation_cell_data_pin : true;
    } /* end pin group */
    pin(iso) {
      direction : input;
      related_power_pin : vddc;
      related_ground_pin : vss;
      isolation_cell_enable_pin : true;
    } /* end pin group */
    pin(en) {
      direction : input;
      related_power_pin : vdd;
      related_ground_pin : vss;
      isolation_cell_enable_pin : true;
    } /* end pin group */
    pin(Y) {
      direction : output;
      related_power_pin : vddc;
      related_ground_pin : vss;
      function : " (!iso * en * a)";
      clamp_0_function : " (iso + !en) ";
      power_down_function : "!vdd + vss";
     } /* end pin group*/
   } /*end cell group*/

  cell(isoorhi) {
    is_isolation_cell : true;
    pg_pin(vdd) {
      voltage_name : vdd;
      pg_type : primary_power;
    }
    pg_pin(vddc) {
      voltage_name : vddc;
      pg_type : backup_power;
    }
    pg_pin(vss) {
      voltage_name : vss;
      pg_type : primary_ground;
    }
    pin(a) {
      direction : input;
      related_power_pin : vdd;
      related_ground_pin : vss;
      isolation_cell_data_pin : true;
    } /* end pin group */
    pin(iso) {
      direction : input;
      related_power_pin : vddc;
      related_ground_pin : vss;
      isolation_cell_enable_pin : true;
    } /* end pin group */
    pin(en) {
      direction : input;
      related_power_pin : vdd;
      related_ground_pin : vss;
      isolation_cell_enable_pin : true;
    } /* end pin group */
    pin(Y) {
      direction : output;
      related_power_pin : vddc;
      related_ground_pin : vss;
      function : " (iso + !en + a)";
      clamp_1_function : "(!iso + en)";
      power_down_function : "!vdd + vss";
    } /* end pin group*/
  } /*end cell group*/
} /* end library */
```

G.3.11 Modeling a multi-bit isolation cell

A *multi-bit isolation cell* has multiple pairs of input and output pins with each pair serving as a single-bit isolation cell. An example is shown in Figure G.13.

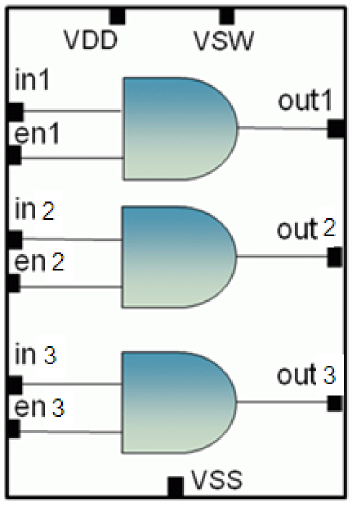


Figure G.13—Multi-bit isolation cell

If the cell uses the same enable pin for all pairs of input and output pins, there is no difference in modeling such a multi-bit cell with respect to the single-bit isolation cell. If the cell has different enable pins for the input and output pairs, model the cell using the **define_isolation_cell** command with the **-pin_groups** option (see 7.4).

The following command can be used to describe the multi-bit isolation cell for the power-switchable domain shown in Figure G.13 (see Figure G.8 for the corresponding single-bit cell):

```
define_isolation_cell -cells IsoLL \
   -power_switchable VSW \
   -power VDD -ground VSS \
   -pin_groups {{in1 out1 en1} {in2 out2 en2} {in3 out3 en3}}
```

```
Liberty Model:
library (mylib) {

  voltage_map(VDD, 1.0);  /* backup power */
  voltage_map(VSW, 1.0);  /* primary power */
  voltage_map(VSS, 0.0);  /* primary ground */
```

```
cell ("IsoLL") {
  is_isolation_cell : true;
  pg_pin (VDD) {
   voltage_name : VDD;
    pg_type : backup_power;
  }
  pg_pin (VSW) {
   voltage_name : VSW;
   pg_type : primary_power;
  }
  pg_pin (VSS) {
   voltage_name : VSS;
   pg_type : primary_ground;
  }
  bundle (in) {
    members (in1, in2, in3);
    direction : input;
    related_power_pin : VSW;
    related_ground_pin : VSS;
    isolation_cell_data_pin : true;
    pin (in1) {
      direction : input;
    }
    pin (in2) {
      direction : input;
    }
    pin (in3) {
      direction : input;
    } /* end pin group */
  } /* end bundle group */
  bundle (en) {
    members (en1, en2, en3);
    isolation_cell_enable_pin : true;
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    pin (en1) {
      direction : input;
    }
    pin (en2) {
      direction : input;
      capacitance : 1.0;
    }
    pin (en3) {
      direction : input;
    } /* end pin group */
  } /* end bundle group */
  bundle (out) {
    members ( out1, out2, out3);
    direction : output;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    function : "in * en";
    power_down_function : "!VDD + VSS";
    pin (out1) {
      …
    }
    pin (out2) {
      …
    }
    pin (out3) {
      …
    } /* end pin group */
    } /* end bundle group */
  } /* end cell group */
} /* end library group */
```

G.4 Modeling level-shifters

G.4.1 Types of level-shifters

To pass signals between portions of the design that operate on different power or ground voltages, level-shifters are needed. The following is a list of the most typical level-shifters:

- Power level-shifters
- Ground level-shifters
- Enabled level-shifters
- Bypass level-shifters
- Multi-stage level-shifters
- Multi-bit level-shifters

All types of level-shifters are defined using the **define_level_shifter_cell** command (see 7.5). The following subclauses indicate which command options to use for each type.

G.4.2 Modeling a power level-shifter

A power level-shifter passes signals between portions of the design that operate on different power voltages, but using the same ground voltages. To model a power level-shifter, use the following options from the **define_level_shifter_cell** command (see 7.5):

define_level_shifter_cell
  -cells *cell_list*
  -input_voltage_range {{*lower_bound upper_bound*}*}
  -output_voltage_range {{*lower_bound upper_bound*}*}
  [**-direction** <**low_to_high** | **high_to_low** | **both**>]
  [**-input_power_pin** *power_pin*] [**-output_power_pin** *power_pin*]
  [**-ground** *ground_pin*]
  [**-valid_location** <**source** | **sink** | **either** | **any**>]

Figure G.14 shows a power domain at `0.8 V` and one at `1.2 V`. The ground voltage for both domains is `0.0 V`. In this case, data signals going from the domain at `0.8 V` to the domain at `1.2 V` need a power level-shifter with direction `low_to_high`, while data signals going from the domain at `1.2 V` to the domain at `0.8 V` need a power level-shifter with direction `high_to_low`.

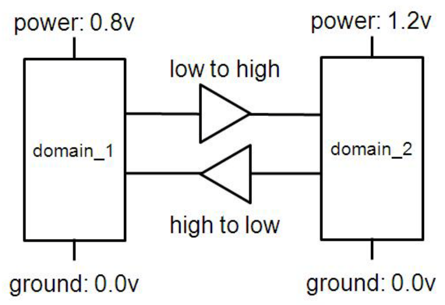


Figure G.14—Power level-shifter

The following commands can be used to model these power level-shifters:

```
    define_level_shifter_cell -cells low_to_high_power \
        -input_voltage_range {{0.8 1.0}} -output_voltage_range {{1.0 1.2}} \
        -input_power_pin VDD_IN -output_power_pin VDD_OUT -ground VSS_IN \
        -direction low_to_high -valid_location source

    Liberty Model :
library(mylib) {

  voltage_map(VDD_IN,  0.8);  /* primary power */
  voltage_map(VDD_OUT, 1.2); /* primary power */
  voltage_map(VSS_IN,  0.0);  /* primary ground */

  cell(up_shifter) {
    is_level_shifter : true;
    level_shifter_type :  LH ;
    pg_pin(VDD_IN) {
      voltage_name : VDD_IN;
      pg_type : primary_power;
      std_cell_main_rail : true;
    }
    pg_pin(VDD_OUT) {
      voltage_name : VDD_OUT;
      pg_type : primary_power;
    }
    pg_pin(VSS_IN) {
      voltage_name : VSS_IN;
      pg_type : primary_ground;
    }
    pin(IN) {
      direction : input;
      related_power_pin : VDD_IN;
      related_ground_pin : VSS_IN;
      input_voltage_range ( 0.8 , 1.0);
    }
    pin(OUT) {
      direction : output;
      related_power_pin : VDD_OUT;
      related_ground_pin : VSS_IN;
      function : "IN";
      power_down_function : "!VDD_IN + !VDD_OUT + VSS_IN";
      output_voltage_range (1.0 , 1.2);
    } /* end pin group */
  } /* end cell group */
} /* end library group */

    define_level_shifter_cell -cells high_to_low_power \
       -input_voltage_range {{1.0 1.2}} -output_voltage_range {{0.8 1.0}} \
       -input_power_pin VDD_IN -output_power_pin VDD_OUT -ground VSS_IN \
       -direction high_to_low -valid_location source

Liberty Model :
library(mylib) {

  voltage_map(VDD_IN,  1.2);  /* primary power */
  voltage_map(VDD_OUT, 0.8); /* primary power */
  voltage_map(VSS_IN,  0.0);  /* primary ground */

  cell(down_shifter) {
    is_level_shifter : true;
    level_shifter_type :  HL ;

    pg_pin(VDD_IN) {
      voltage_name : VDD_IN;
      pg_type : primary_power;
      std_cell_main_rail : true;
    }
    pg_pin(VDD_OUT) {
      voltage_name : VDD_OUT;
      pg_type : primary_power;
    }
    pg_pin(VSS_IN) {
      voltage_name : VSS_IN;
      pg_type : primary_ground;
    }
    pin(IN) {
      direction : input;
      related_power_pin : VDD_IN;
      related_ground_pin : VSS_IN;
      input_voltage_range ( 1.0 , 1.2);
    }
    pin(OUT) {
      direction : output;
      related_power_pin : VDD_OUT;
      related_ground_pin : VSS_IN;
      function : "IN";
      power_down_function : "!VDD_IN + !VDD_OUT + VSS_IN";
      output_voltage_range (0.8 , 1.0);
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.4.3 Modeling a ground level-shifter

A ground level-shifter passes signals between portions of the design that operate on different ground voltages, but using the same power voltages. To model a ground level-shifter, use the following options from the **define_level_shifter_cell** command (see 7.5):

define_level_shifter_cell
  -cells *cell_list*
  -ground_input_voltage_range {{*lower_bound upper_bound*}*}
  -ground_output_voltage_range {{*lower_bound upper_bound*}*}
  [**-direction** <**low_to_high** | **high_to_low** | **both**>]
  [**-input_ground_pin** *power_pin*] [**-output_ground_pin** *power_pin*]
  [**-power** *power_pin*] [**-valid_location** <**source** | **sink** | **either** | **any**>]

The two power domains in Figure G.15 have the same power supply `1.2 V`. However, the ground voltage for the first domain is at `0.0 V`, while the ground voltage for the second domain is at `0.5 V`. The direction of a level-shifter indicates the difference between the voltage swing of the driver and the voltage swing of the receiver. As a result, for data signals going from the domain with ground voltage `0.0 V` to the domain with ground voltage `0.5 V`, a ground level-shifter with direction `high_to_low` is required. Similarly, for data signals going from the domain with ground voltage `0.5 V` to the domain with ground voltage `0.0 V`, a ground level-shifter with direction `low_to_high` is required.

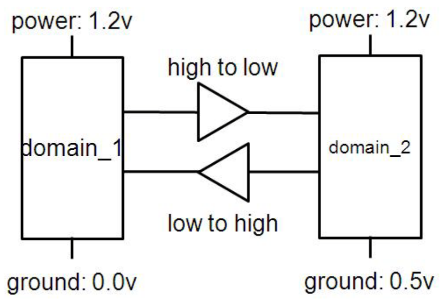


Figure G.15—Ground level-shifter

The following commands can be used to model these ground level-shifters:

```
    define_level_shifter_cell -cells high_to_low_ground \
        -ground_input_voltage_range {{0.0 0.1}} \
        -ground_output_voltage_range {{0.4 0.5}} \
        -input_ground_pin VSS_IN -output_ground_pin VSS_OUT -power VDD_IN \
        -direction high_to_low -valid_location source

    Liberty Model:
library(mylib) {

  voltage_map(VDD_IN, 1.2);  /* primary power */
  voltage_map(VSS_IN, 0.0);  /* primary ground */
  voltage_map(VSS_OUT, 0.5); /* primary ground */

  cell(down_shift) {
    is_level_shifter : true;
    level_shifter_type :  HL ;
    pg_pin(VDD_IN) {
      voltage_name : VDD_IN;
      pg_type : primary_power;
    }
    pg_pin(VSS_IN) {
      voltage_name : VSS_IN;
      pg_type : primary_ground;
      std_cell_main_rail : true;
    }
    pg_pin(VSS_OUT) {
      voltage_name : VSS_OUT;
      pg_type : primary_ground;
    }
    pin(IN) {
      direction : input;
      related_power_pin : VDD_IN;
      related_ground_pin : VSS_IN;
      ground_input_voltage_range : "(0.0, 0.1)";
    }
    pin(OUT) {
      direction : output;
      related_power_pin : VDD_IN;
      related_ground_pin : VSS_OUT;
      function : "IN";
      power_down_function : "!VDD_IN + VSS_IN + VSS_OUT";
      ground_output_voltage_range : "(0.4 , 0.5)";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/

   define_level_shifter_cell -cells low_to_high_ground \
      -ground_input_voltage_range {{0.4 0.5}} \
      -ground_output_voltage_range {{0.0 0.1}} \
      -input_ground_pin VSS_IN -output_ground_pin VSS_OUT -power VDD_IN \
      -direction low_to_high -valid_location source

Liberty Model:
library(mylib) {

  voltage_map(VDD_IN,  1.2);  /* primary power */
  voltage_map(VSS_IN,  0.5);  /* primary ground */
  voltage_map(VSS_OUT, 0.0);  /* primary ground */

  cell(up_shift) {
    is_level_shifter : true;
    level_shifter_type :  LH ;

    pg_pin(VDD_IN) {
      voltage_name : VDD_IN;
      pg_type : primary_power;
    }
    pg_pin(VSS_IN) {
      voltage_name : VSS_IN;
      pg_type : primary_ground;
      std_cell_main_rail : true;
    }
    pg_pin(VSS_OUT) {
      voltage_name : VSS_OUT;
      pg_type : primary_ground;
    }
    pin(IN) {
      direction : input;
      related_power_pin : VDD_IN;
      related_ground_pin : VSS_IN;
      ground_input_voltage_range : "(0.4 , 0.5)";
    }
    pin(OUT) {
      direction : output;
      related_power_pin : VDD_IN;
      related_ground_pin : VSS_OUT;
      function : "IN";
      power_down_function : "!VDD_IN + VSS_OUT + VSS_IN";
      ground_output_voltage_range : "(0.0 , 0.1)";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.4.4 Modeling a power and ground level-shifter

A power and ground level-shifter passes signals between portions of the design that operate on different power and ground voltages. To model a ground level-shifter, use the following options from the **define_level_shifter_cell** command (see 7.5):

> **define_level_shifter_cell**
> **-cells** *cell_list*
> **-input_voltage_range** {{*lower_bound upper_bound*}*}
> **-output_voltage_range** {{*lower_bound upper_bound*}*}
> **-ground_input_voltage_range** {{*lower_bound upper_bound*}*}
> **-ground_output_voltage_range** {{*lower_bound upper_bound*}*}
> [**-direction** <**low_to_high** | **high_to_low** | **both**>]
> [**-input_power_pin** *power_pin*] [**-output_power_pin** *power_pin*]
> [**-input_ground_pin** *power_pin*] [**-output_ground_pin** *power_pin*]
> [**-valid_location** <**source** | **sink** | **either** >]

The two power domains in Figure G.16 have different power and ground voltages. `domain_1` is the region where power is `0.8 V` and ground is `0.5 V`. `domain_2` is the region where power is `1.2 V` and ground is `0 V`. As shown, the voltage swing of the `domain_1` is `0.3 V` and the voltage swing of the `domain_2` is `1.2 V`. As a result, a `low_to_high` direction power and ground level-shifter is needed going from `domain_1` to `domain_2`. Similarly, going from `domain_2` to `domain_1` requires a power and ground level-shifter in the `high_to_low` direction.

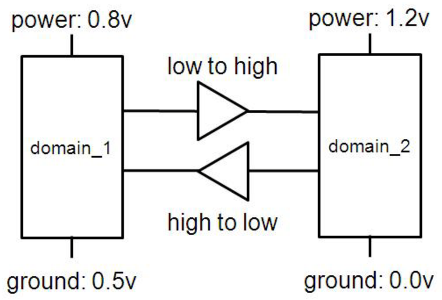


Figure G.16—Power and ground level-shifter

The following commands model the power and ground level-shifter to go from `domain_1` to `domain_2`:

```
define_level_shifter_cell -cells low_to_high \
   -input_voltage_range {{0.8 1.0}} -output_voltage_range {{1.0 1.2}} \
   -ground_input_voltage_range {{0.4 0.5}} \
   -ground_output_voltage_range {{0.0 0.1}} \
   -input_ground_pin VSS_IN -output_ground_pin VSS_OUT \
   -input_power_pin VDD_IN -output_power_pin VDD_OUT \
   -direction low_to_high -valid_location source
```

Liberty model

```
library(mylib) {

  voltage_map(VDD_IN,  0.8);  /* primary power */
  voltage_map(VDD_OUT, 1.2);  /* primary power */
  voltage_map(VSS_IN,  0.5);  /* primary ground */
  voltage_map(VSS_OUT, 0.0);  /* primary ground */
  cell(up_shift {
    is_level_shifter : true;
    level_shifter_type :  LH ;
    pg_pin(VDD_IN) {
      voltage_name : VDD_IN;
      pg_type : primary_power;
      std_cell_main_rail : true;
    }
    pg_pin(VDD_OUT) {
      voltage_name : VDD_OUT;
      pg_type : primary_power;
    }
    pg_pin(VSS_IN) {
      voltage_name : VSS_IN;
      pg_type : primary_ground;
      std_cell_main_rail : true;
    }
    pg_pin(VSS_OUT) {
      voltage_name : VSS_OUT;
      pg_type : primary_ground;
    }
    pin(IN) {
      direction : input;
      related_power_pin : VDD_IN;
      related_ground_pin : VSS_IN;
      input_voltage_range (0.8, 1.0);
      ground_input_voltage_range : "(0.4, 0.5)";
    } /* end pin group */
    pin(OUT) {
      direction : output;
      related_power_pin : VDD_OUT;
      related_ground_pin : VSS_OUT;
      function : "IN";
      power_down_function : "!VDD_IN + !VDD_OUT + VSS_IN + VSS_OUT";
      output_voltage_range (1.0, 1.2);
      ground_output_voltage_range : "(0.0 , 0.1)";
    } /* end pin group */
  } /* end cell group */
} /* end library group */
```

The following commands model the power and ground level shift to go from `domain_2` to `domain_1`:

```
   define_level_shifter_cell -cells high_to_low \
      -input_voltage_range {{1.0 1.2}} -output_voltage_range {{0.8 1.0}} \
      -ground_input_voltage_range {{0.0 0.1}} \
      -ground_output_voltage_range {{0.4 0.5}} \
      -input_ground_pin VSS_IN -output_ground_pin VSS_OUT \
      -input_power_pin VDD_IN -output_power_pin VDD_OUT \
      -direction high_to_low -valid_location sink
```

Liberty model

```
library(mylib) {

  voltage_map(VDD_IN, 1.2);  /* primary power */
  voltage_map(VDD_OUT, 0.8);  /* primary power */
  voltage_map(VSS_IN,  0.0);  /* primary ground */
  voltage_map(VSS_OUT, 0.5);  /* primary ground */
  define(ground_input_voltage_range, pin, string);
  define(ground_output_voltage_range, pin, string);

  cell(down_shift) {
    is_level_shifter : true;
    level_shifter_type :  HL ;
    pg_pin(VDD_IN) {
      voltage_name : VDD_IN;
      pg_type : primary_power;
    }
    pg_pin(VDD_OUT) {
      voltage_name : VDD_OUT;
      pg_type : primary_power;
      std_cell_main_rail : true;
    }
    pg_pin(VSS_IN) {
      voltage_name : VSS_IN;
      pg_type : primary_ground;
    }
    pg_pin(VSS_OUT) {
      voltage_name : VSS_OUT;
      pg_type : primary_ground;
      std_cell_main_rail : true;
    }
    pin(IN) {
      direction : input;
      related_power_pin : VDD_IN;
      related_ground_pin : VSS_IN;
      input_voltage_range (1.0, 1.2);
      ground_input_voltage_range : "(0.0 , 0.1)";
    }
    pin(OUT) {
      direction : output;
      related_power_pin : VDD_OUT;
      related_ground_pin : VSS_OUT;
      function : "IN";
      power_down_function : "!VDD_IN + !VDD_OUT + VSS_OUT + VSS_IN";
      output_voltage_range (0.8, 1.0);
      ground_output_voltage_range : "(0.4 , 0.5)";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.4.5 Modeling an enabled level-shifter

An *enabled level-shifter* is the level-shifter with an enable pin, which allows the level-shifter to be used for isolation purpose in some cases. To model such a cell, use the **define_level_shifter_cell** command with the **-enable** option (see 7.5).

This type of cell uses an enable pin to control the voltage shifting. Typically, the enable pin is related to the output supplies of the level-shifter. In other words, the enable control needs to have the same voltage as the receiving domain. If both domains are powered on, then the enable can be tied to a constant, such that the level-shifter is always active.

To model an isolation-level-shifter combo cell, see G.4.9.

G.4.5.1 Modeling an enabled power level-shifter

Assume the power level-shifter shown in Figure G.14 also has an enable pin to enable the level-shifting functionality, as shown in Figure G.17.

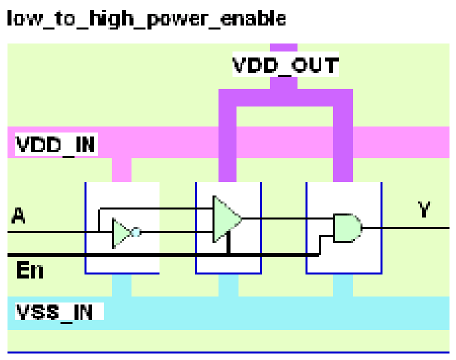


Figure G.17—Enabled power level-shifter

In this cell, when the enable signal `En` is inactive (at `logic 0`), it protects the level-shifter cell when the input power supply is powered down and causes the output to be a specific logic value determined by its functionality. `VLO` and `VSS` are the primary power (low voltage) and ground pin, respectively, and `VHI` is the additional power pin (high voltage). As it is indicated by the primary power connection, the cell needs to be placed in the low-voltage domain. For such a cell to be used for isolation purposes when the driving domain is switched off using a header power switch, its input power pin needs to be connected to the primary power net of the driving domain because the driver of the level-shifter data pin is not protected, e.g., the inverter connected to `A`. In this case, the definition should be adjusted as follows:

```
define_level_shifter_cell -cells low_to_high_power_enable \
   -input_voltage_range {{0.8 1.0}} -output_voltage_range {{1.0 1.2}} \
   -input_power_pin VDD_IN -output_power_pin VDD_OUT -ground VSS_IN \
   -direction low_to_high -valid_location source \
   -enable En
```

The enable pin is related to the output supplies of the level-shifter.

Liberty model

```
library(mylib) {

 voltage_map(VDD_IN,   0.8);  /* primary power */
 voltage_map(VDD_OUT, 1.2);   /* primary power */
 voltage_map(VSS_IN,   0.0);  /* primary ground */

 cell(up_shift) {
   is_level_shifter : true;
   level_shifter_type :  LH ;

   pg_pin(VDD_IN) {
     voltage_name : VDD_IN;
     pg_type : primary_power;
     std_cell_main_rail : true;
   }
   pg_pin(VDD_OUT) {
     voltage_name : VDD_OUT;
     pg_type : primary_power;
   }
   pg_pin(VSS_IN) {
     voltage_name : VSS;
     pg_type : primary_ground;
   }
   pin(A) {
     direction : input;
```

```
      related_power_pin : VDD_IN;
      related_ground_pin : VSS_IN;
      input_voltage_range ( 0.8 , 1.0);
      level_shifter_data_pin : true;
    }
    pin(En) {
      direction : input;
      related_power_pin : VDD_OUT;
      related_ground_pin : VSS_IN;
      input_voltage_range ( 1.0 , 1.2);
      level_shifter_enable_pin : true;
    }
    pin(Y) {
      direction : output;
      related_power_pin : VDD_OUT;
      related_ground_pin : VSS_IN;
      function : "A * En";
      power_down_function : "!VDD_IN + !VDD_OUT + VSS_IN";
      output_voltage_range (1.0 , 1.2);
    } /* end pin group */
  } /* end cell group */
} /* end library group */
```

G.4.5.2 Modeling an enabled ground level-shifter

Assume the ground level-shifter shown in Figure G.15 also has an enable pin to enable the level-shifting functionality. `VDD` and `VSS_IN` are the primary power and ground pin (for higher ground voltage), respectively, and `VSS_OUT` is the additional ground pin (for normal ground voltage). The enable pin connection is analogous to the connection of the enabled power level-shifter in Figure G.16. In this case, the definition should be adjusted as follows:

```
define_level_shifter_cell -cells low_to_high_ground_enable \
    -ground_input_voltage_range {{0.4 0.5}} \
    -ground_output_voltage_range {{0.0 0.1}} \
    -input_ground_pin VSS_IN -output_ground_pin VSS_OUT -power VDD \
    -direction low_to_high -valid_location source \
    -enable en
```

The enable pin is related to the output supplies of the level-shifter.

Liberty model—Low to high ground enable level-shifter

```
library(mylib) {

  voltage_map(VDD,     1.2);     /* primary power */
  voltage_map(VSS_IN,  0.5);  /* primary ground */
  voltage_map(VSS_OUT, 0.0);  /* primary ground */

  cell(up_shift) {
    is_level_shifter : true;
    level_shifter_type :  LH ;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(VSS_IN) {
      voltage_name : VSS_IN;
      pg_type : primary_ground;
      std_cell_main_rail : true;
    }
    pg_pin(VSS_OUT) {
      voltage_name : VSS_OUT;
      pg_type : primary_ground;
    }
    pin(A) {
      direction : input;

      related_power_pin : VDD;
      related_ground_pin : VSS_IN;
      ground_input_voltage_range : "(0.4, 0.5)";
    } /* end pin group */
    pin(en) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS_OUT;
      ground_input_voltage_range : "(0.0, 0.1)";
      level_shifter_enable_pin : true;
    } /* end pin group */
    pin(Z) {
      direction : output;
      related_power_pin : VDD;
      related_ground_pin : VSS_OUT;
      function : "A * en";
      power_down_function : "!VDD_IN + VSS_IN + VSS_OUT";
      ground_output_voltage_range : "(0.0 , 0.1)";
    } /* end pin group */
  } /* end cell group */
} /* end library group */
```

Liberty model—High to low ground enable level-shifter

```
library(mylib) {

  voltage_map(VDD,    1.2);  /* primary power */
  voltage_map(VSS_IN, 0.0);  /* primary ground */
  voltage_map(VSS_OUT, 0.5); /* primary ground */

  cell(down_shift) {
    is_level_shifter : true;
    level_shifter_type :  HL ;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(VSS_IN) {
      voltage_name : VSS_IN;
      pg_type : primary_ground;
      std_cell_main_rail : true;
    }
    pg_pin(VSS_OUT) {
      voltage_name : VSS_OUT;
      pg_type : primary_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS_IN;
      ground_input_voltage_range : "(0.0 , 0.1)";
    }
    pin(en) {
      direction : input;
      related_power_pin : VDD;
```

```
        related_ground_pin : VSS_OUT;
        ground_input_voltage_range : "(0.4 , 0.5)";
        level_shifter_enable_pin : true;
      }
      pin(Z) {
        direction : output;
        related_power_pin : VDD;
        related_ground_pin : VSS_OUT;
        function : "A * en";
        power_down_function : "!VDD + VSS_OUT + VSS_IN";
        ground_output_voltage_range : "(0.4 , 0.5)";
      } /* end pin group */
    } /* end cell group */
  } /* end library group*/
```

G.4.6 Modeling a bypass level-shifter

To model a level-shifter whose level-shifting functionality can be bypassed under certain conditions, use the **define_level_shifter_cell** command with the **-bypass_enable** option (see 7.5).

An example of such a cell is shown in Figure G.18. When the `bp_enable` signal is *True*, the level-shifting functionality is bypassed and the signal `OUT` comes from the top buffer.

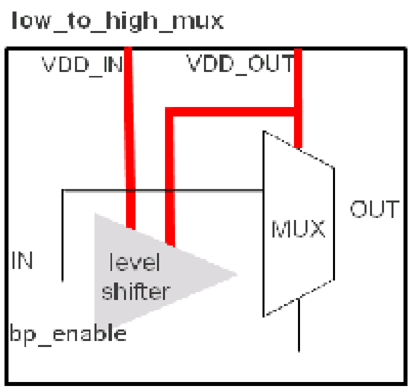


Figure G.18—Bypass level-shifter cell

The following command can be used to describe a bypass level-shifter:

```
    define_level_shifter_cell -cells low_to_high_mux \
        -input_voltage_range {{0.8 1.0}} -output_voltage_range {{1.0 1.2}} \
        -input_power_pin VDD_IN -output_power_pin VDD_OUT -ground VSS \
        -direction low_to_high -valid_location source -bypass_enable bp_enable
```

To apply such a cell for a specific level-shifter strategy, use the **-port_map** option of the **use_interface_cell** command (see 6.55) to explicitly describe the pin connection for the bypass enable pin of the cell.

Liberty model

```
library(mylib) {

  voltage_map(VDD_IN,  0.8);      /* primary power */
  voltage_map(VDD_OUT, 1.2);      /* primary power */
  voltage_map(VSS,     0.0);     /* primary power */

  cell(low_to_high_mux) {
    is_level_shifter : true;
    level_shifter_type :  LH ;
    pg_pin(VDD_IN) {
      voltage_name : VDD_IN;
      pg_type : primary_power;
      std_cell_main_rail : true;
    }
    pg_pin(VDD_OUT) {
      voltage_name : VDD_OUT;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
   pin(IN) {
      direction : input;
      related_power_pin : VDD_IN;
      related_ground_pin : VSS;
      input_voltage_range ( 0.8 , 1.0 );
    }
    pin(INT) {
      direction : internal;
      related_power_pin : VDD_OUT;
      related_ground_pin : VSS;
      output_voltage_range (1.0 , 1.2);
      function : "IN";
    }
    pin(bp_enable) {
      direction : input;
      related_power_pin : VDD_IN;
      related_ground_pin : VSS;
      level_shifter_enable_pin : true;
      input_voltage_range ( 0.8 , 1.0 );
    }
/* When bp_enable is logic high then signals IN and OUT will have the same
voltage value of 1.0v hence is a buffer functionality */
    pin(OUT) {
      direction : output;
      related_power_pin : VDD_OUT;
      related_ground_pin : VSS;
      function : " (!bp_enable * INT + bp_enable * IN) ";
      power_down_function : "!VDD_IN + !VDD_OUT + VSS";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.4.7 Modeling a multi-stage level-shifter

When the voltage difference between the driving (or originating) and receiving (or destination) power domains is large, multiple level-shifters or a single multi-stage level-shifter might be required. To model a single multi-stage level-shifter cell, define the level-shifter cell using the **define_level_shifter_cell** command with the **-multi_stage** option (see 7.5) to identify the stage of the multi-stage level-shifter to which this definition (command) applies.

For a level-shifter cell with *N* stages, *N* definitions shall be specified for the same cell. Each definition needs to associate a number from `1` to *`N`* for this option to indicate the corresponding stage of this definition. A definition cannot have the same stage defined twice.

An example of a single multi-stage level-shifter cell is shown in Figure G.19.

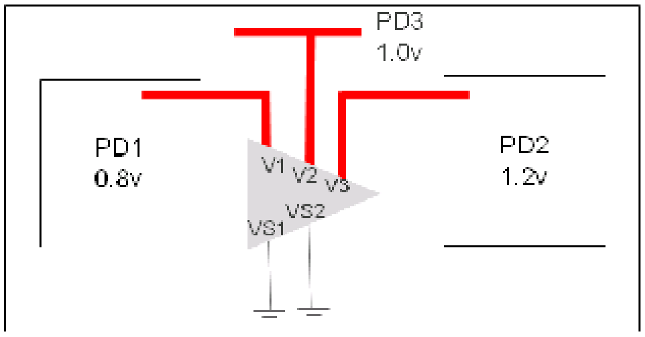


Figure G.19—Multi-stage level-shifter

The following commands can be used to describe the single level-shifter cell shown in Figure G.19:

```
define_level_shifter_cell -cells m_stage_ls -multi_stage 1 -input_power_pin
   V1\
-output_power_pin V2 -input_ground_pin VS1 -output_ground_pin VS2
define_level_shifter_cell -cells m_stage_ls -multi_stage 2 -input_power_pin
   V2\
-input_ground_pin VS2 -output_voltage_pin V3 -output_ground_pin VS2
```

To apply such a cell for a specific level-shifter strategy, use the **-port_map** option of the **use_interface_cell** command (see 6.55) to explicitly describe the pin connections.

Liberty model

```
library(mylib) {

  voltage_map(V1,  0.8);   /* primary power */
  voltage_map(V2,  1.0);   /* primary power */
  voltage_map(V3,  1.2);   /* primary power */
  voltage_map(VS1, 0.0);   /* primary ground */
  voltage_map(VS2, 0.0);   /* primary ground */

  cell(m_stage_ls) {
    is_level_shifter : true;
    level_shifter_type :  LH ;
    pg_pin(V1) {
      voltage_name : V1;
      pg_type : primary_power;
      std_cell_main_rail : true;
    }
    pg_pin(V2) {
      voltage_name : V2;
      pg_type : primary_power;
    }
    pg_pin(V3) {
      voltage_name : V3;
      pg_type : primary_power;
    }
    pg_pin(VS1) {
      voltage_name : VS1;
      pg_type : primary_ground;
    }
    pg_pin(VS2) {
      voltage_name : VS2;
      pg_type : primary_ground;
    }
    pin(A) {
      direction : input;
      related_power_pin : V1;
      related_ground_pin : VS1;
    }
    pin(INT) {
      direction : internal;
      related_power_pin : V2;
      related_ground_pin : VS2;
      function : "A";
    }
    pin(Z) {
      direction : output;
      related_power_pin : V3;
      related_ground_pin : VS2;
      function : "INT";
      power_down_function : "!V1 + !V2 + !V3 + VS1 + VS2";
    } /* end pin group */
  } /* end cell group */
} /* end library group*/
```

G.4.8 Modeling a multi-bit level-shifter cell

A multi-bit level-shifter cell has multiple pairs of input and output pins with each pair serving as a single-bit level-shifter. An example is shown in Figure G.20.

For the following multi-bit level-shifter cells, there is no difference in modeling such a multi-bit cell with respect to a single-bit level-shifter cell:

— Multi-bit simple level-shifter without an enable pin

— Multi-bit enable level-shifter with the same enable pin for all bits

If the cell has different enable pins for the input and output pairs, model the cell using the **define_level_shifter_cell** command with the **-pin_groups** option (see 7.5).

The following command can be used to describe the multi-bit level-shifter cell shown in Figure G.20:

```
define_level_shifter_cell -cells multi_bit_en \
   -input_voltage_range {{0.8 1.0}} -output_voltage_range {{1.0 1.2}} \
   -input_power_pin VDD_IN -output_power_pin VDD_OUT -ground VSS \
   -direction low_to_high -valid_location source \
   -pin_groups {{in1 out1 en1} {in2 out2 en1} {in3 out3 en2}}
```

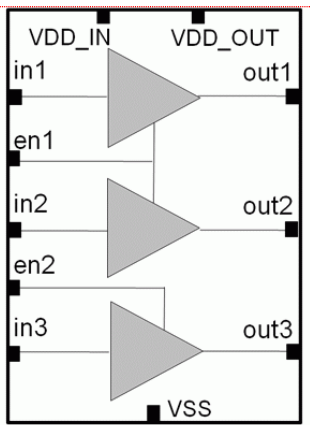


Figure G.20—Multi-bit level-shifter

Liberty model

```
library (mylib) {

  voltage_map(VDD_IN,  0.8);  /* primary power */
  voltage_map(VDD_OUT, 1.2);  /* primary power */
  voltage_map(VSS,     0.0);  /* primary ground */

  cell ("multi_bit_en") {
    is_level_shifter : true;
    pg_pin (VDD_IN) {
      voltage_name : VDD_IN;
      pg_type : primary_power;
      std_cell_main_rail : true;
    }
    pg_pin (VDD_OUT) {
      voltage_name : VDD_OUT;
      pg_type : primary_power;
    }
    pg_pin (VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    bundle (in) {
      members (in1, in2, in3);
      direction : input;
      related_power_pin : VDD_IN;
      related_ground_pin : VSS;
      level_shifter_data_pin : true;
    } /* end bundle group */
    pin (en1) {
      level_shifter_enable_pin : true;
      direction : input;
      related_power_pin : VDD_OUT;
      related_ground_pin : VSS;
    }
    pin (en2) {
      level_shifter_enable_pin : true;
      direction : input;
      related_power_pin : VDD_OUT;
     related_ground_pin : VSS;
    }
    bundle (out) {
      members (out1, out2, out3);
      direction : output;
      related_power_pin : VDD_OUT;
      related_ground_pin : VSS;
     power_down_function : "VDD_IN + !VDD_OUT + VSS";
      pin (out1) {
        direction : output;
        function : "in1 * en1";
    }/* end pin group */
    pin (out2) {
      direction : output;
      function : "in2 * en1";
    } /* end pin group */
    pin (out3) {
      direction : output;
      function : "in3 * en2";
    } /* end pin group */
  } /* end bundle group */
 } /* end cell group */
} /* end library group */
```

G.4.9 Modeling an isolation level-shifter combo cell

A combo cell isolates or protects the input when the driving logic is powered down and generates an output isolation value at the same voltage as the output supply of the cell. Typically, the enable pin is related to the input supplies of the cell. The most common combo cells are the isolation cells with high-to-low shifting capabilities.

Modeling a combo cell requires two commands. For example, to model an isolation cell for power-switchable domain that is also a power level-shifter, use the following definitions:

define_isolation_cell
  -cells *cell_list*
  {**-enable** *pin* | **-no_enable** <**high** | **low** | **hold**>}
  -power_switchable *power_pin*
  -power *power_pin* **-ground** *ground_pin*
  [**-valid_location** <**source** | **sink**>]

define_level_shifter_cell
  -cells *cell_list*
  -input_voltage_range {{*lower_bound upper_bound*}*}
  -output_voltage_range {{*lower_bound upper_bound*}*}
  -direction high_to_low
  [**-input_power_pin** *power_pin*] [**-output_power_pin** *power_pin*]
  [**-ground_pin** *power_pin*] [**-valid_location** <**source** | **sink**>]
  [**-always_on_pins** *pin_list*]

NOTE—The **-enable** option cannot be used in the **define_level_shifter_cell** definition. In addition, the same value for the **-valid_location** option needs to be specified in both the **define_isolation_cell** and **define_level_shifter_cell** commands.

To model an enabled level-shifter, see G.4.5.

G.5 Modeling power-switch cells

G.5.1 Types of power-switch cells

To connect and disconnect the power (or ground) supply from the gates in internal switchable power domains, power-switch logic needs to be added. The following is a list of the most typical cells:

— Single-stage power-switch cell single transistor that controls the primary power supply to the logic of an internal switchable domain

— Single-stage ground-switch cell single transistor that controls the primary ground supply to the logic of an internal switchable domain

— Dual-stage power switch with a weak and strong transistor to control the primary power supply to the logic of an internal switchable domain

— Dual-stage ground switch with a weak and strong transistor to control the primary ground supply to the logic of an internal switchable domain

All types of power-switch cells are defined using the **define_power_switch_cell** command (see 7.6). The following subclauses indicate which command options to use for each type.

G.5.2 Modeling a single-stage power-switch cell

To model a single-stage power-switch cell, use the following options from the **define_power_switch_cell** command (see 7.6):

define_power_switch_cell
    -cells *cell_list* **-type header**
    -power_switchable *power_pin* **-power** *power_pin*
    -stage_1_enable *expression* [**-stage_1_output** *expression*]
    [**-ground** *ground_pin*]
    [**-always_on_pins** *pin_list*]

NOTE—The **-stage_1_output** and **-stage_1_ground** options do not need to be specified for an unbuffered power-switch cell.

Figure G.21 shows a power-switch cell with an internal buffer. `VIN` is the pin connected to the unswitched power. `VSW` is the pin connected to the switchable power that is connected to the logic. When the enable signal `Ei` is activated, the unswitched power is supplied to the logic. As shown in Figure G.22, this type of cell usually contains a buffer that allows multiple power-switch cells to be chained together to form a power-switch column or ring. However, the power and ground of this buffer need to be unswitchable.

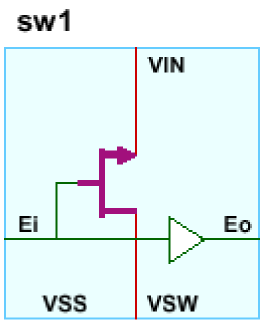


Figure G.21—Single-stage power switch

The following command models the power-switch cell shown in Figure G.21:

```
    define_power_switch_cell -cells sw1 \
        -stage_1_enable Ei -stage_1_output Eo \
        -type header -power_switchable VSW -power VIN -ground VSS

Liberty Model :
library(mylib) {

  voltage_map(VIN, 1.0);  /* primary power */
  voltage_map(VSW, 1.0);  /* internal power */
  voltage_map(VSS, 0.0);  /* primary ground */

  /* templates */
  lu_table_template (c_grain) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
    index_1("0.0, 0.2, 0.5, 1.2");
    index_2("0.0, 0.5, 1.08, 1.2");
  }
  cell(sw1) {
    switch_cell_type : coarse_grain;
    pg_pin(VIN) {
      voltage_name : VIN;
      pg_type : primary_power;
      direction : input;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
      direction : inout;
    }
    pg_pin (VSW) {
      voltage_name : VSW;
      pg_type : internal_power;
      switch_function : "!Ei";
      pg_function : "VIN";
      direction : output;
    }
    dc_current (c_grain) {
      related_switch_pin : Ei;
      related_pg_pin : VIN;
      related_internal_pg_pin : VSW;
      values ("0.01, 0.002, 0.003, 0.0005", \
              "0.01, 0.003, 0.001, 0.0006", \
              "0.03, 0.004, 0.002, 0.0006", \
              "0.05, 0.006, 0.003, 0.0008");
    } /* end dc_current group */
    pin (Ei) {
      switch_pin : true;
      related_power_pin : VIN;
      related_ground_pin : VSS;
    } /* end pin group */
    pin (Eo) {
      direction : output;
      function : "Ei";
      related_power_pin : VIN;
      related_ground_pin : VSS;
      power_down_function : "!VIN + VSS";
    } /* end pin group */
  } /* end cell group*/
} /* end library group*/
```

G.5.3 Modeling a power-switch cell with gate bias

To model a single-stage power-switch cell with gate bias, use the following options from the **define_power_switch_cell** command (see 7.6):

> **define_power_switch_cell**
>   **-cells** *cell_list* **-type header**
>   **-gate_bias_pin** *power_pin*
>   **-stage_1_enable** *expression* [**-stage_1_output** *expression*]
>   **-power_switchable** *power_pin* **-power** *power_pin*
>   **-ground** *ground_pin* [**-always_on_pins** *pin_list*]

Typically, the enable pin is related to the power and the ground pin. With gate bias, the enable pin is typically related to the gate bias pin and the ground. The voltage on the gate bias pin is larger than the voltage of the power pin. Such a cell creates less leakage power compared to the cell without gate bias.

In Figure G.22, the gate bias pin is `VGB`. Assume the input voltage `VIN` is at `1.2 V` and the gate bias pin is at `3.3 V`.

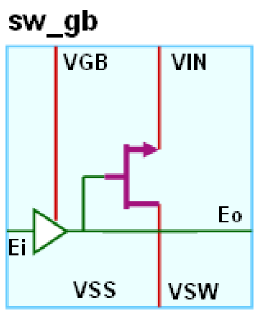


Figure G.22—Single-stage power switch with gate bias

The following command models the power-switch cell shown in Figure G.22:

```
define_power_switch_cell \
   -cells sw1 \
   -stage_1_enable Ei -stage_1_output Eo -gate_bias_pin VGB\
   -type header \
   -power_switchable VSW -power VIN -ground VSS
```

```
Liberty Model:
library(mylib) {

  voltage_map(VIN, 1.0);  /* primary power */
  voltage_map(VSW, 1.0);  /* internal power */
  voltage_map(VGB, 1.5);  /* primary power */
  voltage_map(VSS, 0.0);  /* primary ground */
  /* templates */
  lu_table_template (c_grain) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
    index_1("0.0, 0.2, 0.5, 1.2");
    index_2("0.0, 0.5, 1.08, 1.2");
  }
  cell(sw1) {
    switch_cell_type : coarse_grain;
    pg_pin(VIN) {
      voltage_name : VIN;
      pg_type : primary_power;
      direction : input;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
      direction : inout;
      related_bias_pin : VGB;
    }
    pg_pin (VSW) {
      voltage_name : VSW;
      pg_type : internal_power;
      switch_function : "!Ei";
      pg_function : "VIN";
      direction : output;
    }
    pg_pin (VGB) {
      voltage_name : VGB;
      pg_type : primary_power;
    }
    dc_current (c_grain) {
      related_switch_pin : internal;
      related_pg_pin : VIN;
      related_internal_pg_pin : VSW;
      values ("0.01, 0.002, 0.003, 0.0005", \
              "0.01, 0.003, 0.001, 0.0006", \
              "0.03, 0.004, 0.002, 0.0006", \
              "0.05, 0.006, 0.003, 0.0008");
    } /* end dc_current group */
    pin (Ei) {
      direction : input;
      switch_pin : true;
      related_power_pin : VGB;
      related_ground_pin : VSS;
    } /* end pin group */
    pin (internal) {
      direction : internal;
    } /* end pin group */
    pin (Eo) {
      direction : output;
      function : "Ei";
      related_power_pin : VIN;
      related_ground_pin : VSS;
      power_down_function : "!VGB + !VIN + VSS";
    } /* end pin group */
  } /*end cell group*/
} /* end library group*/
```

G.5.4 Modeling a single-stage ground-switch cell

To model a single-stage ground-switchable power-switch cell, use the following options from the **define_power_switch_cell** command (see 7.6):

> **define_power_switch_cell**
>   **-cells** *cell_list* **-type footer**
>   **-stage_1_enable** *expression* [**-stage_1_output** *expression*]
>   **-ground_switchable** *ground_pin* **-ground** *ground_pin*
>   **-power** *power_pin* [**-always_on_pins** *pin_list*]

Figure G.23 shows a ground-switch cell. `VSS` is the pin connected to the unswitched ground. `VSW` is the pin connected to the switchable ground that is connected to the logic. When the enable signal `Ei` is activated, the unswitched ground is supplied to the logic. As shown in Figure G.23, this type of cell usually contains a buffer that allows multiple ground-switch cells to be chained together to form a ground-switch column or ring. However, the power and ground of this buffer need to be unswitchable.

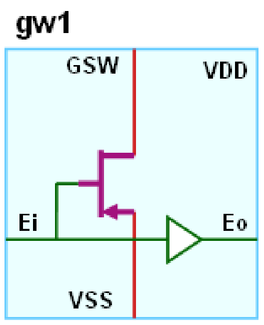


Figure G.23—Single-stage ground switch

The following command models the ground-switch cell shown in Figure G.23:

```
    define_power_switch_cell -cells gw1 \
       -stage_1_enable Ei -stage_1_output Eo \
       -type footer -ground_switchable GSW -ground VSS -power VDD
```

Liberty model

```
library(mylib) {

  voltage_map(VDD, 1.0);  /* primary power */
  voltage_map(GSW, 0.0);  /* Internal ground */
  voltage_map(VSS, 0.0);  /* primary ground */
  /* templates */
  lu_table_template (c_grain) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
    index_1("0.0, 0.2, 0.5, 1.2");
    index_2("0.0, 0.5, 1.08, 1.2");
  }
  cell(gw1) {
    switch_cell_type : coarse_grain;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
      direction : input;
    }
    pg_pin (GSW) {
      voltage_name : GSW;
      pg_type : internal_ground;
      switch_function : "!Ei";
      pg_function : "VSS";
      direction : output;
    }
    dc_current (c_grain) {
      related_switch_pin : Ei;
      related_pg_pin : VSS;
      related_internal_pg_pin : GSW;
      values ("0.01, 0.002, 0.003, 0.0005", \
              "0.01, 0.003, 0.001, 0.0006", \
              "0.03, 0.004, 0.002, 0.0006", \
              "0.05, 0.006, 0.003, 0.0008");
    }
    pin (Ei) {
      switch_pin : true;
      related_power_pin : VDD;
      related_ground_pin : VSS;
    }
    pin (Eo) {
      direction : output;
      function : "Ei";
      related_power_pin : VDD;
      related_ground_pin : VSS;
      power_down_function : "!VDD + VSS";
    } /* end pin group */
  } /*end cell group*/
} /* end library group*/
```

G.5.5 Modeling a dual-stage power-switch cell

To model a power-switch cell with two stages, use the following options from the **define_power_switch_cell** command (see 7.6):

define_power_switch_cell
  -cells *cell_list* **-type header**
  -power_switchable *power_pin* **-power** *power_pin*
  -stage_1_enable *expression* [**-stage_1_output** *expression*]
  -stage_2_enable *expression* [**-stage_2_output** *expression*]
  -ground *ground_pin* [**-always_on_pins** *pin_list*]

Figure G.24 shows a dual-stage power-switch cell. `VIN` is the pin connected to the unswitched power. `VSW` is the pin connected to the switchable power that is connected to the logic. Only when both enable signals `Ri` and `Ei` are activated can the unswitched power be supplied to the logic. The `Ri` enable signal drives the stage-1 (weak) transistor, which requires less current to restore the unswitched power. The `Ei` enable signal drives the stage-2 (strong) transistor, which requires more current to fully supply the unswitched power to the logic. This type of cell usually contains two buffers that allow multiple power-switch cells to be chained together to form a power-switch column or ring. However, the power and ground of these buffers need to be unswitchable.

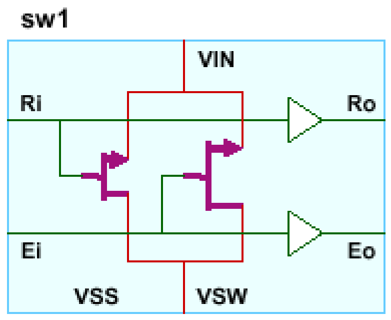


Figure G.24—Dual-stage power switch

The following command models the power-switch cell shown in Figure G.24:

```
    define_power_switch_cell -cells sw1 \
        -stage_1_enable Ri -stage_1_output Ro \
        -stage_2_enable Ei -stage_2_output Eo \
        -type header -power_switchable VSW -power VIN -ground VSS
```

Liberty model

```
library(mylib) {

  voltage_map(VIN, 1.0);  /* primary power */
  voltage_map(VSW, 1.0);  /* Internal power */
  voltage_map(VSS, 0.0);  /* primary ground */
  /* templates */
  lu_table_template (c_grain) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
    index_1("0.0, 0.2, 0.5, 1.2");
    index_2("0.0, 0.5, 1.08, 1.2");
  }
  cell(sw1) {
    switch_cell_type : coarse_grain;
    pg_pin(VIN) {
      pg_type    : primary_power;
      direction : input ;
      voltage_name : VIN;
    }
    pg_pin ( VSW ) {
      pg_type : internal_power;
      voltage_name : VSW;
      direction : output ;
      switch_function : "(Ei * Ri)";
      pg_function : "VIN" ;
    }
    pg_pin ( VSS ) {
      pg_type : primary_ground;
      voltage_name : VSS;
    }
    pin(Ei) {
      direction : input;
      related_power_pin : VIN;
      related_ground_pin : VSS;
      switch_pin : true;
    }
    pin(Ri) {
      direction : input;
```

```
        related_power_pin : VIN;
        related_ground_pin : VSS;
        switch_pin : true;
      }
      /* DC current table when first header is ON */
      dc_current (c_grain) {
       related_switch_pin : Ei;
       related_pg_pin : VIN;
       related_internal_pg_pin : VSW;
       values ("0.01, 0.002, 0.003, 0.0005", \
               "0.01, 0.003, 0.001, 0.0006", \
               "0.03, 0.004, 0.002, 0.0006", \
               "0.05, 0.006, 0.003, 0.0008");
      }
      /* DC current when second header is ON */
      dc_current (c_grain) {
       related_switch_pin : Ri;
       related_pg_pin : VIN;
       related_internal_pg_pin : VSW;
        values ("0.02, 0.003, 0.004, 0.0006", \
                "0.02, 0.004, 0.005, 0.0007", \
                "0.04, 0.005, 0.0001, 0.0008", \
                "0.06, 0.007, 0.008, 0.0009");
      } /* end dc_current group */
      pin(Eo) {
        direction : output;
        related_power_pin : VIN;
        related_ground_pin : VSS;
        function : "Ei";
        power_down_function : "!VIN + VSS";
      } /* end pin group */
      pin(Ro) {
        direction : output;
        related_power_pin : VIN;
        related_ground_pin : VSS;
        function : "Ri";
        power_down_function : "!VIN + VSS";
      } /* end pin group */
    } /*end cell group*/
  } /* end library group*/
```

G.5.6 Modeling a dual-stage ground-switch cell

To model a ground-switch cell with two stages, use the following options from the **define_power_switch_cell** command (see 7.6):

> **define_power_switch_cell**
>   **-cells** *cell_list* **-type footer**
>   **-ground_switchable** *ground_pin* **-ground** *ground_pin*
>   **-stage_1_enable** *expression* [**-stage_1_output** *expression*]
>   **-stage_2_enable** *expression* [**-stage_2_output** *expression*]
>   **-power** *power_pin* [**-always_on_pins** *pin_list*]

Figure G.25 shows a dual-stage ground-switch cell. `VSS` is the pin connected to the unswitched ground. `GSW` is the pin connected to the switchable ground that is connected to the logic. Only when both enable signals `Ri` and `Ei` are activated can the unswitched ground be supplied to the logic. The `Ri` enable signal drives the stage-1 (weak) transistor, which requires less current to restore the unswitched ground. The `Ei` enable signal drives the stage-2 (strong) transistor, which requires more current to fully supply the unswitched ground to the logic. This type of cell usually contains two buffers that allow multiple ground-switch cells to be chained together to form a ground-switch column or ring. However, the power and ground of these buffers need to be unswitchable.

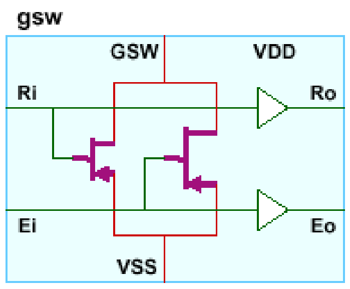


Figure G.25—Dual-stage ground switch

The following command models the ground-switch cell shown in Figure G.25:

```
    define_power_switch_cell -cells gsw \
        -stage_1_enable Ri -stage_1_output Ro \
        -stage_2_enable Ei -stage_2_output Eo \
        -type footer -ground_switchable GSW -ground VSS -power VDD
```

Liberty model

```
library(mylib) {

  voltage_map(VDD, 1.0);  /* primary power */
  voltage_map(GSW, 0.0);  /* Internal power */
  voltage_map(VSS, 0.0);  /* primary ground */
  /* templates */
  lu_table_template (c_grain) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
    index_1("0.0, 0.2, 0.5, 1.2");
    index_2("0.0, 0.5, 1.08, 1.2");
  }
  cell(gsw) {
    switch_cell_type : coarse_grain;
    pg_pin(VDD) {
      pg_type    : primary_power;
      voltage_name : VDD;
    }
    pg_pin ( GSW ) {
      pg_type : internal_ground;
      voltage_name : GSW;
      direction : output ;
      switch_function : "(!Ei * !Ri)";
      pg_function : "VSS" ;
    }
    pg_pin ( VSS ) {
      pg_type : primary_ground;
      voltage_name : VSS;
      direction : input ;
    }
    pin(Ei) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      switch_pin : true;
    }
    pin(Ri) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      switch_pin : true;
    }
    /* DC current atble when first header is ON */
    dc_current (c_grain) {
      related_switch_pin : Ei;
      related_pg_pin : VSS;
      related_internal_pg_pin : GSW;
      values ("0.01, 0.002, 0.003, 0.0005", \
              "0.01, 0.003, 0.001, 0.0006", \
              "0.03, 0.004, 0.002, 0.0006", \
              "0.05, 0.006, 0.003, 0.0008");
    }
    /* DC current table when second header is ON */
    dc_current (c_grain) {
      related_switch_pin : Ri;
      related_pg_pin : VSS;
      related_internal_pg_pin : GSW;
      values ("0.02, 0.003, 0.004, 0.0006", \
              "0.02, 0.004, 0.005, 0.0007", \
              "0.04, 0.005, 0.0001, 0.0008", \
              "0.06, 0.007, 0.008, 0.0009");
     } /* end dc_current group */
     pin(Eo) {
       direction : output;
       related_power_pin : VDD;
       related_ground_pin : VSS;
       function : "Ei";
       power_down_function : "!VDD + VSS";
     } /* end pin group */
     pin(Ro) {
       direction : output;
       related_power_pin : VDD;
       related_ground_pin : VSS;
       function : "Ri";
       power_down_function : "!VDD + VSS";
     } /* end pin group */
   } /*end cell group*/
 } /* end library group*/
```

G.6 Modeling state retention cells

G.6.1 Types of state retention cells

State retention cells are used for sequential cells to keep their previous state prior to power-down. The following is a list of the most typical state retention cells:

- State retention cell with explicit save control
- State retention cell with explicit restore control
- State retention cells with explicit save and restore controls
- State retention cells without explicit save or restore control

All types of state retention cells are defined using the **define_retention_cell** command (see 7.7). The following subclauses indicate which command options to use for each type.

G.6.2 State retention cell that restores when power is turned on

To model a state retention cell that saves the current value when the retention control pin becomes active while the power is on, retains the saved value when power is off, and restores the saved value when the power is turned on, use the following options from the **define_retention_cell** command (see 7.7):

define_retention_cell
  -cells *cell_list* [**-cell_type** *string*]
  -save_function {{**pin** <**high** | **low** | **posedge** | **negedge**}}
  [**-always_on_pins** *pin_list*]
  [**-clock_pin** *pin*]
  [**-restore_check** *expression*] [**-save_check** *expression*]
  [**-retention_check** *expression*] [**-hold_check** *pin_list*]
  [**-always_on_components** *component_list*]
  [**-power_switchable** *power_pin*] [**-ground_switchable** *ground_pin*]
  [**-power** *power_pin*] [**-ground** *ground_pin*]

Figure G.26 shows an example of such a cell.

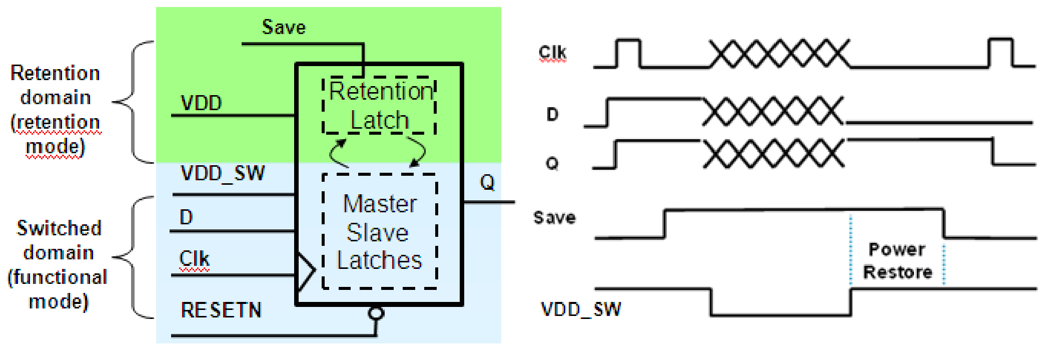


Figure G.26—State retention with save control

To model the cell shown in Figure G.26, use the following command:

```
define_retention_cell -cells SR1 \
    -clock_pin Clk \
    -save_function {save posedge} \
    -restore_check !Clk -save_check !Clk \
    -power_switchable VDD_SW \
    -power VDD -ground VSS
```

If the UPF retention strategy is specified as follows:

```
set_retention ret -domain PD \
    -save_signal {save save_net posedge} \
    -restore_signal {save_net negedge} \
    ...
```

then the retention cells specified above are used to implement the strategy.

Liberty model

```
library(mylib) {

  voltage_map (VDD, 1.0); /* backup power */
  voltage_map (VSW, 1.0); /* primary power */
  voltage_map (VSS, 0.0); /* primary ground */

  cell(SR1) {
    retention_cell : RET;
    pg_pin (VDD) {
      voltage_name : VDD;
      pg_type : backup_power;
    }
    pg_pin (VSS) {
     voltage_name : VSS;
     pg_type : primary_ground;
    }
    pg_pin (VSW) {
     voltage_name : VSW;
     pg_type : primary_power;
    }
    ff(Q1, QN1 ) {
      clocked_on : " Clk ";
      next_state : " D ";
      clear : "(!save * !Q2) + !RESETN";
      preset : "!save * Q2";
      clear_preset_var1 : "H";
      clear_preset_var2 : "H";
      power_down_function : "!VSW+VSS";
    }
    latch("Q2", "QN2") {
      enable : " save ";
      data_in : " Q1 ";
      power_down_function : "!VDD+VSS";
    }
    clock_condition() {
      clocked_on : "Clk";
      required_condition  :  "!save";  /*  cell  in  legal  state  when  save  =
logic_low */
      hold_state : "N"; /* retention data is restored to either master or slave
latch */
    }
    pin(save) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      retention_pin(save_restore, "0"); /* cell is in normal mode and works as
a D-flop when save is logic high */
      save_action  :  "H";  /*  The  save  happens  at  the  AO  latch  at  the  signal
leading edge */
      restore_action  :  "H";  /*  The  restore  happens  at  the  output  is  at  the
signal leading edge */
      save_condition : "!Clk"; /* side condition for successful save */
      restore_condition : "!Clk"; /* side condition for successful restore */
      restore_edge_type  :  "leading";  /*  Edge  when  the  cell  is  starting  to
restore */
    }
    retention_condition() {
     required_condition : "save";
      power_down_function : "!VSW + VSS";
    }
    clear_condition() { /* When clear asserts, save must be high to allow Low
value to be transferred to Flop output */
      input : "!RESETN";
      required_condition : "!save";
    }
    pin(D) {
      direction : input;
      related_power_pin : VSW;
      related_ground_pin : VSS;
    }
    pin(Clk) {
      direction : input;
      related_power_pin : VSW;
      related_ground_pin : VSS;
    }
    pin(RESETN) {
      direction : input;
      related_power_pin : VSW;
      related_ground_pin : VSS;
    }
    pin(Q) {
      direction : output;
      function : "Q1";
      power_down_function : "!VSW + VSS";
      related_power_pin : VSW;
      related_ground_pin : VSS;
    } /* end pin Group */
  } /* end cell group */
} /* end library group */
```

For a retention cell with output `Q` driven by a buffer powered by the retention supply (`VDD`), `Q` shall be specified in the **-always_on** option of the command, as follows:

```
   define_retention_cell -cells SR1 \
      -clock_pin Clk \
      -always_on_pins {Q}
      -save_function {save posedge} \
      -restore_check !Clk -save_check !Clk \
      -power_switchable VDD_SW \
      -power VDD -ground VSS
```

Such a cell shall then be used to implement a retention strategy specified with **-use_retention_as_primary**, such as:

```
   set_retention ret -domain PD \
      -save_signal {save save_net posedge} \
      -restore_signal {save_net negedge} \
      -use_retention_as_primary \
      …
```

Liberty model

```
library(mylib) {

  voltage_map (VDD, 1.0); /* backup power */
  voltage_map (VSW, 1.0); /* primary power */
  voltage_map (VSS, 0.0); /* primary ground */

  cell(SR1) {
    retention_cell : RET;
    pg_pin (VDD) {
      voltage_name : VDD;
      pg_type : backup_power;
    }
    pg_pin (VSS) {
     voltage_name : VSS;
     pg_type : primary_ground;
    }
    pg_pin (VSW) {
     voltage_name : VSW;
     pg_type : primary_power;
    }
    ff(Q1, QN1 ) {
      clocked_on : " Clk ";
      next_state : " D ";
      clear : "(!save * !Q2) + !RESETN";
      preset : "!save * Q2";
      clear_preset_var1 : "H";
      clear_preset_var2 : "H";
      power_down_function : "!VSW+VSS";
    }
    latch("Q2", "QN2") {
      enable : " save ";
      data_in : " Q1 ";
      power_down_function : "!VDD+VSS";
    }
    clock_condition() {
      clocked_on : "Clk";
      required_condition  :  "!save";  /*  cell  in  legal  state  when  save  =
logic_low */
      hold_state : "N"; /* retention data is restored to either master or slave
latch */
    }
    pin(save) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
     retention_pin(save_restore, "0"); /* cell is in normal mode and works as a
D-flop when save is logic high */
      save_action  :  "H";  /*  The  save  happens  at  the  AO  latch  at  the  signal
leading edge */
      restore_action  :  "H";  /*  The  restore  happens  at  the  output  is  at  the
signal leading edge */
      save_condition : "!Clk"; /* side condition for successful save */
      restore_condition : "!Clk"; /* side condition for successful restore */
      restore_edge_type  :  "leading";  /*  Edge  when  the  cell  is  starting  to
restore */
    }
    retention_condition() {
      required_condition : "save";
      power_down_function : "!VSW + VSS";
    }
    clear_condition()  {  /* When  clear  asserts,  save  must  be  high  to  allow  Low
value to be transferred to Flop output */

      input : "!RESETN";
      required_condition : "!save";
    }
    pin(D) {
      direction : input;
      related_power_pin : VSW;
      related_ground_pin : VSS;
    }
    pin(Clk) {
      direction : input;
      related_power_pin : VSW;
      related_ground_pin : VSS;
    }
    pin(RESETN) {
      direction : input;
      related_power_pin : VSW;
      related_ground_pin : VSS;
    }
    pin(Q) {
      direction : output;
      function : "Q1";
      power_down_function : "!VDD + !VSW + +VSS";
      related_power_pin : VDD;
      related_ground_pin : VSS;
    } /* end pin Group */
  } /* end cell group */
} /* end library group */
```

G.6.3 State retention cell that restores when control signal is deactivated

To model a state retention cell that saves the current value when the retention control pin becomes deactivated and restores the saved value when the control signal becomes activated, use the following options from the **define_retention_cell** command (see 7.7):

define_retention_cell
  -cells *cell_list* [**-cell_type** *string*]
  -restore_function {{**pin** <**high** | **low** | **posedge** | **negedge**}}
  [**-always_on_pins** *pin_list*]
  [**-clock_pin** *pin*]
  [**-restore_check** *expression*] [**-save_check** *expression*]
  [**-retention_check** *expression*] [**-hold_check** *pin_list*]
  [**-always_on_components** *component_list*]
  [**-power_switchable** *power_pin*] [**-ground_switchable** *ground_pin*]
  [**-power** *power_pin*] [**-ground** *ground_pin*]

Figure G.27 shows an example of such a cell.

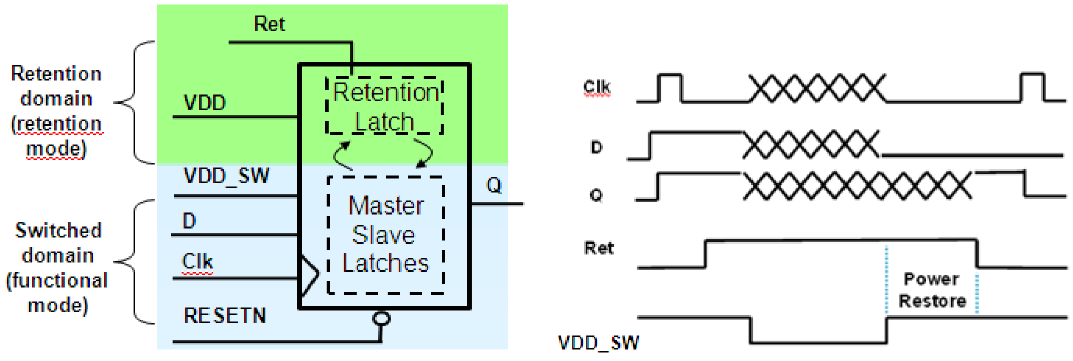


Figure G.27—State retention with restore control

To model the cell shown in Figure G.27, use the following command:

```
define_retention_cell -cells SR1 \
      -clock_pin Clk \
      -restore_function {Ret negedge} \
      -power_switchable VDD_SW \
      -power VDD -ground VSS
```

If the UPF retention strategy is specified as follows:

```
   set_retention ret -domain PD \
      -save_signal {save posedge} \
      -restore_signal {save negedge}
      ...
```

then the retention cells previously specified shall be used to implement the strategy.

Use **-restore_check**, **-save_check**, **-retention_check**, and **-hold_check** if the cell has additional requirements in retention mode.

In the previous example, if the clock signal needs to maintain low at the save and restore time, use the following command:

```
   define_retention_cell -cells SR1 \
      -clock_pin Clk \
      -restore_function {Ret negedge} \
      -restore_check !Clk -save_check !Clk \
      -power_switchable VDD_SW \
      -power VDD -ground VSS
```

If the clock signal needs to also be low when the primary power is switched off, i.e., in retention mode, use the following command:

```
   define_retention_cell -cells SR1 \
      -clock_pin Clk \
      -restore_function {Ret negedge} \
      -restore_check !Clk -save_check !Clk -retention check !Clk \
      -power_switchable VDD_SW \
      -power VDD -ground VSS
```

If the clock signal does not have to be low or high in at the save or restore, but it needs to maintain the same value before the cell entering retention mode and after the cell exiting retention mode, use the following command:

```
   define_retention_cell -cells SR1 \
      -clock_pin Clk \
      -restore_function {Ret negedge} \
      -hold_check Clk \
      -power_switchable VDD_SW \
      -power VDD -ground VSS
```

```
Liberty Model :

  library(mylib) {

    voltage_map (VDD, 1.0); /* backup power */
    voltage_map (VSW, 1.0); /* primary power */
    voltage_map (VSS, 0.0); /* primary power */

    cell(SR1) {
      retention_cell : RET;
      pg_pin (VSW) {
        voltage_name : VSW;
        pg_type : primary_power;
      }
      pg_pin (VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
      }
      pg_pin (VDD) {
        voltage_name : VDD;
        pg_type : backup_power;
      }
      ff(Q1, QN1 ) {
        clocked_on : " Clk ";
        next_state : " D ";
        clear : "(!Ret * !Q2) + !RESETN";
        preset : "!Ret * Q2";
        clear_preset_var1 : "H";
        clear_preset_var2 : "H";
        power_down_function : "!VSW+VSS";
      }
      latch("Q2", "QN2") {
        enable : " Ret ";
        data_in : " Q1 ";
        power_down_function : "!VDD+VSS";
      }
      clock_condition() {
        clocked_on : "Clk";
        required_condition  :  "!Ret";  /*  cell  in  legal  state  when  Ret  =
logic_low */
        hold_state  :  "N";  /*  retention  data  is  restored  to  either  master  or
slave latch */
      }
      pin(Ret) {
        direction : input;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        retention_pin(save_restore,  "0");  /*  cell  is  in  normal  mode  and  works
as a D-flop when Rer is logic low */
        save_action  :  "H";  /*  When  the  save  happens  at  the  AO  latch  at  the
signal leading edge */
        restore_action  :  "L";  /*  When  the  restore  happens  at  the  output  is  at
the signal trailing edge */
        save_condition : "!Clk"; /* side condition for successful save */
        restore_condition : "!Clk"; /* side condition for successful restore */
        restore_edge_type  :  "trailing";  /*  Edge  when  the  cell  is  starting  to
restore */
      }
      retention_condition() {
        required_condition : "Ret * !Clk";
        power_down_function : "!VSW + VSS";
      }
      clear_condition() {
        input : "!RESETN";
        required_condition  :  "!Ret";  /*  When  clear  asserts,  Ret  must  be  low  to
clear flop output */
      }
      pin(D) {
        direction : input;
        related_power_pin : VSW;
        related_ground_pin : VSS;
      }
      pin(Clk) {
        direction : input;
        related_power_pin : VSW;
        related_ground_pin : VSS;
      }
      pin(RESETN) {
        direction : input;
        related_power_pin : VSW;
        related_ground_pin : VSS;
      }
      pin(Q) {
        direction : output;
        function : "Q1";
        power_down_function : "!VSW+VSS";
        related_power_pin : VSW;
        related_ground_pin : VSS;
      } /* end pin Group */
   } /* end cell group */
} /* end library group */
```

G.6.4 State retention cells with save and restore controls

For a state retention cell with both save and restore controls, the cell saves the current value when the save control pin is activated and the power is on, while the cell restores the saved value when the restore control pin is activated. To model such a cell, use the following options from the **define_retention_cell** command (see 7.7):

define_retention_cell
  -cells *cell_list* [**-cell_type** *string*] **-save_function** {**{pin** <**high** | **low** | **posedge** | **negedge}**}
  -restore_function {**{pin** <**high** | **low** | **posedge** | **negedge}**}
  [**-always_on_pins** *pin_list*] [**-clock_pin** *pin*]
  [**-restore_check** *expression*] [**-save_check** *expression*]
  [**-retention_check** *expression*] [**-hold_check** *pin_list*]
  [**-always_on_components** *component_list*]
  [**-power_switchable** *power_pin*] [**-ground_switchable** *ground_pin*]
  [**-power** *power_pin*] [**-ground** *ground_pin*]

In this case, the cell saves the current value when the save expression is *True* and the power is on. The cell restores the saved value when the restore expression is *True* and the power is on. Figure G.28 shows an example of such a cell.

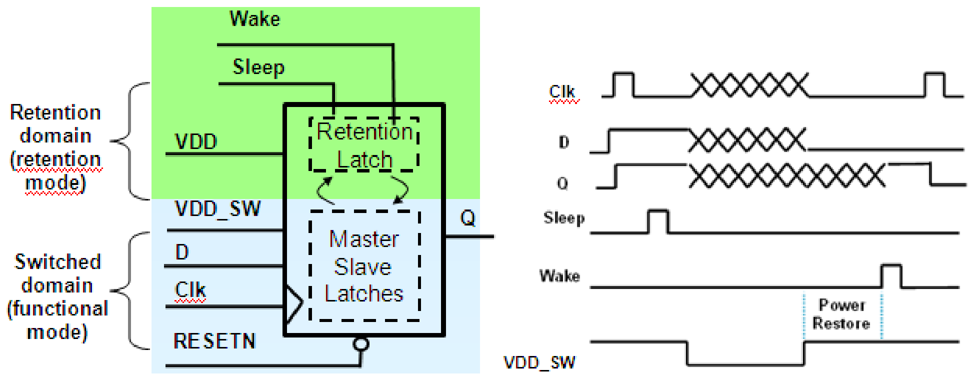


Figure G.28—State retention with save and restore controls

To model the cell shown in Figure G.28, use the following command:

```
define_retention_cell -cells SR2 \
    -clock_pin Clk \
    -restore_function {Wake high} -save_function {Sleep high} \
    -restore_check !Clk -save_check !Clk \
    -power_switchable VDD_SW \
    -power VDD -ground VSS
```

The state is saved when `Sleep` is active and the clock is down, and the state is restored when `Wake` is active and the clock is down.

If the UPF retention strategy is specified as follows:

```
set_retention ret -domain PD \
    -save_signal {save_net high} \
    -restore_signal {restore_net high}
    ...
```

then the retention cells previously specified shall be used to implement the strategy.

Liberty model

```
library(mylib) {

  voltage_map (VSW, 1.0); /* primary power */
  voltage_map (VDD, 1.0); /* backup power */
  voltage_map (VSS, 0.0); /* primary ground */

  cell(SR2) {
    retention_cell : RET;
  pg_pin (VDD) {
    voltage_name : VDD;
    pg_type : backup_power;
  }
  pg_pin (VSS) {
   voltage_name : VSS;
   pg_type : primary_ground;
  }
  pg_pin (VSW) {
   voltage_name : VSW;
   pg_type : primary_power;
  }
  ff(Q1, QN1 ) {
    clocked_on : " Clk ";
    next_state : " D ";
    clear : "(Wake * !Q2) + !RESETN";
    preset : "Wake * Q2";
    clear_preset_var1 : "L";
    clear_preset_var2 : "H";
    power_down_function : "!VSW+VSS";
  }
  latch("Q2", "QN2") {
    enable : " Sleep ";
    data_in : " Q1 ";
    power_down_function : "!VDD+VSS";
  }
  clock_condition() {
    clocked_on : "Clk";
    required_condition  :  "!Sleep";  /*  cell  in  legal  state  when  Sleep  =
logic_low */
    hold_state : "N"; /* retention data is restored to either master or slave
latch */
  }
  pin(Wake) {
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    retention_pin(restore, "0"); /* cell is in normal mode and works as a D-
flop when Wake is logic low */
    restore_action : "H"; /* The restore happens at the signal Leading edge */
    restore_condition : "!Clk"; /* Side condition for successful restore */
    restore_edge_type : "leading"; /* Edge when the cell is starting to restore
*/
  }
  pin(Sleep) {
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    retention_pin(save, "0"); /* cell is in normal mode and works as a D-flop
when Sleep = logic_high */
    save_action : "H";   /* The save happens at the signal leading edge */
    save_condition : "!Clk"; /* Side condition for successful restore */
  }
  retention_condition() {
    required_condition : "Sleep";
    power_down_function : "!VSW + VSS";
  }
  clear_condition() {
    input : "!RESETN";
    required_condition  :  "!Wake";  /*  When  clear  signal  asserts,  Wake  signal
must be low to allow the flop output to clear */
  }
  pin(D) {
    direction : input;
    related_power_pin : VSW;
    related_ground_pin : VSS;
  }
  pin(Clk) {
    direction : input;
    related_power_pin : VSW;
    related_ground_pin : VSS;
  }
  pin(RESETN) {
    direction : input;
    related_power_pin : VSW;
    related_ground_pin : VSS;
  }
  Pin(Q) {
    direction : output;
    function : "Q1";
    related_power_pin : VSW;
    related_ground_pin : VSS;
    power_down_function : "!VSW + VSS";
  } /* end pin group */
 } /* end cell group */
} /* end library group */
```

G.6.5 State retention cells without save or restore control

A master-slave type state retention cell does not have a dedicated save or restore control pin; it has a secondary power or ground pin to provide continuous power supply to the slave latch. Such a cell always saves a copy of the current value before entering the retention mode and the saved value is restored when the primary power is restore.

To model such a cell use the following **define_retention_cell** command options, without **-save_function** or **-restore_function**:

define_retention_cell
-cells *cell_list* [**-cell_type** *string*]
[**-always_on_pins** *pin_list*] [**-clock_pin** *pin*]
[**-restore_check** *expression*] [**-save_check** *expression*]
[**-retention_check** *expression*] [**-hold_check** *pin_list*]
[**-always_on_components** *component_list*]
[**-power_switchable** *power_pin*] [**-ground_switchable** *ground_pin*]
[**-power** *power_pin*] [**-ground** *ground_pin*]

To specify a state retention strategy that targets these types of state retention cells, use the **set_retention** command (see 6.49) and do not use the **-save_signal** or **-restore_signal** options.

The following example models the master-slave retention cell `ms_ret`:

```
define_retention_cell -cells ms_ret \
   -clock_pin CLK \
   -restore_check {!CLK} -save_check {!CLK}
```

The following command shows the state retention strategy that targets cell `ms_ret` for all registers with the power domain `PD1`:

```
set_retention sr1 -domain PD1 \
    -retention_condition {!clock && nreset} \
    -use_retention_as_primary \
    ...
```

Liberty model

```
library(mylib) {

  voltage_map (VSW, 1.0); /* primary power */
  voltage_map (VDD, 1.0); /* backup power */
  voltage_map (VSS, 0.0); /* primary ground */

  cell (ms_ret) {
    retention_cell : 0_pin_clk_low_retention;
    pg_pin (VSW) {
      pg_type : primary_power;
      voltage_name : VSW;
    }
    pg_pin (VDD) {
      pg_type : backup_power;
      voltage_name : VDD;
    }
    pg_pin (VSS) {
      pg_type : primary_ground;
      voltage_name : VSS;
    }
    pin (Q) {
      direction : output;
      function : "IQ2";
      related_ground_pin : VSS;
      related_power_pin : VSW;
      power_down_function : "(CLK * !VSW) + !VDD + VSS";
    } /* end pin group */
    pin (D) {
      direction : input;
      related_ground_pin : VSS;
      related_power_pin  : VSW;
    } /* end pin group */
    pin (CLK) {
      direction : input;
      related_ground_pin : VSS;
      related_power_pin  : VDD;
    } /* end pin group */
    latch (IQ1,IQN1) {
      enable : "CLK";
      data_in : "D";
      power_down_function : "!VSW + VSS";
    }
    latch (IQ2,IQN2) {
      enable : "CLK";
      data_in : "IQ1";
      power_down_function : "!VDD + VSS";
    }
    clock_condition() {
      clocked_on : "CLK";
      hold_state : "L";
    }
  } /* end cell group */
} /* end pin group */
```

Annex H

(informative)

IP power modeling for system-level design

The purpose of this annex is to provide both an informative background into the scope, structure, and expected use of system-level intellectual property (IP) power models.

H.1 Overview of system-level IP power models

Within a system-level design environment we are operating at fairly high levels of design abstraction which enables fast simulation and analysis generally. This analysis performance is attained by abstracting away details of our platform that are not relevant for the types of analysis we wish to perform. In order to extend this analysis to accommodate power, we need to annotate power information onto the simulation. The annotation of power information is performed through the use of system-level IP power models, which are power models of IP components specifically for use in system-level design.

These power models are intended to be used in system-level design although there is nothing to prevent their use in other types of analysis at different levels of abstraction. However, these are highly abstract models of the power behavior of an IP component and so their value outside of system-level design may be somewhat limited. The standard provides support for modeling all types of IP components and provides no limitations on use.

An IP power model exists together with a host model which controls the power model during simulation and activates specific power states within the power model. The host model would typically be a functional model of some kind, but could be anything that ultimately activates and controls the power model.

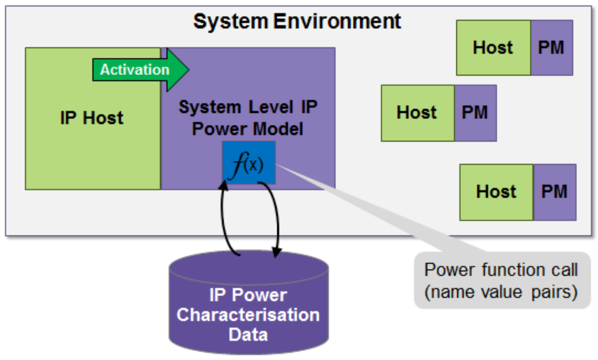

System Environment
Activation
IP Host
System Level IP Power Model
f(x)
Host
PM
Host
PM
Host
PM
IP Power Characterisation Data
Power function call (name value pairs)

Figure H.1—Power model overview

The power model acts entirely as a slave to the host model and cannot directly make any changes to the state of the system.

The power model simply responds to direction from the host and returns data to that same host. Power models can only communicate through the host to which it is attached. Power models cannot communicate directly with each other and so one power model cannot directly affect the power state in another power model.

It is expected that system-level IP power models are developed and distributed by IP teams (whether they be IP vendors or IP implementation teams within larger platform development groups). As such, system-level IP power models are considered to be context independent in nature.

Typically, the abstraction of power information contained within an IP power model is aligned to the functionality of the IP component itself and to the various low power modes supported by the component. The IP power model should make no assumption about the context in which it is to be used. This aligns well with the fundamental principles of IP reuse, portability, and interoperability. The IP power model would typically be instantiated into a context-dependent environment where the binding between objects in the power model to objects in the design can be completed.

H.2 Content of system-level IP power models

H.2.1 Overview

There are three key parts to a system-level IP power model:

- Power state enumeration
- Power state power (or current) consumption data
- Definition of all legal power state transitions

H.2.2 Power state enumeration

Any IP power state which is required during system-level design should be enumerated within the IP power model. The power states defined in the IP power model can represent both:

- Operating modes of the IP
- Supply states within the IP

The various operating modes of an IP component can have a wide range of power consumption figures, and operating modes can be selected without any corresponding manipulation of the supply networks in the design. It is necessary therefore to model this type of behavior within the power model to ensure that power data generated for the platform during simulation accurately reflects the state in which the component is operating. For example, enabling single instruction, multiple data (SIMD) operation within a microprocessor would typically not require a corresponding change in the supply network for the processor but it would result in a significant change to the power consumption of the processor while it is processing SIMD instructions. We may wish therefore to model this SIMD mode as a dedicated power state within the power model of the processor.

Other operation modes of an IP component may be triggered by changes to the supply network itself; power gated shutdown is one example. In this case, we would model the shutdown mode as a power state.

It is recognized that the granularity of power state information may vary depending on the type of system-level analysis being performed and power state hierarchy can be used to help manage this granularity. With the specification of hierarchical power states we can provide the ability to deal with fundamental power states or refinements of those fundamental power states if more accuracy is required.

Power states for an IP power model cannot be defined during run time, but must be explicitly defined within the power model prior to the power model being read.

All legal power state transitions shall be defined within the power model and an attempt to transition between two power states using a transition which is not defined to be a legal transition shall be an error.

H.2.3 Power state power consumption

A power model can be used to calculate either power or current consumption for a power state. IEEE Std 1801 supports both modes of operation and this annex refers to the return of power consumption.

Each power state enumerated within the power model can include specification of the power consumption of that power state. Specification of both static and dynamic power consumption information is required. IEEE Std 1801 requires that these two types of power consumption are calculated and managed separately since power-management schemes address static and dynamic power mitigation separately using different techniques.

Power consumption data can be specified in one of two ways within a power model using the -power_expr option:

- Via the use of floating point data values
- Via the use of power functions

The use of floating point data values (one each for static and dynamic power) is a straightforward approach to annotating power consumption onto a given power state. It is however limited in its ability to support more advanced types of power management like Dynamic Voltage and Frequency Scaling (DVFS), etc. as there is no way to scale these floating point values accurately with voltage or temperature since reference data points are not specified.

Power functions offer considerably greater benefit in terms of accommodating dynamic changes in voltage, frequency, and other run time parameters on which the power consumption of the IP depends and these power functions would typically be provided as a part of the "system-level IP power model" deliverable.

H.2.4 Legal power state transitions

In addition to enumeration of all power states of interest, the power model must also define the legal transitions between these power states. Power state activation will only be successful if it causes a state transition within the component that is legal. All legal power state transitions are defined within the power model using the add_state_transition command.

H.3 Power calculation using power functions

Power functions can be used to calculate the power consumption of a design when in a particular power state using a selection of parameters from the environment (e.g., system simulation). The power functions access raw power characterization data for the IP component in order to compute the power consumption for the component in any given power state using the values of the specific parameters in the design.

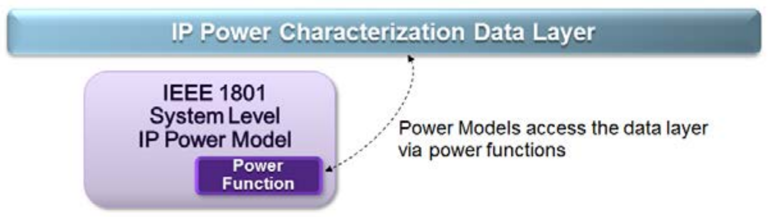

IP Power Characterization Data Layer
IEEE 1801 System Level IP Power Model
Power Function
Power Models access the data layer via power functions

Figure H.2—Power model interface to characterisation data layer

This raw characterization data is referred to as the *data layer* for the IP component and is managed externally to the power model. The IP power model would typically be provided together with an IP data layer. It is important to note that IEEE Std 1801 makes no attempt to standardize the way in which IP is characterized for power or how that power characterization data is stored and represented (the data layer). Both are outside the scope of this standard and would be tightly coupled to the power function definition.

The power consumption of a design is sensitive to many parameters including voltage, frequency, temperature, silicon process, utilization, bandwidth, etc. During system simulation many of these parameters will be modified due to requirements from the scenario (activity), activation of various power-management techniques, platform exploration effects, etc., and so the system architect must ensure that the power consumption values that are provided for each state during simulation accurately reflect the state of the design and the environment in which it is placed.

System-level IP power model parameters can take one of three forms:

— Build time parameters that do not change during run time

— Run time parameters that can change during run time

— Rate-based parameters that can change during run time

Parameters that do not change during simulation are defined using the **-type buildtime** option of the add_parameter command. The silicon process that is being targeted for a platform would be an example of a build time parameter. The power function will need to know which process is being targeted in order to calculate accurate power consumption data. The targeted silicon process will not change during simulation.

Parameters that change value during simulation are defined using the **-type runtime** options of the add_parameter command. These parameters pass values into the power functions that could change during execution of the simulation and would need to trigger re-calculation of the power consumption when they change. Run time parameters would be included in the set of calling options to a power function and this set of options effectively becomes a sensitivity list for the power function: anytime one of the options changes value during simulation, the power function would be called, new power consumption data calculated and returned by the power function to the environment. The supply voltage of a component could be a run time parameter in a DVFS environment.

A special set of run time parameters that model rates within the system are also supported. Rate-based parameters are used to model time-related effects where we wish to capture event counts over time (rates) and use these rates to adjust the power consumption of the object in the current power state. Rate-based parameters would be updated by the EDA tool at specific time intervals (time interval specification is set in the EDA tool and is outside the scope of IEEE Std 1801) and when rate-based parameters form part of the sensitivity list of a power function, a change in their value would trigger an invocation of the power function. Cache miss rate would be an example of a rate-based parameter.

With the power function approach, power consumption of an object (component or power domain) in a given power state is calculated on entry into that power state and then again every time there is a change in a parameter to which the power function for the current power state is sensitive.

It is anticipated that, in many cases, IP vendors and silicon teams may wish to protect the intellectual property rights (IPR) within these power functions. The way in which the power functions are associated with a power state is specified by IEEE Std 1801, but the content and complexity of the power function is not. This way, the standard ensures that interoperability exists between power models in terms of the way power functions are called, but places no restrictions on the complexity of the power functions themselves.

H.4 Power model structure

H.4.1 Power model encapsulation

System-level IP power models are expressed entirely using IEEE Std 1801 language and are encapsulated within the begin_power_model and end_power_model commands. All UPF commands inside this encapsulation are considered to be a part of the power model.

```
begin_power_model <power_model_name>
    [ power model contents ]
end_power_model
```

H.4.2 Power model partitioning

Power functions return static and dynamic power consumption for the primary supply of the component and only data for this one supply is returned for each power state in the component. For more complex components, which are considered to have multiple supplies in a functional mode (not backup or retention), the power model for the component should be partitioned using power domains in such a way that each power domain has a single primary supply for power calculation purposes. Power states within the power model are then defined per power domain.

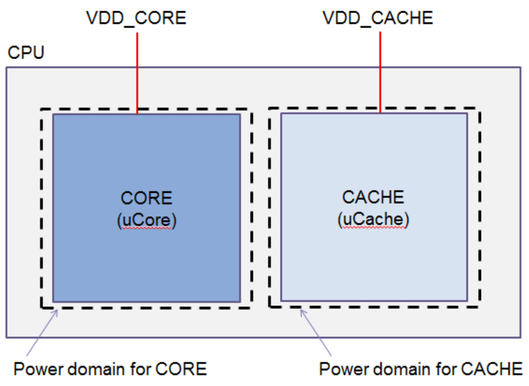


Figure H.3—Power domains within a power model

For example, Figure H.3 shows a CPU component that comprises two top-level components—a core (uCore) and a cache (uCache)—each of which has a separate primary supply, VDD_CORE and VDD_CACHE, respectively. A power model is to be created for the CPU. In order to be able to report power consumption separately for the core and the cache it is necessary to partition the CPU into power domains (using the create_power_domain command). For this example, we create a power domain for the core and a power domain for the cache.

```
create_power_domain PD_CORE -elements uCORE
create_power_domain PD_CACHE -elements uCACHE
```

Power states are now defined relative to these power domains and take advantage of the fact that a power domain has only one primary supply. Power functions that are then defined for power states in the cache, for example, will return power consumption data relative to the VDD_CACHE supply only.

It is recommended that power domains be used even for blocks that do not exhibit this form of complexity as it provides a clean and consistent approach to system-level IP power modeling across IP types.

H.4.3 Parameter definitions

All parameters that are passed to a power function must be defined and initialized within the power model before they are used. It is not possible for a power function in one power model to use parameters that have been defined in a different power model. The scope of a power model parameter is solely within the power model in which it is defined.

```
add_parameter process \
-type buildtime -default 1.0 \
-description "Process Scaling Factor"

add_parameter vddCore \
-type runtime -default 900mV \
-description "Voltage supply for CORE"

add_parameter CacheMiss \
-type rate -default 0.01 \
-description "Cache Miss Rate"
```

H.4.4 Legal state transitions

To close out the power model, the set of legal state transitions must be defined using the add_state_transition command.

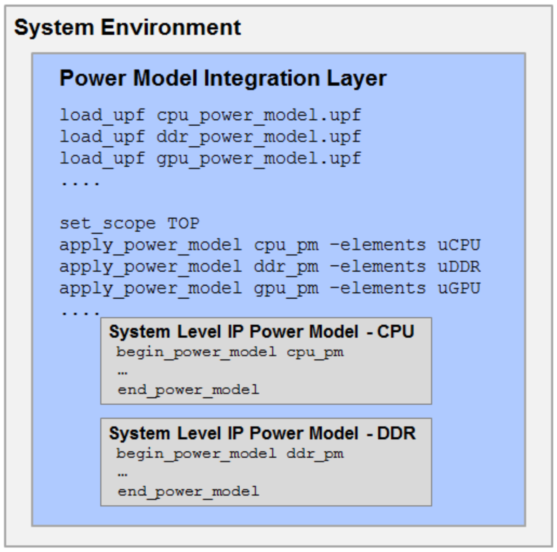
```
add_state_transition -object PD_CORE -from ACTIVE -to WFI –legal
add_state_transition -object PD_CORE -from WFI -to ACTIVE –legal
add_state_transition -object PD_CORE -from WFI -to OFF –legal
add_state_transition -object PD_CORE -from OFF -to WFI –legal
```

Power state activation can only take place when a legal power state transition results.

H.5 Power model instantiation—example approach

It is expected that the system-level IP power model would be a context-independent abstraction of the power behavior of a component and would contain all relevant power-related information for the component required to perform system-level power analysis. As such, the power model itself would not reference any objects (pins, ports, registers, etc.) outside of its own scope.

The process of instantiating a power model into a design environment would ensure binding between the context-independent handles within the power model to context-dependent objects within the environment. This process of placing the power model in context could happen through the use of a power model integration layer.


System Environment
Power Model Integration Layer
System Level IP Power Model - CPU
System Level IP Power Model - DDR

```
load_upf cpu_power_model.upf
load_upf ddr_power_model.upf
load_upf gpu_power_model.upf
....

set_scope TOP
apply_power_model cpu_pm -elements uCPU
apply_power_model ddr_pm -elements uDDR
apply_power_model gpu_pm -elements uGPU
....
```

```
begin_power_model cpu_pm
…
end_power_model
```

```
begin_power_model ddr_pm
…
end_power_model
```

Figure H.4—Power model integration layer

The integration layer can be implemented in UPF with the individual system-level IP power models being instantiated using the apply_power_model command. The apply_power_model command will complete the binding of a power model to one (or many) instances within the design. For example, a single CPU power model couple be applied to many CPU instances within the design via the apply_power_model command, however, since this command is also used to bind parameters within the model to objects within the design, an individual binding is more likely.

To bind parameters within the power model to objects in the environment, the –parameters option of the apply_power_model command is used. This –parameters option is the mechanism by which names of objects within the environment can be passed to handles within the power model. The integration layer approach is an example of how the system-level IP power model could be instantiated within a design.

```
apply_power_model CPU_PM -elements uCPU -parameters {\
                          {temperature uCPU/p_temperature} \
                          {vdd_CORE SR1/p_v_out} \
                          {vdd_CACHE SR2/p_v_out} \
                          {freq_CORE PLL1/p_clk_out} \
                          {freq_CACHE PLL2/p_clk_out} \
                          {L1DACCESS uCPU/L1_data_access} \
                          {IPC uCPU/instruction_per_cycle_uCPU} \
}
```

Annex I

(normative)

Switching Activity Interchange Format

The Switching Activity Interchange Format (SAIF) is designed to assist in the extraction and storing of the switching activity information generated by electronic design automation (EDA) tools.

A SAIF file containing switching activity information can be generated using an HDL simulator and then the switching activity can be back-annotated into the power analysis/optimization tool as shown in Figure I.1. This type of SAIF file is called a *backward SAIF file*.

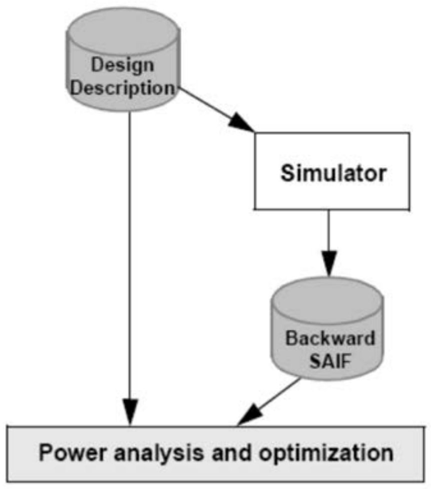


Figure I.1—Backward SAIF file

The power analysis/optimization tool, or some other EDA tool, may issue directives (instructions) to the backward SAIF file generation application on the format of the required SAIF file. These directives can be stored into a SAIF file, called a *forward SAIF file*, as shown in Figure I.2.

This annex provides the syntax and semantics of the backward SAIF file and the following two kinds of forward SAIF files:

a) The library or gate-level forward SAIF file, which contains the directives for generating state-dependent and path-dependent switching activity.

b) The RTL forward SAIF file, which contains the directives for generating switching activity from the simulation of RTL hardware descriptions.

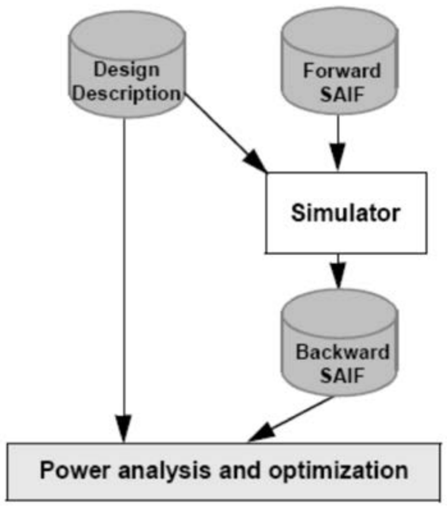


Figure I.2—Forward SAIF file

I.1 Syntactic conventions

The syntax of the SAIF file is described using the Backus-Naur Form (BNF), as follows:

Lowercase words (some containing underscores) are used to denote syntactic categories, e.g.,

backward_instance_info

Boldface words are used to denote the reserved keywords, operators, and punctuation marks that are a required part of the syntax, e.g.,

INSTANCE * ( )

A non-boldface vertical bar (|) separates alternative items, e.g.,

binary_operator ::=
    ***** | **^** | **|**

Note that the last vertical bar is in **boldface** and therefore represents an actual operator rather than a separator between the alternative operators.

Non-boldface square brackets ([]) enclose optional items, e.g.,

date ::=
    (DATE [string]**)**

Non-boldface braces ({}) enclose items that can be repeated 0 or more times, e.g.,

backward_saif_info ::=
    {backward_instance_info}

I.2 Lexical conventions

I.2.1 Overview

SAIF files are a stream of lexical tokens that consist of one or more characters. Except for one-line comments (see the following), the layout of SAIF files is free-format, i.e., spaces and newlines are only syntactically significant as token separators.

The following are types of *lexical tokens* in SAIF files:

— white spaces

— comments

— numbers

— strings

— parenthesis

— operators

— hierarchical separator character

— identifiers

— keywords

The rest of this subclause describes the lexical tokens used in SAIF files and their conventions.

I.2.2 White spaces

White spaces are sequences of spaces, tabs, newlines, and form-feeds. White spaces separate the other lexical tokens.

I.2.3 Comments

The SAIF format allows for both one-line comments and block comments. *One-line comments* start with the character sequence // and end with a newline. *Block comments* start with the character sequence /* and end with the first occurrence of the sequence */. Block comments are not nested.

I.2.4 Numbers

Numbers in SAIF files are either of the following:

— Non-negative decimal integers, which are represented by a sequence of decimal characters, e.g., `12`, `012`, or `1200`.

— Non-negative real numbers, which are non-negative IEEE standard double-precision floating-point number representations, e.g., `1`, `3.4`, `.7`, `0.3`, `2.4e2`, or `5.3e-1`.

I.2.5 Strings

A *string* in SAIF files is a possibly empty sequence of characters enclosed by double-quotes characters ("") and contained on a single line, e.g.,`"SAIF version 2.0"` or `""`.

I.2.6 Parenthesis

Most of the constructs in SAIF files are enclosed between the left-parenthesis character (() and the right-parenthesis character ()).

I.2.7 Operators

An *operator* in SAIF files is one of the following characters: !, *, ^, and |. Operators are used in conditional expressions.

I.2.8 Hierarchical separator character

The *hierarchical separator* is a special character used in composing hierarchical port/pin/net/instance names from simple identifiers. The hierarchical separator character is defined in the header of SAIF files and can be either the / character or the . character.

I.2.9 Identifiers

A SAIF *identifier* is a non-empty sequence of alphanumeric characters, the underscore character (_) and escaped characters, followed by an optional decimal number enclosed in brackets ([]). *Escaped identifiers* consist of the \ character followed by a non-white space character. A SAIF identifier cannot start with a decimal digit (.) character and cannot contain the hierarchical separator character, unless it is escaped. The \ character used in an escaped character is not part of the identifier, so `abc` and `a\b\c` represent the same identifier. SAIF identifiers are case-sensitive, `abc` and `ABC` represent two different identifiers.

Examples

`clk`, `clk_net`, `clk[4]`, `clk\#4`, `clk\(4\)`, `\1clk`, or `mod\/net`

Where the hierarchical separator character is presumed to be `/`.

I.2.10 Keywords

A SAIF *keyword* is a special sequence of alphanumeric characters. SAIF keywords can be used as identifiers; to avoid possible ambiguity, escape the first character of identifiers that can be mistaken for keywords. SAIF keywords are case-sensitive. Table I.1 shows the set of SAIF keywords.

Table I.1—SAIF keywords

COND	LEAKAGE	TB
COND_DEFAULT	LIBRARY	TC
DATE	MODULE	TG
DESIGN	NET	TIMESCALE
DIRECTION	PORT	TX
DIVIDER	PROGRAM_NAME	TZ
DURATION	PROGRAM_VERSION	VENDOR
FALL	RISE	VIRTUAL_INSTANCE
IG	RISE_FALL	fs
IK	SAIFILE	ms
INSTANCE	SAIFVERSION	ns
IOPATH	T0	ps
IOPATH_DEFAULT	T1	s
		us

I.2.11 Syntactic categories for token types

The syntax of the SAIF files described in this document use the syntactic categories shown in Table I.2 for token types.

Table I.2—Token type categories

Syntactic category	Token type
dnumber	Non-negative integer numbers
rnumber	Non-negative real numbers
string	Strings
hchar	Possible hierarchical separator characters
identifier	Simple (non-hierarchical) identifiers
hierarchical_identifier	Hierarchical identifiers

I.3 Backward SAIF file

I.3.1 Overview

This subclause describes the format of the *backward SAIF file*, which contains hierarchical instance-specific switching activity information.

I.3.2 SAIF file

The backward SAIF file consists of a left-parenthesis (( ), the **SAIFILE** keyword, the backward SAIF header, the backward SAIF info, and a right-parenthesis ( )), as shown in Syntax 1.

```
backward_saif_file ::=
    (SAIFILE backward_saif_header backward_saif_info)
```

Syntax 1—backward_saif_file

I.3.3 Header

I.3.3.1 Overview

Syntax 2 defines the backward SAIF file header.

```
backward_saif_header ::=
    backward_saif_version
    direction
    design_name
    date
    vendor
    program_name
    program_version
    hierarchy_divider
    time_scale
    duration
```

Syntax 2—backward_saif_header

Each backward SAIF header construct is described in the following subclauses.

I.3.3.2 backward_saif_version

Syntax 3 defines the `backward_saif_version`.

```
backward_saif_version ::=
    (SAIFVERSION string)
```

Syntax 3—backward_saif_version

The *string* in this construct represents the version number of the SAIF file, i.e., **2.0**.

I.3.3.3 direction

Syntax 4 defines the `direction`.

```
direction ::=
    (DIRECTION string)
```

Syntax 4—direction

The *string* in this construct represents the type of the SAIF file, i.e., **backward**.

I.3.3.4 design_name

Syntax 5 defines the `design_name`.

```
design_name ::=
    (DESIGN [string])
```

Syntax 5—design_name

The optional *string* in this construct represents the design for which the switching activity in the SAIF file has been generated.

I.3.3.5 date

Syntax 6 defines the `date`.

```
date ::=
    (DATE [string])
```

Syntax 6—date

The optional *string* in this construct represents the date the SAIF file was generated.

I.3.3.6 vendor

Syntax 7 defines the `vendor`.

```
vendor ::=
    (VENDOR [string])
```

Syntax 7—vendor

The optional *string* in this construct represents the name of the vendor whose application was used to generate the SAIF file.

I.3.3.7 program_name

Syntax 8 defines the `program_name`.

```
program_name ::=
    (PROGRAM_NAME [string])
```

Syntax 8—program_name

The optional *string* in this construct represents the name of the application used to generate the SAIF file.

I.3.3.8 program_version

Syntax 9 defines the `program_version`.

```
program_version ::=
    (PROGRAM_VERSION [string])
```

Syntax 9—program_version

The optional *string* in this construct represents the version number of the application used to generate the SAIF file.

I.3.3.9 hierarchy_divider

Syntax 10 defines the `hierarchy_divider`.

```
hierarchy_divider ::=
    (DIVIDER [hchar])
```

Syntax 10—hierarchy_divider

The optional *hchar* in this construct represents the hierarchical separator character used in hierarchical identifiers. Only the / and **.** characters shall be specified as the hierarchical separator character; the default is the **.** character.

I.3.3.10 time_scale

Syntax 11 defines the `time_scale`.

```
time_scale ::=
    (TIMESCALE [dnumber timeunit])
timeunit ::=
    s | ms | us | ns | ps | fs
```

Syntax 11—time_scale

This construct specifies the units used for all time values in the SAIF file. The *dnumber* shall be **1**, **10**, or **100**; it represents the scaling factor of the time values. For example, if the `time_scale` of a SAIF file is

```
(TIMESCALE 100 us)
```

then all the time values in the SAIF file are specified in hundreds of microseconds. If the decimal number and time unit are not specified, the default time scale is `1 ns`.

I.3.3.11 duration

Syntax 12 defines the `duration`.

```
duration ::=
    (DURATION rnumber)
```

Syntax 12—duration

This construct specifies the total time duration applied to the switching activity in the SAIF file.

I.3.3.12 Example

This is an example of a valid backward SAIF file header.

```
(SAIFVERSION "2.0")
(DIRECTION "backward")
(DESIGN "alu")
(DATE "Fri Jan 18 10:30:00 PDT 2002")
(VENDOR "SAIF'R'US Corp.")
(PROGRAM_NAME "saifgenerator")
(PROGRAM_VERSION "1.0")
(DIVIDER /)
(TIMESCALE 1 ns)
(DURATION 5000)
```

I.3.4 Simple timing attributes

This construct specifies the total duration (in time values) that some particular design net/port/pin (specified elsewhere) has some particular value. Syntax 13 defines this construct.

```
simple_timing_attribute ::=
        (T0 rnumber)
      | (T1 rnumber)
      | (TX rnumber)
      | (TZ rnumber)
      | (TB rnumber)
```

Syntax 13—simple_timing_attribute

The different types of simple timing attributes are as follows:

- **T0** is the total time the design object has the value `0`.
- **T1** is the total time the design object has the value `1`.
- **TX** is the total time the design object has an unknown value.
- **TZ** is the total time the design object is in a floating bus state. A *floating bus state* is the state when all drivers on a particular bus are disabled and the bus has a floating logic value.
- **TB** is the total time the design object is in a bus contention state. A *bus contention state* is the state when two or more drivers simultaneously drive a bus to different logic levels.

Example

If the time scale is 100 µs, then the following three simple timing attribute constructs:

```
(T0 100)
(T1 92.5)
(TX 7.5)
```

specify a particular design object has the value `0` for a total `10 000 µs`, the value `1` for a total of `9250 µs`, an unknown value for a total of `750 µs`, and it never reaches the floating bus and bus contention states.

I.3.5 Simple toggle attributes

I.3.5.1 Overview

This attribute construct specifies the number on a particular type of toggle registered on a particular design net/port/ pin (specified elsewhere). Syntax 14 defines this construct.

```
simple_toggle_attribute ::=
        (TC rnumber)
      | (TG rnumber)
      | (IG rnumber)
      | (IK rnumber)
```

Syntax 14—simple_toggle_attribute

The different types of c are as follows:

— **TC** is the number of `0` to `1` plus the number of `1` to `0` transitions. This is usually referred to as the *toggle count*.

— **TG** is the number of transport glitch edges (see I.3.5.2).

— **IG** is the number of inertial glitch edges (see I.3.5.3).

— **IK** is the inertial glitch de-rating factor. To estimate this factor, see I.3.5.4.

Example

The following simple toggle attributes:

```
(TC 200)
(IG 6)
```

specify a total of 200 transitions between the `0` and `1` logic states, and a total of six inertial glitch edges are registered on some particular design object(s).

I.3.5.2 Transport glitch

*Transport glitche*s are extra transitions at the output of the gate before the output signal reaches its steady state and, unlike inertial glitches (see I.3.5.3), cannot be canceled by an inertial delay algorithm. A transport glitch consumes the same amount of power as a normal toggle transition and is an ideal candidate for power minimization during the optimization process. Transport glitches at the output of the gate have a pulse width longer than the gate delay and do not contribute to the functional behavior of the circuit.

In general, the number of transport glitch transitions occurring in the circuit is the difference between the total number of toggle transitions obtained from a full-timing simulation and that from a cycle-based simulation, assuming all inertial glitches (see I.3.5.3) have been filtered out by the timing simulator, i.e., the total number of toggles obtained from the timing simulator does not include inertial glitches. Figure I.3. shows a possible way to have transport glitches in the circuit. Although steady-state analysis of the circuit indicates that node N, the output of the XOR gate, should always remain at logic 1 regardless of the primary input, the additional timing delay due to the inverter causes a glitch at N whenever the input changes its state.

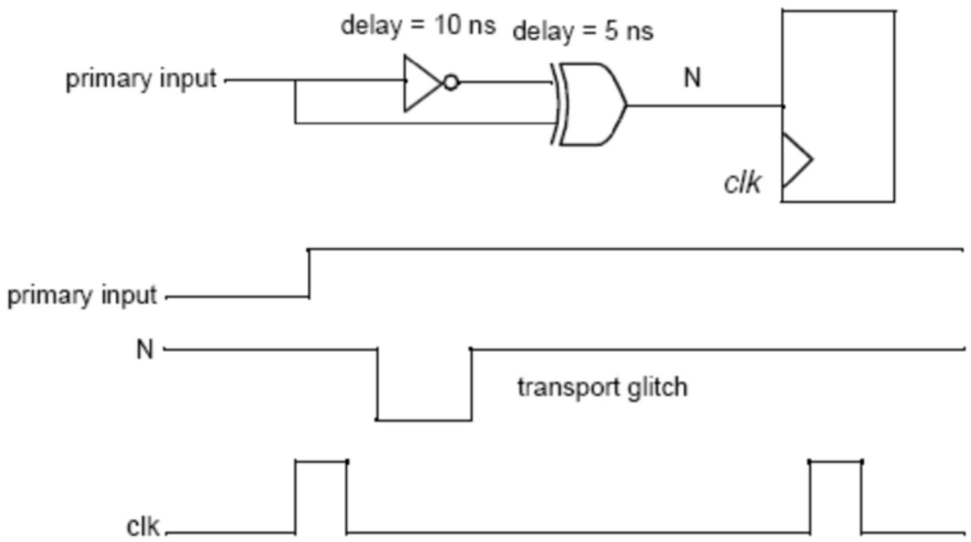


Figure I.3—Transport glitch

I.3.5.3 Inertial glitch

Inertial glitches are signal transitions occurring at the output of the gate, which can be filtered out if an inertial delay algorithm is applied. A simple example (see Figure I.4) best explains inertial glitches.

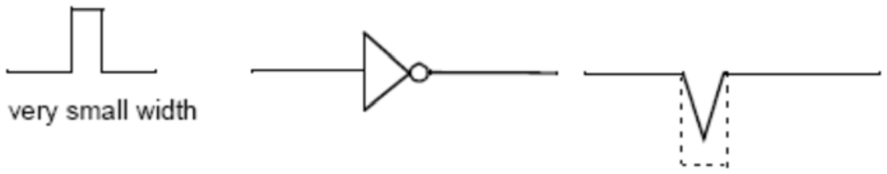


Figure I.4—Inertial glitch

A VHDL description for this inverter looks something like:

`OUT <= not IN after 5 ns` (inertial delay is implicitly presumed)

If the input pulse has a width less than `5 ns`, the inertial delay algorithm shall cancel the signal transitions at the output of the inverter. However, some power is still consumed due to the two partial transitions at the output. Therefore, it is necessary to report these two inertial glitch transitions in a SAIF file.

NOTE—SAIF counts the number of glitches by signal edges, not signal pulses.

I.3.5.4 De-rating factor for inertial glitch

In I.3.5, glitching activities are categorized into two types, transport glitches and inertial glitches, and the number of glitch transitions are reported in the SAIF file. Transport glitches consume the same amount of power as normal toggles, so power consumption can be accurately calculated based on the number of transitions. For inertial glitches, however, the number of transitions is not enough to accurately estimate the inertial glitching power dissipation.

To improve the accuracy for inertial glitching power estimation, it is recommended that a simulator provide a de-rating factor for each node in the circuit that has inertial glitches. Described as follows, this de-rating factor can be used to scale the inertial glitch count to an effective count of normal toggle transition. Power analysis tools can use the adjusted inertial glitch count to improve estimation accuracy.

Assume a gate has a total number of k delays, with a delay value of T_i ($i = 1...k$) for each delay.

Define N_i ($i = 1...k$) as the total number of inertial glitch pulses due to the delay T_i, and δ_{ij} as the timing difference of the input events that cause glitch j ($j = 1...N_i$) due to the delay T_i.

Define N_e as the total number of inertial glitch edges of the gate. It is easy to see that N_i and N_e satisfy Equation (I.1).

$$\sum_{i=1}^{k} N_i = \frac{N_\mathrm{e}}{2} \tag{I.1}$$

NOTE—The total number of the glitch pulses is half of the total number of the glitch edges.

With the parameters previously defined, a de-rating factor can be defined as shown in Equation (I.2).

$$K = 2 \times \frac{\sum_{i}^{k} \sum_{j}^{N_i} \frac{\delta_{ij}}{T_i}}{N_\mathrm{e}} \tag{I.2}$$

Here is an example of how to use the de-rating factor. Consider again the example of the inverter shown in Figure I.5.

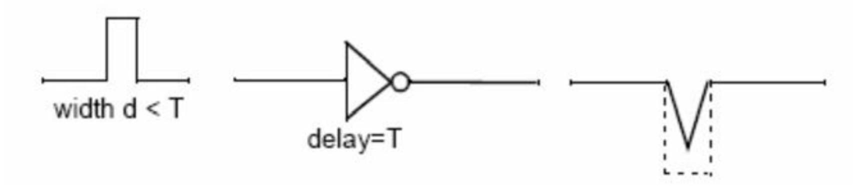


Figure I.5—Inverter

The power consumption at the output can be approximated as shown in Equation (I.3).

$$P = \frac{\delta}{T} \times 2 \times P0 \qquad 0 \le \delta \le T \tag{I.3}$$

where

$P0$	is the power consumption of the gate during one normal full-level transition
δ	is the timing difference of the two input events that cause the glitch
T	is the delay of the inverter

Equation (E.3) indicates that the inertial glitching power dissipation can be roughly modeled by the timing difference of the input events that causes the glitch and the delay of the gate beyond which there is no inertial glitch.

Accordingly, for a node with a total of N_i number of inertial glitch pulses due to the delay T_i (i = 1...k), the total power consumption can be estimated as shown in Equation (I.4).

$$P = \sum_{i=1}^{k} \sum_{j=1}^{N_i} \frac{\delta_{ij}}{T_i} \times 2 \times P0 \tag{I.4}$$

Rearranging Equation (I.2) and substituting Equation (I.4), the power consumption can be simplified as shown in Equation (I.5).

$$P = K \times N_e \times P0 \tag{I.5}$$

This suggests that the inertial glitching power can be calculated by converting the number of glitching transitions into the number of normal transitions by applying a de-rating factor.

I.3.6 State-dependent timing attributes

State-dependent timing attributes specify the time duration when a cell is in particular states. The *state* of a cell is defined as the logic value of its pins. Syntax 15 defines this construct.

```
state_dep_timing_attributes ::=
    (state_dep_timing_item {state_dep_timing_item}
    [COND_DEFAULT sd_simple_timing_attributes])
state_dep_timing_item ::=
    COND cond_expr sd_simple_timing_attributes
cond_expr ::=
      port_name
    | unary_operator cond_expr
    | cond_expr binary_operator cond_expr
    | (cond_expr)
port_name ::=
    identifier
unary_operator ::=
    !
binary_operator ::=
    * | ^ | |
sd_simple_timing_attributes ::=
    {sd_simple_timing_attribute}
sd_simple_timing_attribute ::=
      (T1 rnumber)
    | (T0 rnumber)
```

Syntax 15—state_dep_timing_attributes

Here `cond_expr` represents conditional expressions on pin names; `sd_simple_timing_attribute` can only contain one of the following:

— **T1** is the total time duration in which the cell is in any of its associated states.

— **T0** is the total time duration in which the cell is not in any of its associated states.

A *conditional expression* specifies the set of states for which the condition holds. For example, given a cell with three inputs, A, B, and C, and one output Y, the conditional expression

```
A | B
```

represents all the cell states when the input pin A is 1 or the input B is 1, while C and Y can have any value.

The precedence of the operators in conditional expressions is shown in the following sequence: **!** (logical not), ***** (logical and), **^** (logical exclusive or), and **|** (logical or), where **!** has the highest precedence.

A state-dependent timing attribute construct determines a priority-encoded specification of the timing attributes attrs1, ..., attrs_default:

```
(COND expr1 attrs1
COND expr2 attrs2
...
COND exprn attrsn
COND_DEFAULT attrs_default)
```

In other words, the attributes attrs1 apply for the set of states for which the condition expr1 holds, while the attributes attrs2 apply for the set of states where the condition expr2 holds and expr1 does not hold, etc. The attributes attrs_default apply for all the states where none of the conditional expressions hold.

Example

The state-dependent timing attributes of the cell given in Figure I.6 during the time duration given in the wave diagram in Figure I.7 can be specified as follows:

```
(COND (A * B * Y) (T1 1) (T0 8)
COND (!A * B * Y) (T1 1) (T0 8)
COND (A * !(B * C)) (T1 2) (T0 7)
COND B (T1 1) (T0 8)
COND C (T1 1) (T0 8)
COND_DEFAULT (T1 3) (T0 6))
```

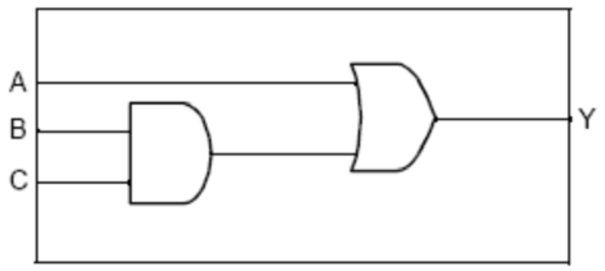


Figure I.6—A cell and its internal behavior

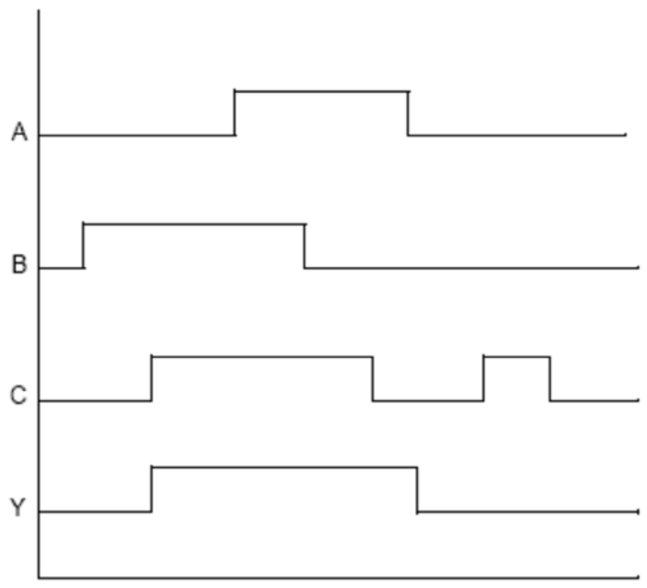


Figure I.7—A wave diagram

I.3.7 State-dependent toggle attributes

The toggle attributes on cell pins can be *state dependent*, i.e., the attributes are relevant only to particular cell states. Syntax 16 defines this construct.

```
state_dep_toggle_attributes ::=
      (state_dep_toggle_item {state_dep_toggle_item}
      [state_dep_default_toggle_item])
state_dep_toggle_item ::=
      COND cond_expr [(edge_type)] simple_toggle_attribute
state_dep_default_toggle_item ::=
        COND_DEFAULT simple_toggle_attribute
      | COND_DEFAULT (edge_type) simple_toggle_attribute
        [COND_DEFAULT (edge_type) simple_toggle_attribute]
edge_type ::=
      RISE | FALL
```

Syntax 16—state_dep_toggle_attributes

Similar to state-dependent timing attributes, the state-dependent toggle attributes construct represents a priority-encoded attribute specification. The optional `edge_type` is used to further differentiate the toggle count between `0` to `1` (**RISE**) and `1` to `0` (**FALL**) transitions.

The state-dependent toggle attributes construct can end with an optional **COND_DEFAULT** specification that has no edge restrictions. Otherwise, it can end with up to two **COND_DEFAULT** specifications having different edge restrictions.

Example

The following state-dependent toggle attributes construct specifies a total toggle count of 50.

```
 (COND A (RISE) (TC 20)
COND A (FALL) (TC 15)
COND B (RISE) (TC 5)
COND B (FALL) (TC 10))
```

Of the 25 rise transitions, 20 occur when pin A has a value of 1, and 5 occur when pin A has a value of 0 and B is 1. Of the 25 fall transitions, 15 occur when the pin A is 1, and 10 occur when the pin A is 0 and B is 1.

The state associated with an input pin transition is the cell state just before the time of the transition. For example, in the wave diagram given in Figure I.8, the state associated with the rise transition on input pin A at time 10 is represented by the expression A * !B * !Y.

The state associated with an output pin transition is the cell state just before the time of the input pin transition, causing the output pin transition. For example, in the wave diagram given in Figure I.8, the rise transition on the output pin Y at time 13 is caused by the rise transition on the input pin B at time 10. The state associated with the rise transition on Y is the cell state just before time 10 (not time 13). This state is represented by the expression !A * B * !Y.

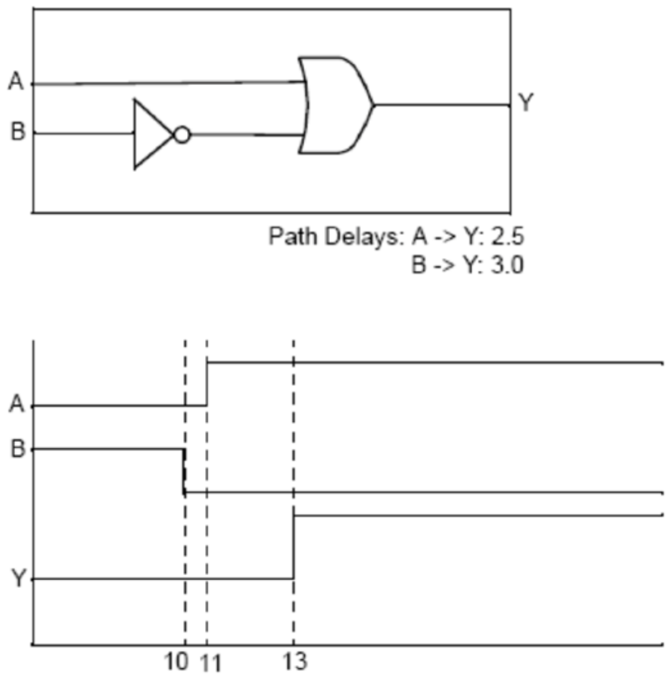


Figure I.8—A cell and its wave diagram

I.3.8 Path-dependent toggle attributes

The toggle attributes on output cell pins can be *path dependent*, i.e., the attributes are relevant only to particular input pins causing the output toggles. Syntax 17 defines this construct.

```
path_dep_toggle_attributes ::=
    (path_dep_toggle_item {path_dep_toggle_item}
    [IOPATH_DEFAULT simple_toggle_attribute])
path_dep_toggle_item ::=
    IOPATH port_name {port_name} simple_toggle_attribute
```

Syntax 17—path_dep_toggle_attributes

Given the path-dependent toggle attributes construct below, the attribute `attrs1` represents toggles caused by the input pins in `pins1`, the attribute `attrs2` represents toggles caused by the input pins in `pins2`, etc.

```
(IOPATH pins1 attrs1
IOPATH pins2 attrs2
...
IOPATH pinsn attrsn
IOPATH_DEFAULT attrs_default)
```

The pin lists `pins1`, `...`, `pinsn` are mutually exclusive. The attribute `attrs_default` represents toggles caused by the cell input pins not present in `pins1`, `...`, `pinsn`. The pin lists `pins1`, `...`, `pinsn` are also called the *path conditions* or *related pins*.

Example

The following path-dependent toggle attributes construct specifies a total of 35 toggle edges on a cell output port, of which 10 are caused by transitions on the input port `A`, 20 are caused by transitions on the input port `B`, and 5 are caused either by a transition on the input port `C` or `D`.

```
(IOPATH A (TC 10)
IOPATH B (TC 20)
IOPATH C D (TC 5))
```

I.3.9 State- and path-dependent toggle attributes

The toggle attributes on output cell pins can be both state dependent and path dependent. The syntax of such toggle attributes is that of simple toggle attributes and path-dependent toggle attributes nested inside a state-dependent toggle attributes construct, as shown in Syntax 18.

```
sdpd_toggle_attributes ::=
    (sdpd_toggle_item {sdpd_toggle_item}
    [sdpd_default_toggle_item])
sdpd_toggle_item ::=
    COND cond_expr [(edge_type)] potential_pd_toggle_attributes
potential_pd_toggle_attributes ::=
      path_dep_toggle_attributes
    | simple_toggle_attribute
sdpd_default_toggle_item ::=
      COND_DEFAULT potential_pd_toggle_attributes
    | COND_DEFAULT (edge_type) potential_pd_toggle_attributes
      [COND_DEFAULT (edge_type) potential_pd_toggle_attributes]
```

Syntax 18—sdpd_toggle_attributes

Similar to state-dependent toggle attributes and path-dependent toggle attributes, the SDPD toggle attributes construct represents a priority-encoded attribute specification.

Example

This is an example of an SDPD toggle attributes construct:

```
(COND A (RISE) (IOPATH B (TC 1))
COND A (FALL) (IOPATH B (TC 2))
COND B (RISE) (IOPATH A (TC 1))
COND B (FALL) (IOPATH A (TC 0))
COND_DEFAULT (RISE) (IOPATH A (TC 1)
IOPATH B (TC 0))
COND_DEFAULT (FALL) (IOPATH A (TC 0)
IOPATH B (TC 1)))
```

I.3.10 Net, port, and leakage-switching specifications

I.3.10.1 Overview

The constructs for net, port, and leakage-switching specification associate switching activity (given in terms of timing and toggle attributes) to individual design nets, ports, and cells.

I.3.10.2 Net-switching specifications

The net-switching specification construct associates switching activity to individual nets. Syntax 19 defines the `backward_net_spec`.

```
backward_net_spec ::=
    (NET backward_net_info {backward_net_info})
backward_net_info ::=
    (net_name net_switching_attributes)
net_name ::=
    identifier
net_switching_attributes ::=
    {net_switching_attribute}
net_switching_attribute ::=
     simple_timing_attribute
    | simple_toggle_attribute
```

Syntax 19—backward_net_spec

The switching attributes that can be associated to nets are simple timing attributes and simple toggle attributes.

Example

This is an example of a net-switching specification assigning switching activity to the nets `clk`, `rst`, `in1`, `in2`, and `out`:

```
(NET
(clk (T0 100) (T1 100) (TC 50))
(rst (T0 180) (T1 20) (TC 2))
(in1 (T0 60) (T1 140) (TC 22))
(in2 (T0 80) (T1 120) (TC 12))
(out (T0 120) (T1 60) (TX 20) (TC 10))
)
```

I.3.10.3 Port-switching specifications

The port-switching specification construct associates switching activity to individual design ports and cell pins. Syntax 20 defines the `backward_port_spec`.

```
backward_port_spec ::=
    (PORT backward_port_info {backward_port_info})
backward_port_info ::=
    (port_name port_switching_attributes)
port_name ::=
    identifier
port_switching_attributes ::=
    {port_switching_attribute}
port_switching_attribute ::=
     simple_timing_attribute
    | simple_toggle_attribute
    | state_dep_toggle_attributes
    | path_dep_toggle_attributes
    | sdpd_toggle_attributes
```

Syntax 20—backward_port_spec

The toggle attributes that can be associated to input cell pins can be simple or state dependent. The toggle attributes that can be associated to output cell pins can be simple, state dependent, path dependent, or both state and path dependent. The toggle attributes that can be associated to design ports have to be simple. The timing attributes that can be associated to design ports and cell pins have to be simple.

Example

This is an example of the port-switching specification construct applied to an `AND` gate:

```
(PORT
(A (T0 8) (T1 7)
(COND B (RISE) (TC 1)
COND B (FALL) (TC 2)
COND_DEFAULT (TC 1)))
(B (T0 9) (T1 6)
(COND A (RISE) (TC 2)
COND A (FALL) (TC 1)
COND_DEFAULT (TC 3)))
(Y (T0 10) (T1 5)
(COND A (RISE) (IOPATH B) (TC 2)
COND A (FALL) (IOPATH B) (TC 1)
COND B (RISE) (IOPATH A) (TC 1)
COND B (FALL) (IOPATH A) (TC 2)
COND_DEFAULT (TC 0)))
)
```

I.3.10.4 Leakage-switching specifications

The leakage-switching specification construct specifies the duration that a particular cell spends in particular states. This construct is a list of state-dependent timing attributes, as shown in Syntax 21.

```
backward_leakage_spec ::=
    (LEAKAGE state_dep_timing_attributes {state_dep_timing_attributes})
```

Syntax 21—backward_leakage_spec

Example

This is an example of a leakage-switching specification:

```
(LEAKAGE
(COND (A * B) (T1 5) (T0 10))
COND (A | B) (T1 6) (T0 9))
(COND_DEFAULT (T1 4) (T0 11)))
)
```

I.3.11 Backward SAIF info and instance data

Design-switching activity is organized hierarchically in the backward SAIF info construct (that follows the SAIF header in a backward SAIF file). The *backward SAIF info* is a list of backward instance info constructs, as shown in Syntax 22.

```
backward_saif_info ::=
    {backward_instance_info}
backward_instance_info ::=
     (INSTANCE [string] path {backward_instance_spec} {backward_instance_info})
    | (VIRTUAL_INSTANCE string path backward_port_spec)
backward_instance_spec ::=
     backward_net_spec
    | backward_port_spec
    | backward_leakage_spec
```

Syntax 22—backward_saif_info

`backward_instance_info` contains the switching activity of a particular cell or design instance. The optional *string* following the **INSTANCE** keyword is the cell/design name that is instantiated, and *path* is the hierarchical name of the actual instance. This is followed by a possibly empty list of instance switching specifications, which are the net, port, and leakage-switching specifications described in I.3.10. For design instances, the instance info can recursively contain the switching activity of its sub-design and library cell instances.

`backward_instance_info` can also be used to specify the switching activity of cell instances where the port names of the instance are not known, e.g., in design flows where switching activity generated by RTL simulation is annotated to the synthesized gate-level netlist of the RTL design.

In this case, the *string* following the **VIRTUAL_INSTANCE** keyword represents the type of cell instance; it needs to be recognized by the application reading the backward SAIF file. The *path* represents the name of the instance, and `backward_port_spec` assigns switching activity to logical port names. The application reading the SAIF file needs to map the logical port names to the actual cell instance port names.

Example

For example, the following virtual instance construct gives the switching activity of the positive output pin of a sequential element:

```
(VIRTUAL_INSTANCE "sequential" A_reg
(PORT
(Q (T0 220) (T1 370) (TC 122))
)
)
```

The actual name of the output pin depends on the library cell that is used to implement the sequential cell, i.e., it can have a different name than `Q`.

I.4 Library forward SAIF file

I.4.1 Overview

The *library forward SAIF file* contains the SDPD directives needed by simulators and other applications generating backward SAIF files that contain state-dependent and path-dependent switching activity. The SDPD directives can be generated from cell libraries with SDPD power characterization by using the appropriate tools.

For a description of state and path dependency, see I.3.

I.4.2 The SAIF file

The library forward SAIF file consists of a left-parenthesis (`(`), the **SAIFILE** keyword, the library forward SAIF header, the library forward SAIF info, and a finishing right-parenthesis (`)`), as shown in Syntax 23.

```
lforward_saif_file ::=
    (SAIFILE lforward_saif_header lforward_saif_info)
```

Syntax 23—lforward_saif_file

I.4.3 Header

Syntax 24 defines the *library forward SAIF file header*.

```
lforward_saif_header ::=
    lforward_saif_version
    direction
    design_name
    date
    vendor
    program_name
    program_version
    hierarchy_divider
```

Syntax 24—forward_saif_header

Each library forward SAIF header construct is described in the following subclauses.

I.4.3.1 lforward_saif_version

Syntax 25 defines the `lforward_saif_version`.

```
lforward_saif_version ::=
    (SAIFVERSION string [string])
```

Syntax 25—lforward_saif_version

The first *string* in this construct represents the version number of the SAIF file, i.e., **2.0**.

The second *string* is optional and is either the string **"lib"** or **"LIB"**; this is used to specify that the SAIF file is a library forward SAIF file.

I.4.3.2 direction

Syntax 26 defines the `direction`.

```
direction ::=
    (DIRECTION string)
```

Syntax 26—direction

The *string* in this construct represents the type of the SAIF file, i.e., **forward**.

I.4.3.3 design_name

Syntax 27 defines the `design_name`.

```
design_name ::=
    (DESIGN [string])
```

Syntax 27—design_name

The optional *string* in this construct represents the design for which the forward SAIF file has been generated.

I.4.3.4 date

Syntax 28 defines the `date`.

```
date ::=
    (DATE [string])
```

Syntax 28—date

The optional *string* in this construct represents the date the SAIF file was generated.

I.4.3.5 vendor

Syntax 29 defines the `vendor`.

```
vendor ::=
    (VENDOR [string])
```

Syntax 29—vendor

The optional *string* in this construct represents the name of the vendor whose application was used to generate the SAIF file.

I.4.3.6 program_name

Syntax 30 defines the `program_name`.

```
program_name ::=
    (PROGRAM_NAME [string])
```

Syntax 30—program_name

The optional *string* in this construct represents the name of the application used to generate the SAIF file.

I.4.3.7 program_version

Syntax 31 defines the `program_version`.

```
program_version ::=
    (PROGRAM_VERSION [string])
```

Syntax 31—program_version

The optional *string* in this construct represents the version number of the application used to generate the SAIF file.

I.4.3.8 hierarchy_divider

Syntax 32 defines the `hierarchy_divider`.

```
hierarchy_divider ::=
    (DIVIDER [hchar])
```

Syntax 32—hierarchy_divider

The optional *hchar* in this construct represents the hierarchical separator character used in hierarchical identifiers. Only the / and **.** characters shall be specified as the hierarchical separator character; the default is the **.** character.

Example

This is an example of a valid library forward SAIF file header.

```
(SAIFVERSION "2.0" "lib")
(DIRECTION "forward")
(DESIGN)
(DATE "Fri Jan 18 10:00:00 PDT 2002")
(VENDOR "SAIF'R'US Corp.")
(PROGRAM_NAME "libsaifgenerator")
(PROGRAM_VERSION "1.0")
(DIVIDER /)
```

I.4.4 State-dependent timing directive

State-dependent timing directives instruct the backward SAIF generator on the state conditions required in state-dependent timing attributes. Syntax 33 defines the `state_dep_timing_directive`.

```
state_dep_timing_directive ::=
    (state_dep_timing_directive_item
    {state_dep_timing_directive_item}
    [COND_DEFAULT])
state_dep_timing_directive_item ::=
    COND cond_expr
```

Syntax 33—state_dep_timing_directive

A state-dependent timing directive is a list of directive items. The state-dependent timing attributes generated using such a timing directive contain switching activity assigned to a number of the states given in the directive. The order of any states in the timing attribute shall be the same as that in the timing directive.

Example

This is an example of a state-dependent timing directive.

```
 (COND (A * B * C)
COND (!A * B * C)
COND (A * !(B * C))
COND B
COND C
COND_DEFAULT)
```

I.4.5 State-dependent toggle directive

State-dependent toggle directives instruct the backward SAIF generator on the state and rise/fall conditions required in state-dependent toggle attributes. Syntax 34 defines the `state_dep_toggle_directive`.

```
state_dep_toggle_directive ::=
    (state_dep_toggle_directive_item
    {state_dep_toggle_directive_item}
    [COND_DEFAULT [RISE_FALL]])
state_dep_toggle_directive_item ::=
    COND cond_expr [RISE_FALL]
```

Syntax 34—state_dep_toggle_directive

A state-dependent toggle directive is a list of directive items, each followed by an optional **RISE_FALL** keyword. The item list is followed by an optional **COND_DEFAULT** keyword, which can also be followed by an optional **RISE_FALL** keyword.

The state-dependent toggle attributes generated using such a toggle directive contain switching activity for a number of the states given in the directive. The order of any states in the toggle attribute shall be the same as that in the toggle directive. The **RISE_FALL** keyword instructs the backward SAIF generator that rise and fall edges can be differentiated and state-dependent toggle attribute items with **RISE** and/or **FALL** keywords can be generated.

Example

This is an example of a state-dependent toggle directive construct:

```
(COND (A*B) RISE_FALL
COND A RISE_FALL
COND B RISE_FALL
COND_DEFAULT)
```

I.4.6 Path-dependent toggle directive

Path-dependent toggle directives instruct the backward SAIF generator on the path conditions required in path-dependent toggle attributes for cell output pins. A *path condition* is a list of input port pins. Syntax 35 defines the `path_dep_toggle_directive`.

```
path_dep_toggle_directive ::=
    (path_dep_toggle_directive_item
    {path_dep_toggle_directive_item}
    [IOPATH_DEFAULT])
path_dep_toggle_directive_item ::=
    IOPATH port_name {port_name}
```

Syntax 35—path_dep_toggle_directive

A path-dependent toggle directive is a list of directive items. The path-dependent toggle attributes generated using such a toggle directive contain switching activity for a number of the path conditions (input pin lists) given in the directive. The order of the path conditions in the toggle attribute shall be the same as that in the toggle directive.

Example

This is an example of a path-dependent toggle directive construct:

```
(IOPATH A
IOPATH B
IOPATH C D)
```

I.4.7 SDPD toggle directives

SDPD toggle directives instruct the backward SAIF generator on the state and path conditions required in SDPD toggle attributes for cell output pins. The syntax of this construct is that of the path-dependent toggle directive embedded in the state-dependent toggle directive, as shown in Syntax 36.

```
sdpd_toggle_directive ::=
    (sdpd_toggle_directive_item {sdpd_toggle_directive_item}
    [COND_DEFAULT [RISE_FALL] [path_dep_toggle_directive]])
sdpd_toggle_directive_item ::=
    COND cond_expr [RISE_FALL] [path_dep_toggle_directive]
```

Syntax 36—sdpd_toggle_directive

The SDPD toggle attributes generated using such a toggle directive contain switching activity for a number of the state and path conditions given in the directive. The order of the conditions in the toggle attribute shall be the same as that in the toggle directive.

Example

This is an example of an SDPD toggle directive construct.

```
(COND A RISE_FALL (IOPATH B)
COND B RISE_FALL (IOPATH A)
COND_DEFAULT RISE_FALL
(IOPATH A
IOPATH B
IOPATH_DEFAULT))
```

I.4.8 Module SDPD declarations

Module SDPD declarations instruct the backward SAIF generator on the type and structure of the required switching activity for particular cells. Syntax 37 defines this construct.

```
module_sdpd_declaration ::=
    (MODULE module_name {module_sdpd_directive})
module_name ::=
    identifier
module_sdpd_directive ::=
      port_declaration
    | leakage_declaration
port_declaration ::=
    (PORT port_name {port_directive})
port_directive ::=
      state_dep_toggle_directive
    | path_dep_toggle_directive
    | sdpd_toggle_directive
leakage_declaration ::=
    (LEAKAGE {state_dep_timing_directive})
```

Syntax 37—module_sdpd_declaration

The module name *identifier* represents the library cell name.

A *port declaration* assigns port directives to the individual cell pins. Port directives are either state-dependent toggle directives, path-dependent toggle directives, or SDPD toggle directives.

A *leakage declaration* consists of the **LEAKAGE** keyword followed by a state-dependent timing directive, which instructs the backward SAIF generator on the state conditions for the state-dependent timing attributes in backward leakage specifications.

Examples

This is an example of a port declaration.

```
(PORT
(A
(COND B RISE_FALL
COND_DEFAULT))


(B
(COND A RISE_FALL
COND_DEFAULT))
(Y
(COND A RISE_FALL (IOPATH B)
COND B RISE_FALL (IOPATH A)
COND_DEFAULT))
)
```

This is an example of a leakage declaration.

```
(LEAKAGE
(COND (A * B)
COND (A | B)
COND_DEFAULT)
)
```

I.4.9 Library SDPD information

The *SDPD declarations* for each library cell are listed in the library SDPD info constructs (that follow the SAIF header in the library forward SAIF file). Syntax 38 defines the `library_sdpd_info`.

library_sdpd_info ::=
  (LIBRARY *string* [*string*]
  {module_sdpd_declaration}**)**

Syntax 38—library_sdpd_info

The first *string* following the **LIBRARY** keyword represents the name of the library. The second (optional) *string* sets the path of the directory containing the library and can be used for locating it.

I.5 RTL forward SAIF file

I.5.1 Overview

The *RTF forward SAIF file* lists the synthesis invariant points of an RTL design and provides a mapping from the RTL identifiers of these design objects to their synthesized gate-level identifiers. *Synthesis invariant points* are design objects (nets, ports, etc.) in the RTL description that are mapped directly to equivalent design objects in the synthesized gate-level descriptions. Examples of such points are the design ports and RTL identifiers (variables, signals, wires, etc.) that are mapped to the outputs of sequential cells.

I.5.2 SAIF file

I.5.2.1 Overview

The RTF forward SAIF file consists of a left-parenthesis (`(`), the **SAIFILE** keyword, the RTL forward SAIF header, the RTL forward SAIF info, and a finishing right-parenthesis (`)`), as shown in Syntax 39.

```
rforward_saif_file ::=
    (SAIFILE rforward_saif_header rforward_saif_info)
```

Syntax 39—rforward_saif_file

I.5.2.2 Header

Syntax 40 defines the *RTL forward SAIF file header*.

```
rforward_saif_header ::=
    rforward_saif_version
    direction
    design_name
    date
    vendor
    program_name
    program_version
    hierarchy_divider
```

Syntax 40—rforward_saif_header

Each RTL forward SAIF header construct is described in the following subclauses.

I.5.2.3 rforward_saif_version

Syntax 41 defines the `rforward_saif_version`.

```
rforward_saif_version ::=
    (SAIFVERSION string)
```

Syntax 41—rforward_saif_version

The *string* in this construct represents the version number of the SAIF file, i.e., **2.0**.

I.5.2.4 direction

Syntax 42 defines the `direction`.

```
direction ::=
    (DIRECTION string)
```

Syntax 42—direction

The *string* in this construct represents the type of the SAIF file, i.e., **forward**.

I.5.2.5 design_name

Syntax 43 defines the `design_name`.

```
design_name ::=
    (DESIGN [string])
```

Syntax 43—design_name

The optional *string* in this construct represents the design for which the forward SAIF file has been generated.

I.5.2.6 date

Syntax 44 defines the `date`.

```
date ::=
    (DATE [string])
```

Syntax 44—date

The optional *string* in this construct represents the date the SAIF file was generated.

I.5.2.7 vendor

Syntax 45 defines the `vendor`.

```
vendor ::=
      (VENDOR [string])
```

Syntax 45—vendor

The optional *string* in this construct represents the name of the vendor whose application was used to generate the SAIF file.

I.5.2.8 program_name

Syntax 46 defines the `program_name`.

```
program_name ::=
      (PROGRAM_NAME [string])
```

Syntax 46—program_name

The optional *string* in this construct represents the name of the application used to generate the SAIF file.

I.5.2.9 program_version

Syntax 47 defines the `program_version`.

```
program_version ::=
      (PROGRAM_VERSION [string])
```

Syntax 47—program_version

The optional *string* in this construct represents the version number of the application used to generate the SAIF file.

I.5.2.10 hierarchy_divider

Syntax 48 defines the `hierarchy_divider`.

hierarchy_divider ::=
    (DIVIDER [*hchar*]**)**

Syntax 48—hierarchy_divider

The optional *hchar* in this construct represents the hierarchical separator character used in hierarchical identifiers. Only the / and **.** characters shall be specified as the hierarchical separator character; the default is the **.** character.

Example

The following is an example of a valid library forward SAIF file header:

```
(SAIFVERSION "2.0")
(DIRECTION "forward")
(DESIGN "alu")
(DATE "Fri Jan 18 11:00:00 PDT 2002")
(VENDOR "SAIFíRíUS Corp.")
(PROGRAM_NAME "rtlsaifgenerator")
(PROGRAM_VERSION "1.0")
(DIVIDER /)
```

I.5.3 Port and net mapping directives

The *port and net mapping directives* in the RTL forward SAIF file contain a list of synthesis invariant port and net identifiers and their corresponding synthesized gate-level identifiers. Syntax 49 defines these constructs.

port_mapping_directives ::=
    (PORT {**(**rtl_name mapped_name [*string*]**)**}**)**
rtl_name ::=
    hierarchical_identifier
mapped_name ::=
    hierarchical_identifier
net_mapping_directives ::=
    (NET {**(**rtl_name mapped_name**)**}**)**

Syntax 49—Port and net mapping directives

Here, the `rtl_name` is mapped into the gate-level identifier `mapped_name`. Both the RTL name and mapped name in these constructs are represented by hierarchical identifiers.

In `port_mapping_directives`, the optional *string* is used for generating virtual instance data in the backward SAIF file and represents the type of the virtual instance.

I.5.4 Instance declarations

The port and net mapping directives in the RTL forward SAIF file are organized hierarchically in RTL forward instance declarations, which comprise the RTL forward SAIF instance info that follows the header in the forward SAIF file. Syntax 50 defines the RTL forward SAIF info constructs.

```
rforward_saif_info ::=
    {rforward_instance_declaration}
rforward_instance_declaration ::=
    (INSTANCE [string] instance_name {rforward_instance_directive}
    {rforward_instance_declaration})
instance_name ::=
    hierarchical_identifier
rforward_instance_directive ::=
     port_mapping_directives
    | net_mapping_directives
```

Syntax 50—RTL forward SAIF info constructs

The *RTL forward SAIF info* is a list of instance declarations. The optional *string* following the **INSTANCE** keyword represents the design name and the *hierarchical_identifier* following it is the actual instance name. The port and net mapping directives follow the instance name. The instance declarations of any sub-design instances can be included recursively in this construct.